Université de Paris
École Doctorale 386 – Sciences Mathématiques de Paris Centre
Institut de Recherche en Informatique Fondamentale

A Semantic Foundation for Gradual Set-Theoretic Types

Thèse de doctorat en informatique

Présentée et soutenue par

Victor Lanvin

le 9 novembre 2021

devant le jury composé de

Directeur de thèse	Giuseppe Castagna	Directeur de Recherche	Université de Paris
Rapporteur	Ronald Garcia	Associate Professor	University of British Columbia
Rapporteur	Peter Thiemann	Professor	University of Freiburg
Examinatrice	Amal Ahmed	Associate Professor	Northeastern University
Examinateur	Avik Chaudhuri	PhD	Facebook
Examinateur	Erik Ernst	PhD	Google
Examinateur	François Pottier	Directeur de Recherche	INRIA Paris
Examinateur	Jeremy Siek	Professor	Indiana University

Prelude

Abstract

Title: A Semantic Foundation for Gradual Set-Theoretic Types

Keywords: Type Systems · Programming Languages · Gradual Typing · Set-Theoretic Types · Semantic Subtyping · Denotational Semantics · Type Inference · Dynamic Typing · Static Typing

In this thesis, we study the interaction between set-theoretic types and gradual typing. Set-theoretic types are types containing union, intersection, and negation connectives, which are useful to type several programming language constructs very precisely. For example, union types can be used to give a very precise type to a conditional instruction, while intersection types can encode function overloading. On the other hand, gradual typing allows a programmer to bypass the type-checker, which can be useful when prototyping.

Set-theoretic types are well-suited to a semantic-based approach called "semantic subtyping", in which types are interpreted as sets of values, and subtyping is defined as set-containment between these sets. We adopt this approach throughout the entirety of this thesis. Since set-theoretic types are characterized by their semantic properties, they can be easily embedded in existing type systems. This contrasts with gradual typing, which is an intrinsically syntactic concept since it relies on the addition of a type annotation to inform the type-checker not to perform some checks. In most of the existing literature, gradual typing is added by extending the subtyping relation in a syntactic way. This makes the approach very difficult to extend and generalize as this heavily depends on the system at hand.

In this thesis, we try and reconcile the two concepts, by proposing several semantic interpretations of gradual typing. The manuscript is divided into two parts. In the first part, we propose a new approach to integrate gradual typing in an existing static type system. The originality of this approach comes from the fact that gradual typing is added in a declarative way to the system by adding a single logical rule. As such, we do not need to revisit and modify all the existing rules. We then propose, for each declarative type system, a corresponding algorithmic type system, based on constraint solving algorithms. We illustrate this approach on two different systems. The first system is a Hindley-Milner type system without subtyping. We present a gradually-typed source language, a target language featuring dynamic type checks (or "casts"), as well as a compilation algorithm from the former to the latter. We then extend this language with set-theoretic types and subtyping on gradual set-theoretic types, and repeat this presentation.

In the second part of this manuscript, we explore a different approach to reconcile set-theoretic types and gradual typing. While the first part of the thesis can be seen as a logical approach to tackle this problem, the second part sets off along a more semantic strategy. In particular, we study whether it is possible to reconcile the interpretation of types proposed by the semantic subtyping approach and the interpretation of the terms of a language. The ultimate goal being to define a denotational semantics for a gradually-typed language. We tackle this problem

in several steps. First, we define a denotational semantics for a simple lambda-calculus with set-theoretic types, based directly on the semantic subtyping approach. This highlights several problems, which we explain and fix by adapting the approach we used. We then extend this by giving a formal denotational semantics for the functional core of CDuce, a language featuring set-theoretic types and several complex constructs, such as type-cases, overloaded functions, and non-determinism. Finally, we study a gradually-typed lambda-calculus, for which we present a denotational semantics. We also give a set-theoretic interpretation of gradual types, which allow us to derive some very powerful results about the representation of gradual types.

Résumé

Titre : Types Ensemblistes Graduels

Mots-clefs : Systèmes de types · Langages de programmation · Typage graduel · Types ensemblistes · Sous-typage sémantique · Sémantique dénotationnelle · Inférence de types · Typage dynamique · Typage statique

Cette thèse porte sur l'étude des interactions entre les types ensemblistes et le typage graduel. Les types ensemblistes sont des types proposant des connecteurs d'union, d'intersection, et de négation, qui permettent de typer des programmes de manière très fine et puissante. Par exemple, l'union permet de spécifier très précisément le type d'une instruction conditionnelle, tandis que l'intersection permet d'encoder la surcharge de fonctions dans le système de types. À l'opposé, le typage graduel permet au programmeur d'outrepasser les vérifications statiques réalisées par le système de types, afin, par exemple, d'accélérer la phase de prototypage.

Les types ensemblistes se prêtent naturellement à une approche sémantique, dans laquelle laquelle les types sont interprétés comme des ensembles de valeurs, et le sous-typage est défini comme l'inclusion ensembliste sur ces ensembles. Cette approche, dite du sous-typage sémantique, est adoptée tout au long de cette thèse. À l'inverse, le typage graduel est une notion beaucoup plus syntaxique: c'est à l'aide d'une annotation explicite que le programmeur spécifie au type-checker de ne pas réaliser de vérifications. Dans la plupart des travaux existants, la relation de sous-typage est étendue de manière ad-hoc et syntaxique pour supporter le typage graduel, ce qui la rend très rigide et peu extensible.

Dans cette thèse, nous tâchons de réconcilier les deux approches, en proposant des interprétations plus sémantiques du typage graduel. Le manuscrit est composé de deux parties. Dans la première partie, nous proposons une nouvelle approche permettant d'étendre de manière automatique un système de types avec du typage graduel. L'originalité de cette approche vient du fait que le typage graduel est ajouté de manière déclarative au système à l'aide d'une simple règle logique. Ainsi, il n'est pas nécessaire de modifier les règles existantes, comme cela est souvent fait. Nous proposons ensuite des versions algorithmiques, basées sur des algorithmes de résolution de contraintes, pour chaque système déclaratif. Nous illustrons cette approche sur deux systèmes différents. Le premier est un système de types à la Hindley-Milner sans sous-typage. Nous décrivons un langage source graduellement typé, un langage cible comportant des vérifications de type dynamiques (aussi appelées « casts »), ainsi qu'un algorithme de compilation pour passer du premier au second. Nous répétons ensuite cette présentation en étendant ce langage avec des types ensemblistes, ainsi qu'avec une relation de sous-typage sur les types graduels ensemblistes.

Dans la deuxième partie du manuscrit, nous abordons la réconciliation des types graduels et des types ensemblistes sous un autre angle. Tandis que la première partie propose une approche

logique, la seconde partie tente de fournir une approche plus sémantique de ce problème. Plus particulièrement, dans cette partie nous tentons de réconcilier l'interprétation des types proposées par l'approche du sous-typage sémantique avec l'interprétation des expressions d'un langage, ceci dans l'optique de proposer une sémantique dénotationnelle pour un langage graduellement typé. Nous attaquons ce problème en plusieurs étapes. Tout d'abord, nous proposons une sémantique dénotationnelle pour un lambda-calcul simple, basée directement sur l'approche du sous-typage sémantique. Ceci nous mène à remarquer quelques problèmes, que nous corrigeons en modifiant l'approche considérée. Nous continuons ensuite en fournissant une sémantique dénotationnelle complète pour la partie fonctionnelle du langage CDuce, un langage supportant des types ensemblistes ainsi que diverses constructions complexes (fonctions surchargées, tests de type dynamiques, non-déterminisme). Enfin, nous nous intéressons à un lambda-calcul graduellement typé, pour lequel nous proposons une sémantique dénotationnelle. Nous proposons aussi une interprétation ensembliste des types graduels qui nous mène à des résultats puissants portant sur la représentation des types graduels.

Résumé long

Titre : Types Ensemblistes Graduels

Mots-clefs : Systèmes de types · Langages de programmation · Typage graduel · Types ensemblistes · Sous-typage sémantique · Sémantique dénotationnelle · Inférence de types · Typage dynamique · Typage statique

Cette thèse porte sur l'étude des interactions entre les types ensemblistes et le typage graduel. Les types ensemblistes sont des types proposant des connecteurs d'union, d'intersection, et de négation, qui permettent de typer des programmes de manière très fine et puissante. Par exemple, l'union permet de spécifier très précisément le type d'une instruction conditionnelle, tandis que l'intersection permet d'encoder la surcharge de fonctions dans le système de types. À l'opposé, le typage graduel permet au programmeur d'outrepasser les vérifications statiques réalisées par le système de types, afin, par exemple, d'accélérer la phase de prototypage.

Les types ensemblistes se prêtent naturellement à une approche sémantique, dans laquelle les types sont interprétés comme des ensembles de valeurs, et le sous-typage est défini comme l'inclusion ensembliste sur ces ensembles. Cette approche, dite du sous-typage sémantique, est adoptée tout au long de cette thèse. À l'inverse, le typage graduel est une notion beaucoup plus syntaxique: c'est à l'aide d'une annotation explicite que le programmeur spécifie au type-checker de ne pas réaliser de vérifications. Dans la plupart des travaux existants, la relation de sous-typage est étendue de manière ad-hoc et syntaxique pour supporter le typage graduel, ce qui la rend très rigide et peu extensible.

Dans cette thèse, nous tâchons de réconcilier les deux approches, en proposant des interprétations plus sémantiques du typage graduel. Le manuscrit est composé de deux parties. La première présente une approche logique qui permet d'ajouter du typage graduel à un système existant de manière très simple. La seconde partie propose une étude plus sémantique de l'interaction entre types graduels et types ensemblistes. Le point culminant de cette partie est la définition d'une sémantique ensembliste des types graduels, qui permet de simplifier grandement l'approche présentée dans la première partie.

Une approche logique du typage graduel

Dans la première partie de ce manuscrit, nous présentons une approche dite déclarative du typage graduel. Ce travail avait initialement pour but d'intégrer le typage graduel dans un langage supportant à la fois polymorphisme, types ensemblistes, et inférence de types. Cependant, cela a débouché sur une nouvelle approche permettant d'ajouter du typage graduel à tout langage supportant des types polymorphes.

L'aspect principal de notre contribution à ce problème réside dans notre interprétation des types graduels comme des types polymorphes. Dans la littérature existante, les types graduels

sont souvent manipulés à l'aide d'une relation appelée *consistency*, notée $\sim$, définie syntaxiquement à l'aide de règles d'inférence telles que:

$$\frac{}{? \sim \tau} \qquad \frac{}{\tau \sim ?} \qquad \frac{\tau_1 \sim \tau_1' \quad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}$$

Ces règles ont la particularité de définir une relation $\sim$ non-transitive: sa fermeture transitive est la relation totale sur les types, car tout type est en relation avec ?, qui est lui-même en relation avec tout type. Ceci empêche la relation $\sim$ d'être utilisée dans une règle de *subsumption* comme le serait une règle de sous-typage, et elle doit donc être ajoutée de manière ad-hoc dans les règles d'élimination. Par exemple, les règles de typage des applications peuvent s'écrire:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau}\ \tau_1 \sim \tau_2 \qquad \frac{\Gamma \vdash e_1 : ? \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : ?}$$

Bien que cette approche fonctionne, elle est difficilement extensible. En particulier, la relation $\sim$ étant définie très syntaxiquement, il est difficile de l'étendre à des types inhéremment sémantiques, comme les types ensemblistes. Il est aussi nécessaire de modifier chaque règle de typage pour y ajouter la possibilité d'utiliser cette relation.

Afin de résoudre ces problèmes, notre approche propose d'introduire le typage graduel via une simple règle de *subsumption* définie sémantiquement. Nous définissons d'abord une opération appelée *discrimination* qui remplace chaque occurrence de ? dans un type graduel par une variable de type arbitraire. Cette opération nous permet de transformer des types graduels en des types statiques (i.e., non-graduels) polymorphes. Puis, en raisonnant en termes de substitutions de variables de type, nous définissons une opération de *précision*, notée $\preccurlyeq$, telle qu'un type τ_1 est dit être plus précis qu'un type τ_2 si le second peut-être obtenu en remplaçant des occurrences de ? dans le premier par d'autres types.

Cette relation de précision a l'avantage d'être transitive. Ainsi, elle peut être utilisée dans une règle de *subsumption* de la façon suivante:

$$[\mathrm{T}_{\mathsf{Mater}}]\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'}\ \tau \preccurlyeq \tau'$$

Nous montrons ainsi que beaucoup des systèmes de types graduels présentés dans la littérature existante peuvent être obtenus très facilement en partant d'un système de types standards (non graduels) et en y ajoutant simplement la règle $[\mathrm{T}_{\mathsf{Mater}}]$.

Pour illustrer notre approche, nous considérons dans cette partie trois systèmes de complexité croissante. Dans le Chapitre 4, nous considérons un langage à la ML, avec polymorphisme du let, mais sans sous-typage. Nous expliquons, étape par étape, comment définir le langage source et son système de types de manière déclarative, comment en déduire une présentation déclarative du langage de casts associé, puis nous abordons les aspects algorithmiques. Nous présentons notamment la sémantique opérationnelle du langage de casts, ainsi que l'algorithme d'inférence de types et de compilation permettant de passer du langage source au langage de casts.

Dans la fin du Chapitre 4, nous discutons brièvement des problèmes que l'ajout du sous-typage causerait pour l'algorithme d'inférence de types. En particulier, nous montrons qu'afin de préserver la complétude de cet algorithme en présence de sous-typage, il est nécessaire d'introduire des types ensemblistes.

Ceci nous mène au Chapitre 5, dans lequel nous ajoutons des types ensemblistes et du sous-typage sémantique à notre langage. Cependant, la simple relation de sous-typage sémantique définie en Chapitre 2 s'avère insuffisante, car elle n'est pas définie sur les types graduels. Il faut donc l'étendre à ces derniers. Pour ce faire, nous nous basons une fois de plus sur l'opération de *discrimination* que nous avons défini dans le chapitre précédent. En interprétant les types graduels comme des types statiques polymorphes, nous montrons comment réduire le problème du sous-typage des types ensemblistes graduels au sous-typage sémantique tel que défini dans le Chapitre 2. Cela nous permet, entre autres, de réutiliser tous les algorithmes de sous-typage et d'inférence de types existants.

Sémantique dénotationnelle

La seconde partie du manuscrit est dédiée à l'étude de sémantiques dénotationnelles. Bien que puissant, le formalisme développé en première partie a mis en valeur d'importantes lacunes dans notre compréhension des types graduels ensemblistes. En particulier, notre définition de la relation de précision reste fondamentalement syntaxique, ce qui se ressent particulièrement lors de la définition de la sémantique opérationnelle du langage de casts présenté en Chapitre 5. De nombreux invariants syntaxiques doivent être préservés par réduction, ce qui rend les règles de réduction particulièrement complexes.

En étudiant les types graduels ensemblistes d'un point de vue dénotationnel, nous espérions pouvoir obtenir une interprétation ensembliste de ces types, et en déduire des définitions sémantiques des relations de sous-typage et de précision. Cependant, définir une sémantique dénotationnelle pour un langage graduellement typé avec types ensemblistes s'est avéré être une tâche particulièrement complexe. Ainsi, nous avons procédé en plusieurs étapes de difficulté croissante.

Tout d'abord, dans le Chapitre 9, nous considérons des types ensemblistes non-graduels, et nous définissons une sémantique dénotationnelle pour un lambda-calcul simple équipé de ces types. L'importance de cette sémantique réside dans le fait que les termes du langage et les types sont interprétés dans le même domaine. Ainsi, en reliant l'interprétation des types et les valeurs d'un langage, ce chapitre apporte une brique manquante à l'édifice du sous-typage sémantique.

Tout n'est pas cependant parfait : le domaine d'interprétation sus-cité est malheureusement trop limité pour interpréter parfaitement certains termes dénotationnellement. Ainsi, dans le Chapitre 10, nous modifions ce domaine afin de résoudre ce problème. Cela nous permet de corriger notre sémantique dénotationnelle, et nous prouvons qu'elle est à la fois correcte et adéquate, ce qui prouve son exactitude. Nous prouvons de plus que la relation de sous-typage induite par la nouvelle interprétation des types sur ce domaine est exactement identique à celle induite par l'approche du sous-typage sémantique. Ainsi, les conclusions du chapitre précédent restent valables malgré les modifications réalisées.

Dans le Chapitre 11, nous étendons ensuite notre sémantique au langage ℂDuce . Pour ce faire, nous ajoutons trois éléments principaux: la dérivation de types intersections pour les fonctions, une forme de choix non déterministe, et des tests de types dynamiques. Chacun de ces éléments nécessite une modification supplémentaire du domaine d'interprétation des types et des termes du langage.

Il est relativement aisé d'adapter la sémantique dénotationnelle du chapitre précédent afin de supporter le choix non déterministe ainsi que les tests de types dynamiques. En revanche, l'inférence de types intersections pour les fonctions est plus problématique. Notre solution nécessite l'ajout de nouveaux éléments au domaine d'interprétation des types et des termes, ainsi

qu'une modification des règles de typage du langage ℂDuce afin de ne pas autoriser l'inférence de négations de types arbitraires. Nous parvenons alors, avec ces modifications, à définir une sémantique dénotationnelle correcte pour le langage ℂDuce modifié. De plus, à la fin du Chapitre 11, nous démontrons que les modifications apportées à ℂDuce ne sont finalement que peu contraignantes : nous montrons en effet qu'il est possible, pour chaque terme du langage ℂDuce , de trouver un terme de notre langage qui se comporte exactement de la même façon par bisimulation. Ainsi, le formalisme développé dans ce chapitre reste compatible avec le langage ℂDuce .

La sémantique des types graduels

Après avoir développé une sémantique dénotationnelle pour ℂDuce , nous revenons à notre objectif initial : celui de définir une sémantique ensembliste des types graduels, ainsi qu'une sémantique dénotationnelle pour un langage de casts. Cette étude est présentée dans le Chapitre 12. Pour des raisons de simplicité, la sémantique présentée dans ce chapitre est basée sur la sémantique présentée en Chapitre 10, et ne comporte donc pas les concepts plus avancés présentés en Chapitre 11.

Nous commençons ce chapitre par la définition d'une sémantique ensembliste des types graduels ensemblistes. L'idée principale derrière cette sémantique est de distinguer les valeurs appartenant *certainement* à un type des valeurs appartenant *possiblement* à un type, sous réserve que les occurrences du type dynamique ? dans ce type soient résolues vers le bon type statique. Ainsi, par exemple, le type ? contient *possiblement* toutes les valeurs, mais n'en contient aucune certainement. À l'inverse, le type Int des entiers contient certainement toutes les valeurs entières, mais n'en contient aucune autre. Nous distinguons entre les deux notions (*possiblement* et *certainement*) à l'aide d'un « tag » que nous ajoutons aux éléments du domaine d'interprétation.

L'interprétation des types ainsi obtenue est ensembliste : le type $\tau_1 \vee \tau_2$ est interprété comme l'union ensembliste des interprétations de τ_1 et τ_2, et il en va de même pour les types intersections et négations. Ainsi, en définissant la relation de sous-typage sur les types graduels ensemblistes comme l'inclusion ensembliste de leurs interprétations (de la même manière que pour le sous-typage sémantique), nous obtenons une relation qui satisfait de très fortes propriétés.

De plus, à l'aide de cette interprétation des types, nous parvenons aussi à définir une relation de précision de manière ensembliste, plutôt que syntaxique. Nous montrons que les deux relations sont en réalité très liées, et qu'elles peuvent toutes deux être exprimées en utilisant uniquement la relation de sous-typage sémantique définie sur les types statiques. De plus, nous montrons un résultat particulièrement puissant sur la représentation des types graduels ensemblistes, prouvant que tout type peut être représenté de manière équivalente en utilisant une unique occurrence du type dynamique ?.

Pour conclure le Chapitre 12, nous présentons une sémantique dénotationnelle pour un langage de casts. En raison de la complexité de cette tâche, nous nous limitons à un langage avec types simples. Cependant, nous présentons quelques contributions importantes comme la définition de l'action d'un cast sur un élément du domaine d'interprétation, ce qui induit une interprétation ensembliste des casts du langage.

Enfin, nous revenons ensuite en arrière pour appliquer nos nouveaux résultats au travail présenté dans la première partie du manuscrit. Ce développement est présenté dans le Chapitre 6. Nous reprenons le langage présenté dans le Chapitre 5, mais nous introduisons les relations de sous-typage et de précision sémantiques définies dans le Chapitre 12. À l'aide de notre résultat sur la représentation des types graduels, nous réussissons à grandement simplifier la sémantique opérationnelle du langage de casts présenté dans le Chapitre 5. De plus, nous montrons qu'il

est possible de réutiliser aisément beaucoup de résultats provenant de la théorie du sous-typage sémantique pour prouver la correction de notre sémantique.

Remerciements & Acknowledgements

"Moi, si je devais résumer ma vie aujourd'hui avec vous, je dirais que c'est d'abord des rencontres [...]"

Otis, *Astérix et Obélix mission Cléopâtre*

First of all, I would like to give my warmest thanks to my reviewers, Ron Garcia and Peter Thiemann, for accepting to read this (long) manuscript, and for all their amazing feedback. Many thanks also to my examiners, Amal Ahmed, Avik Chaudhuri, Erik Ernst, François Pottier, and Jeremy Siek, for accepting to be part of the jury. I am very honoured by the attention you have given to my work.

Tout d'abord, à toi Giuseppe, pour ton encadrement et ton soutien sans faille durant ces six dernières années. Ce qui devait au départ n'être qu'un simple sujet de stage de M2 s'est révélé être aussi complexe que passionnant, et ma curiosité n'a jamais faibli. Tu es et restera un modèle pour moi, pour ta pédagogie, et pour ta capacité à faire preuve d'une incroyable psychologie même dans les moments les plus difficiles.

À toutes les personnes qui font de l'IRIF un superbe environnement, tant sur le plan humain que professionnel. Parce qu'on n'est jamais à l'abri d'une catégorie cartésienne close au détour d'un article apparemment innocent, il est toujours réconfortant de savoir que l'on peut appeler à l'aide dans le bureau d'à côté. Et pourquoi pas, au passage, aller prendre deux ou trois parts d'un délicieux gâteau, et finir la discussion autour d'une bière.[1] L'article peut attendre, après tout.

À toi Nicolas, pour avoir su égayer même le plus sombre des placards à stagiaires, et pour ce stage de M2 qui n'aurait clairement pas été aussi productif sans toi. À toi Niols, pour avoir pris la relève de Nicolas quelques années plus tard en devenant le cobureau et ami idéal. Pour toutes nos aventures plus tordues les unes que les autres, mais toujours riches en émotions.[2]

À toi Tommaso, pour avoir pris le temps de m'expliquer tous ces concepts qui me paraissaient si obscurs à l'époque, et pour avoir toujours fait preuve d'une patience infinie lorsqu'il s'agissait d'écouter mes théories les plus absurdes.

À toutes les autres personnes avec qui j'ai eu la chance de partager ce magnifique bureau 3035. Jovana, pour ta patience envers Niols et moi qui travaillions parfois très intensivement jusque très tard. Étienne, pour ta bonne humeur permanente et ton expertise, que ce soit pour LaTeX ou pour trouver le formulaire n°45277 de demande de d'extrait de contrat doctoral. Adrien, pour nos moments passés au bar à monopoliser trois tables[3], parce qu'il faut bien ça, pour terraformer Mars. Pierre, pour ces moments très agréables passés à gérer un cours d'IP1.

À tou·te·s les autres doctorant·e·s, stagiaires, et post-doctorant·e·s que j'ai pu fréquenter pendant ces quelques années. Alexandra, Antoine, Baptiste, Cédric, Chait, Clément, Farzad, Guillaume, Léo, Léonard, Marie, Mickaël, Rémi, Théo, Thibaut, Yann, Zeinab (et j'en oublie probablement), pour la bonne ambiance que vous avez toujours faite régner au labo, et pour avoir lutté,

[1] Et repartir sans payer.
[2] Si on pouvait éviter de trop finir au commissariat quand même, ça m'irait.
[3] Sans rien payer, encore une fois.

de près ou de loin, contre l'invasion de la salle détente.[4]

À tous les autres membres de l'IRIF, tous plus formidables les uns que les autres. Juliusz, Laurent, pour toutes vos visites dans le bureau et vos innombrables anecdotes.[5] Arnaud, Ralf, Yann, pour avoir joué un grand rôle dans la naissance de ma passion pour l'enseignement. Dieneba, Étienne, Natalia, Odile, qui n'avez jamais hésité à venir à mon secours au plus profond des méandres administratives.[6]

À tou·te·s mes étudiant·e·s, à qui j'ai toujours adoré enseigner, et qui, en retour, ont toujours su me transmettre une énergie et un bonheur formidables. À vous deux, Asma et Inès, pour avoir accepté de participer à un crash test en étant mes premières stagiaires. Alex, Alexandre, Benoit, Célia, Constance, Ewen[7], Fred, Julia, Léia, Léna, Marie, Marie, Marie[8], Marine, Marylin, Maxime, Miguel, Pierre, Pierre, Raphaëlle, vous m'avez tou·te·s marqué d'une manière ou d'une autre, que ce soit par des discussions, des rires, ou des questions tordues.

À toute ma (grande) famille, à qui je dois énormément. Je suis incroyablement chanceux de vous avoir. Marie, Valentine, Papa, Maman, je vous aime. À tou·te·s mes cousins, cousines, oncles, tantes, et grands-parents, vous m'avez tou·te·s tant apporté. J'ai hâte de vous revoir lors de gigantesques cousinades... Et probablement lors de quelques mariages ?

À mes ami·e·s que je n'ai pas cité·e·s jusque-là, vous qui m'avez toujours soutenu, et qui faites partie de mes plus précieux souvenirs.

À toutes ces merveilleuses personnes rencontrées à l'ENS. En particulier Théo, pour ces heures passées à afficher des théières en 3D plutôt qu'à écouter les cours. Charlie, Félicien, François, Itsaka, Pierre-Léo, Mathias, Mathieu, Olivier, Pierre, Théis, Thomas, pour le spam qui n'a jamais faibli depuis 2013. Mickaël, pour m'avoir agréablement tenu compagnie pendant mon petit détour de quatrième année.

À vous, Alexandre, Étienne, Marie, Pierre, pour toutes ces découvertes ludiques et vidéoludiques, ces soirées à la Felicita, et ces longues marches nocturnes. J'ai hâte de remettre ça.

À tou·te·s mes ami·e·s de la danse, et au Paris Band pour tous ces incroyables moments musicaux, et ces raclettes parfois estivales. Caroline, pour notre amour commun des voyages au marché effrénés. Pascaline, pour ton attention et ta bienveillance constantes. Chloé, pour ce Clair de Lune qu'il faudrait jouer, un jour. Aymeric, pour toutes ces dégustations de Porto. Shannon, pour tous ces « un jour, on la fera, cette rouge ».

À toi Linéa, qui a toujours été là pour me ramasser à la petite cuillère et recoller les morceaux, que ce soit pendant la rédaction ou après une chute à l'escalade.[9] À toi Louise, pour ton soutien inconditionnel pendant cette rédaction, et pour nos longues heures passées à discuter ou à travailler en écoutant le dernier album de PSVP.

Enfin, à toi Anne, pour tant de choses. Pour nos rires, nos larmes, nos souvenirs, et notre futur. Tout ça n'aurait jamais été possible sans toi.

À toutes ces personnes, et toutes celles que j'oublie,

[4] Malheureusement, nous n'avons pas résisté encore et toujours à l'envahisseur.
[5] Et pour ces batailles par Dazibao interposé.
[6] Et ce, même après mon 8ème avenant. Désolé de vous en avoir fait voir de toutes les couleurs.
[7] T'as assisté à assez de mes cours pour être dans cette catégorie, non ?
[8] Ordonnées par ordre alphabétique de nom de famille.
[9] Et bienvenue dans la secte.

Merci.

Contents

List of symbols and definitions

We list here the main definitions present throughout this thesis. For each of them, we state its name, the associated symbol if there is any, and point to where it is first introduced.

Background

Part I

Part II

Introduction

Chapter 1.

Introduction

"The beginning is the most important part of any work."

PLATO, *Republic*

The original goal at the start of this thesis was to study the interaction between *set-theoretic types* and *gradual typing*. Set-theoretic types are types containing union, intersection, and negation connectives, which are useful to type several programming language constructs very precisely. For example, union types can be used to give a very precise type to a conditional instruction, while intersection types can encode function overloading. On the other hand, gradual typing allows a programmer to bypass the type-checker, which can be useful when prototyping, to speed up the development process.

Throughout this manuscript, we build on the *semantic subtyping* approach of Frisch et al. [27], which we extend in several directions: we design a gradually-typed language with polymorphic set-theoretic types, we connect the model of values of Frisch et al. [27] to the semantics of a language, and we extend the semantic subtyping approach to gradual types.

1.1. Problem and motivations

At some point in their early days, every programmer comes to a harsh realization: computers *are not* smart. They are simply machines that are, unless explicitly programmed to, devoid of any ability to interpret and analyze the meaning of their actions. Over time, a programmer goes from asking "Why is this computer not doing what I want?" to wondering "Where is my code wrong?".

Bugs are everywhere. At best, they are merely a nuisance, requiring a few hours of work to find and fix. At worst, they can cost billions in damages. Hence, much research is pursued towards making programming *safer*, *faster*, and less *error-prone*. Part of this research focuses on *type systems* and the development of *type checkers*, which are tools that analyze a program before it is even executed (we say they perform checks *statically*, as opposed to *dynamically*), to ensure that certain conditions are met (for example, that one is not trying to subtract one from a word).

There are often three main properties that are considered when designing a type checker:

Expressiveness: is the type system able to accurately characterize the behaviour of a program, while not hindering its capabilities?

Safety: which errors is the type system able to catch, and which ones are prone to produce false negatives or false positives?

Programming efficiency: does the type system require a lot of additional work from the programmer, or is it easy to use?

Unfortunately, these three properties are closely tied together: stronger and safer type systems are also often more difficult to use, while efficient and flexible systems are often less safe. The type system of the language ℂDuce [1], which is based on the semantic subtyping approach and will be omnipresent throughout this manuscript, arguably falls into the first category. While it allows the programmer to type several constructs very precisely, thus accurately describing the behaviour of a program, it lacks any form of type inference, which means that the programmer has to write many long and cumbersome type annotations.

Recently, efforts have been made to reconcile dynamically-typed languages (languages that do not feature a static type-checker) and statically-typed languages, in the hope of obtaining a language that can be as safe as the programmer wants it to be, while keeping the possibility of disabling the type-checker at any time and at any point of a program, should it prove to be a hindrance. An implementation of this idea is called *gradual typing*. In a gradually-typed language, a programmer can, via the help of a specific type annotation, tune the precision of the type checker on any part of a program. They can also *gradually* introduce more type annotations, to make the program safer after the prototyping phase is over.

In this thesis, we study systems that incorporate both set-theoretic types for their expressiveness and gradual typing for its ease of use, with the goal of obtaining a powerful but flexible type system.

1.1.1. Set-theoretic types

Set-theoretic types are types containing union ($\vee$), intersection ($\wedge$), and negation ($\neg$) connectives. These connectives can be understood as follows:

> *Union:* the union of two types t_1 and t_2, which we write $t_1 \vee t_2$, can be given to values that have *either* type t_1 *or* type t_2;
>
> *Intersection:* the intersection of two types t_1 and t_2, written $t_1 \wedge t_2$, can be given to values that have *both* types t_1 and t_2;
>
> *Negation:* the negation of a type t, written $\neg t$, can be given to values that do not have type t.

Naturally, as we will discuss throughout this manuscript, set-theoretic types can be enriched with various features, such as polymorphism (by adding type variables), recursive types, or gradual types (by adding a dynamic type).

The presence of set-theoretic connectives greatly improves the expressiveness of types, and allows for many programming idioms to be typed very precisely. We present some examples here.

Union types

The first and arguably most natural application of set-theoretic connectives comes from union types and, in particular, conditional expressions. Consider the expression `if b then 1 else false`, and assume that the language we consider does not allow implicit conversions between integers and booleans. In the absence of union types, this leaves two possibilities: either this expression is simply ill-typed because we cannot prove that it always returns an integer or always returns a boolean, or we can give this expression some kind of top type (for example `Any`), which means we lose all the static type information present in the program. However, if our language supports

union types, we can give this expression the type Int ∨ Bool, since it is clear that is either returns an integer *or* a boolean.

More generally, if an expression e_1 has type t_1 and an expression e_2 has type t_2, then the expression `if b then` e_1 `else` e_2 has type $t_1 \vee t_2$. Note that this applies to most heterogeneous data structures: for example, the list [3; *true*], which could be given type `Any list` in a language with a top type but no union connective, can be typed as (Int ∨ Bool) `list` using a union type.

This expressiveness makes union types particularly useful when designing a type system for an originally untyped language. As such, they have been added to several languages such as TypeScript (Microsoft [49]), Flow (Facebook [23]), and Typed Racket (Tobin-Hochstadt and Felleisen [72]).

Overloaded functions

As we explained above, intersection connectives can be used to assign two different types to the same expression. While this is of limited use for constants, this is particularly useful for functions. Suppose, for example, that we want to convert booleans to integers (using the canonical interpretation that 0 corresponds to `false` and every other integer to `true`) and vice-versa. In a language without overloading, the standard way of doing this would be to provide two functions `intToBool` and `boolToInt` that have respectively types Int → Bool and Bool → Int. However, if the language supports it, we can define an overloaded function `convertIntBool` that does both conversions, depending on the type of its argument. Such a function can be applied to and can return both integers and booleans. Using union types, this means that this function can be assigned type (Int ∨ Bool) → (Int ∨ Bool). However, this is not satisfactory: this type only ensures that, if we apply this function to an integer, then the result is of type Int ∨ Bool.

We can get a finer type by using an intersection connective. Since this function returns a boolean whenever it is applied to an integer, and returns an integer whenever it is applied to a boolean, it has both types Int → Bool *and* Bool → Int. It can therefore be assigned the intersection of the two: (Int → Bool) ∧ (Bool → Int). This perfectly represents the behaviour of the function: not only does this intersection type ensures that the function can only be applied to integers *or* booleans, it also induces a correspondence between the type of the argument and the type of the result, which the type (Int ∨ Bool) → (Int ∨ Bool) does not. In fact, as we will discuss in Subsection 1.1.2, this hints at the fact that (Int → Bool) ∧ (Bool → Int) is a subtype of (Int ∨ Bool) → (Int ∨ Bool), and thus that every value of the former type (including the function `convertIntBool`) also has the latter type by subtyping. The converse, naturally, does not hold. Consider for example the identity function that can be applied to both integers and booleans: since it maps integers *or* booleans to integers *or* booleans, it can be given type (Int∨Bool) → (Int∨Bool). However, since it maps integers to integers and booleans to booleans, it can be given type (Int → Int) ∧ (Bool → Bool) but not (Int → Bool) ∧ (Bool → Int).

Notice that we have not yet discussed *how* to implement the function `convertIntBool`. This is because, for set-theoretic types to be truly useful, the language must support some form of dynamic type checking. In our case, if the exact type of the argument of the function `convertIntBool` cannot be resolved statically (as would be the case with a non-deterministic expression for example), the language needs to perform a check to decide whether it should be converted to a boolean or to an integer. In practice, as we will discuss in the second part of this manuscript, we introduce an expression called a *typecase*, of the form $(x = e \in t)?\ e_1 : e_2$, which binds the result of e to x, and evaluates e_1 if it is of type t, or evaluates e_2 if it is not. Using such a construct, our conversion

function can be implemented as $\lambda x{:}\mathsf{Int} \vee \mathsf{Bool}.\ (y = x \in \mathsf{Int})?\ \texttt{intToBool}(y) : \texttt{boolToInt}(y)$. [1]

Negation types

With the introduction of typecases comes the problem of type-checking such expressions. We have already seen that conditional expressions can be typed very precisely using union types. However, typecases differ slightly from standard `if then else` constructs as they introduce a new variable whose type depends on the test. For example, for our above implementation of the function `convertIntBool` to be well-typed, y must have type Int in the first branch ($\texttt{intToBool}(y)$) and type Bool in the second branch ($\texttt{boolToInt}(y)$).

Verifying the former condition is easy: since we only execute the first branch if the test succeeds, that is, if x has type Int, we know that y (which has been bound to the value of x) necessarily has type Int in the first branch. Verifying the second condition is more difficult, and requires some additional reasoning. The second branch is only executed if the test fails. This ensures that, in this branch, y *does not* have type Int. This is where negation types come into play, as we can type the second branch under the hypothesis that y has type $\neg\mathsf{Int}$. This information alone is not sufficient however. We need to also use the fact that x (and thus y) also has type $\mathsf{Int} \vee \mathsf{Bool}$, as guaranteed by the annotation of the function. Since y has both types $\mathsf{Int} \vee \mathsf{Bool}$ and $\neg\mathsf{Int}$, we can assign it the intersection of the two, which is $(\mathsf{Int} \vee \mathsf{Bool}) \wedge \neg\mathsf{Int}$, or, introducing some standard set-theoretic syntax, $(\mathsf{Int} \vee \mathsf{Bool}) \setminus \mathsf{Int}$. Under this new refined hypothesis, the second branch is now well-typed since $(\mathsf{Int} \vee \mathsf{Bool}) \setminus \mathsf{Int}$ is indeed equivalent to Bool.

From a more general point of view, negation types associated to intersection types can be used to formalize *occurrence typing* (Tobin-Hochstadt and Felleisen [73], Pearce [55], Chaudhuri et al. [20]), which is a method of refining types according to conditionals. If an expression e has some type t, and we dynamically check whether it has type t', then it can be assumed that e has type $t \wedge t'$ when the test succeeds, and $t \setminus t'$ when the test does not.

In general, whether it is based on set-theoretic types or not, occurrence typing has been added to many programming languages. Some examples include Flow, Typed Racket, TypeScript, Kotlin (JetBrains [42]), and Ceylon (King [45]).

Recursive types

Another addition to set-theoretic types we will consider throughout this manuscript is recursive types. Recursive types, often represented using an explicit binder for recursion denoted μ, can be a powerful tool to encode recursive data structures without the need for inductive types and variants. For example, in OCaml, the type of lists of integers can be declared as the inductive type `intlist = Emptylist | Cons of (int * intlist)`, which must then be deconstructed by pattern matching. Using recursive set-theoretic types equipped with product types, the type of lists of integers can instead be defined as $\mu X.(\mathsf{Unit} \vee (\mathsf{Int} \times X))$. The intuition being that a value of this type is either of type Unit (the empty list) or a pair of an integer and another value of this type (a list of at least one integer). Rather than being deconstructed via pattern-matching, this type is deconstructed as any union type, that is, using a typecase construct: checking whether a list `l` is empty or contains at least one integer can be done by checking whether `l` has type Unit or not.

[1] In practice, inferring the type $(\mathsf{Int} \to \mathsf{Bool}) \wedge (\mathsf{Bool} \to \mathsf{Int})$ for this function is a difficult problem, and we rather ask the programmer to explicitly annotate the function with the full intersection type. We explore a possible solution in [19] which uses a technique called *occurrence typing*. However, this work is outside the scope of this manuscript and will not be presented here.

From a more theoretical point of view, recursive types are also interesting insofar as they allow the definition of fixed-point combinators. The peculiarity of fixed-point combinators is that they often rely on the definition of a function that can be applied to itself an arbitrary number of times. In most settings, such an operation is not well-typed: if a function has type $t \to t$, then it can only be applied to itself if $t \to t$ is a subtype of t, which is problematic unless t is the top type (which limits the usefulness of such an operator). Using recursive types, however, it is clear that a function of type $\mu X.X \to t$ can be applied to itself an arbitrary number of times.

To illustrate this, consider the following code in ℂDuce :

```
type X = X -> Int -> Int

let fun Z ( f : (Int -> Int) -> Int -> Int ) : Int -> Int =
    let fun Delta ( x : X ) : Int -> Int =
        f ( fun (Int -> Int) v ->( x x v ))
     in Delta Delta;;

let fun fact ( f : Int -> Int) : (Int -> Int) =
    (fun (x: Int) : Int =
        if x = 1 then 1 else (x * (f (x - 1))));;

let factorial = Z fact;;
```

This code defines the factorial function by declaring a fixed-point combinator `Z` and applying it to the function `fact`, of type $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int} \to \mathsf{Int}$, which simply performs a single iteration of the factorial. The crux of this definition lies in the declaration of the function `Delta` of type $\mu X.X \to \mathsf{Int} \to \mathsf{Int}$, which follows the definition of Curry's fixed-point combinator `Y`.

Polymorphic types

The last aspect we discuss here is polymorphism. Until now, we have only considered monomorphic types, which limits their expressiveness. For example, we have shown how recursive set-theoretic types can be used to encode the type of lists of integers using product types. However, we currently have no way of encoding the type of homogeneous lists of any type. If we know in advance that our lists will be of a finite number of types, for example if we only consider lists of booleans and lists of integers, then we can still use union types to encode the type of our lists, for example as `intlist` ∨ `boollist`. However, many functions act on lists of any type (e.g., the function `map` which applies a function `f` to every element of a list `l` to obtain a second list `l'`). We could represent the type of all homogeneous lists using an infinite union, as $\bigvee_t \mu X.(\mathsf{Unit} \vee (t \times X))$, and the function `map` could be typed using the infinite intersection $\bigwedge_{t,t'} t$ `list` $\to (t \to t') \to t'$ `list` (using t `list` as syntactic sugar for the type of lists of type t).

However, this approach presents a major problem. To ensure types are well-founded and that subtyping is decidable, we prohibit infinite unions and intersections.[2] The solution is to introduce polymorphic types, which can finitely represent such types. The type of homogeneous lists of any type becomes α `list` $= \mu X.(\mathsf{Unit} \vee (\alpha \times X))$, and the type of `map` becomes $\forall \alpha, \beta.\ \alpha$ `list` $\to (\alpha \to \beta) \to \beta$ `list`.

Together with set-theoretic types, polymorphic types can be used to great effect. For example, Castagna et al. [15] show that they can be used to type the function that inserts a new node in

[2] This is the *contractivity* property we will introduce in Chapter 2.

a red-black tree in a very precise way, without changing the original implementation of Okasaki [53], enforcing statically three out of the four invariants of red-black trees. Namely, that the root of the tree is black, that the leaves are black, and that no red node has a red child. The only missing invariant being that every path from the root to a leaf should contain the same number of black nodes. The resulting type of the function is:

$$\forall \alpha, \beta. (\alpha \texttt{ Unbal} \to \alpha \texttt{ RTree}) \wedge (\beta \setminus (\alpha \texttt{ Unbal}) \to \beta \setminus (\alpha \texttt{ Unbal}))$$

It uses many of the features presented so far to state that the function maps unbalanced trees of elements of type α to red-rooted balanced trees, and leaves everything else unchanged.

1.1.2. Subtyping on set-theoretic types

We have presented set-theoretic types and given many examples of their expressiveness, but we have glossed over a central aspect of type-checking: subtyping. Without subtyping, set-theoretic types lose most of their usefulness. Suppose, as is customary, that we type applications using a simple *modus ponens* rule, that is, the application of a function `f` to a value `v` is well-typed if and only if `f` has some type $t \to t'$ and `v` has type t. If `f` is an overloaded function, for example of type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$, and `v` is an integer, then for the application $f\,v$ to be well-typed, we must be able to deduce that `f` has type $\mathsf{Int} \to \mathsf{Int}$ or type $(\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \vee \mathsf{Bool})$. This is where subtyping comes into play.

The limitations of a syntactic approach

As anticipated, we naturally expect the subtyping relation to satisfy several properties on types, inspired by set theory. For example, we expect the intersection to be distributive over the union: the two types $t \wedge (t_1 \vee t_2)$ and $(t \wedge t_1) \vee (t \wedge t_2)$ must be, intuitively, equivalent. Here, equivalent means that they can be used interchangeably. That is, from a subtyping point of view, they must be subtype of one another. Similarly, when discussing occurrence typing in the previous subsection, we stated that the type $(\mathsf{Int} \vee \mathsf{Bool}) \wedge \neg\mathsf{Int}$ is equivalent to Bool, glossing over the formalism that allows such a deduction. To prove this equivalence, distributivity alone is not sufficient: one must also deduce that $\mathsf{Int} \wedge \neg\mathsf{Int}$ is equivalent to the empty type $\mathbb{0}$, that $\mathsf{Bool} \wedge \neg\mathsf{Int}$ is equivalent to Bool, and that $\mathbb{0} \vee \mathsf{Bool}$ is itself equivalent to Bool.

As this explanation indicates, defining a suitable subtyping relation on set-theoretic types using syntactic deduction rules is a complex task, as many corner cases must be taken into account. Such definitions are often correct but incomplete, in the sense that they properly reject unsafe programs, but often also cause safe programs to be rejected. This, of course, becomes even more complicated when introducing type constructors. For example, as we hinted before, the type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$ is, intuitively, a subtype of $(\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \vee \mathsf{Bool})$, as every function that maps integers to integers and booleans to booleans can also be seen as a function that maps booleans or integers to booleans or integers, although we lose some information in the process.

Semantic subtyping

As one may have guessed, a semantic approach is, arguably, better suited than a syntactic approach to define subtyping on set-theoretic types. This realization led to the theory of *semantic subtyping* of Frisch et al. [27], which we will use throughout this manuscript. The central idea behind semantic subtyping is that types can be seen as sets of values. For example, Int denotes

the set of all integers, while $\mathsf{Int} \to \mathsf{Int}$ denotes the set of all functions mapping integers to integers.[3] Type connectives can then be interpreted following their set-theoretic counterparts. For example, the type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$ denotes the values belonging to both the sets denoted by $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Bool}$, which corresponds to the functions that map integers to integers *and* booleans to booleans, as anticipated.

The main advantage of semantic subtyping is that subtyping immediately follows from the interpretation of types as sets of values. We can say that a type t is a subtype of a type t' if the set of values denoted by t is a subset of the set of values denoted by t'. Mathematically, it is clear that $S_1 \subseteq S_2$ holds if and only if $S_1 \wedge \overline{S_2}$ is empty (where $\overline{S_2}$ denotes the set-theoretic complement of S_2). Going back to set-theoretic types, we obtain that t is a subtype of t' if and only if $t \wedge \neg t'$ is empty. This simple and intuitive definition of subtyping, which reduces to checking whether a type is empty or not, is the very basis of semantic subtyping as presented by Frisch et al. [27]. In Chapter 2, we will give more details into how to interpret types as sets of values, and how to define formally this subtyping relation.

1.1.3. Gradual typing

Gradual typing is an approach introduced by Siek and Taha [65] that aims at integrating both static typing and dynamic typing in the same framework. The core idea is simple: they add a dynamic type, often denoted $?$, to the syntax of types. As its name suggests, this type informs the type-checker that it must not be checked statically: a value that is given type $?$ statically can be used in any context, and, for example, can be passed to a function expecting an integer.

This is not an *all or nothing* approach: by building types using the dynamic type and the usual type constructors, a programmer can *gradually* tune the behaviour of the static type-checker. For example, a value of type $? \to ?$ is, statically, a function: if one tries to pass it to a function expecting an integer, for example, the type-checker will fail (as a function is not an integer). However, this is the only information this type provides. Such a value can be applied to anything, and can return anything.

Naturally, such flexibility comes at a cost. To avoid undefined behaviour (e.g., if one uses a dynamic type annotation to disable the type-checker and passes a boolean to a function expecting an integer), the checks that are not done statically must be performed dynamically, that is, during the execution of the program. This not only incurs a performance cost, but it also adds a compilation step whose role is to insert dynamic checks (or "casts") into the program. For example, one can write the following code:

$$(\lambda x{:}?.\, x + 1)\texttt{true}$$

This code consists in a function that adds one to its argument, and since its parameter is explicitly given type $?$, it can be applied to anything. Here, we apply it to `true`. While it is clear that such a program is incorrect, it is, nevertheless, accepted by the type-checker. The compilation step is here to ensure that this program will be rejected dynamically, by producing the following compiled code:

$$(\lambda x{:}?.\, x\langle\mathsf{Int}\rangle + 1)\texttt{true}$$

Here, the expression $x\langle\mathsf{Int}\rangle$ is a cast which checks that the value of x is an integer, and fails if it is not the case. At runtime, this code will then reduce to $\texttt{true}\langle\mathsf{Int}\rangle + 1$, and since `true` is not an integer, this will report an error to the programmer.

[3] Note that this does not state anything about the behaviour of the functions on inputs that are not integers. That is, a function of type $\mathsf{Int} \to \mathsf{Int}$ could also map booleans to anything.

Throughout this manuscript, we will study *gradual set-theoretic types*, that is, set-theoretic types which can also contain occurrences of the dynamic type. The interaction between set-theoretic connectives and the dynamic type leads to many interesting problems and programming idioms, which will be detailed later in Chapter 3. This raises questions such as "What is the meaning of $? \vee \mathsf{Int}$? What about $? \wedge \mathsf{Int}$?". At first, one may think that saying that a value is of type $? \vee \mathsf{Int}$ is basically the same as saying that it is of type $?$, since $?$ contains strictly less information than Int, and the information carried by the latter is "erased" by the union connective. However, we will see that the reality is more complex than this, since a value of type $? \vee \mathsf{Int}$ *cannot* be used in the same contexts as a value of type $?$.

1.2. Our contributions

Throughout this manuscript, we focus on set-theoretic types and semantic subtyping, and their interactions with gradual typing. This leads us to consider and study various features of functional programming languages, such as let-polymorphism, type inference, non-determinism, and typecases.

While, at first glance, set-theoretic types may seem complicated to use and formalize, we argue that, not only do they allow us to obtain very expressive type systems, they also form a mutually beneficial relationship with gradual types. On one hand, gradual types can help alleviating the syntactic overhead associated with set-theoretic types. On the other hand, set-theoretic connectives make the transition between dynamic typing and static typing smoother and finer grained, giving even more control to the programmer. We go even further by showing that the very meaning of gradual types can be formalized using the semantic subtyping approach, and that, in the presence of set-theoretic types, gradual types obtain strong properties which allow us to directly reuse many of the existing results on semantic subtyping.

This manuscript is separated into two parts, which focus on different approaches to gradual set-theoretic types. The first part features a *logical* approach to gradual typing, in which we propose to add gradual typing to existing type systems using a simple logical rule. The second part focuses on the denotational semantics associated with semantic subtyping, with the goal of unearthing the semantic meaning of gradual types. This work culminates with the definition of a set-theoretic interpretation of gradual types, which is featured in the last chapters of both parts.

1.2.1. A logical approach to gradual typing

In Part I of this thesis, we describe a new approach to endow a static type system with gradual typing, and apply this approach to several type systems, both with and without subtyping and set-theoretic types.

The novelty of this approach is that, from a declarative point of view, gradual typing can be added by including a single rule to the static type system. This rule is a subsumption-like structural rule, which we call *materialization*, and which involves the *precision* relation commonly present in the gradual typing literature. This contrasts with existing approaches where gradual typing is added by combining subtyping and precision (or consistency) to obtain a non-transitive relation called *consistent subtyping*. Due to its lack of transitivity, this relation has to be embedded in the typing rules, thus making the extension of an existing type system difficult.

As customary in the gradual typing literature, we also study *cast languages*, to which we compile our *source languages*, and we define the semantics of a source language in terms of the semantics of the associated target language. Once again, we show that the declarative presen-

tation of the compilation system is straightforward: to every application of the materialization structural rule corresponds the introduction of a cast in the compiled term. This also provides a powerful insight into the meaning of *blame*, as we show that in our system, blame can never be attributed to the context of an expression.

Naturally, this approach is well-suited to a declarative presentation, but not to an algorithmic presentation (that is, a presentation without structural typing rules). Thus, we show how to define algorithmic type systems following our declarative type systems based on constraint solving algorithms. We show in particular that constraints on gradual types can be solved by converting occurrences of the dynamic type into type variables.

We follow this presentation for two different systems. In Chapter 4, we study a standard Hindley-Milner type system with ML-like polymorphism. We show that our system is as powerful as several existing type systems, despite the absence of consistency and consistent subtyping. We also show that our type inference algorithm, based on unification, is sound and complete.

In Chapter 5, we add subtyping to our system. From a declarative point of view, adding semantic subtyping to our system simply amounts to adding a subsumption rule. However, to do this, the subtyping relation must be extended to gradual types. Rather than extending the interpretation function of semantic subtyping to gradual types, we show that we can translate gradual types into polymorphic non-gradual types, and define the subtyping relation on gradual types in terms of this translation. We also use this translation during the algorithmic presentation, since we solve constraints on gradual types by converting them to polymorphic non-gradual types. This yields a system which we prove to be sound (but not complete). Additionally, we provide a formal operational semantics for a cast language with set-theoretic types, based on complex syntactic operations on types. These operations provide a set-theoretic equivalent of the *ground types* present in the gradual typing literature, and allow us to define the semantics of our language as a sound and conservative, albeit complex, extension of the semantics presented in Chapter 4.

1.2.2. Denotational semantics

The definition of the semantics of the cast language in Chapter 5 brought to light several fundamental problems with our approach, most notably with our definition of the precision relation. Its syntactic nature contrasts heavily with the semantic nature of set-theoretic types and semantic subtyping, which makes the formal reasoning much more difficult.

In an attempt to solve these problems, and fill in the missing pieces in our understanding of gradual set-theoretic types, we decided to try and give a set-theoretic interpretation of gradual types, and a denotational semantics for a gradually-typed language. At the same time, we attempted to reconcile the model of values of semantic subtyping with the semantics of a language, that is, to give a denotational semantics of a language in terms of the interpretation domain used to interpret types. We tackled these problems in settings of increasing difficulty.

In Chapter 9, we study a simple λ-calculus with set-theoretic types but without function overloading or typecases. We present our first attempt at defining a denotational semantics for this calculus, directly in terms of the interpretation domain of semantic subtyping as presented by Frisch et al. [27]. We quickly bring to light a problem with this approach, as the interpretation domain cannot properly represent the semantics of functions without losing information.

We solve this problem in Chapter 10 by slightly modifying the interpretation domain. We prove that the denotational semantics of our simple λ-calculus in this new domain is sound and adequate, that is, it properly models the operational behaviour of our calculus, both for diverging

terms and converging terms. Naturally, modifying the interpretation of types raises the question of whether the subtyping relation induced by this new interpretation is equivalent to the subtyping relation defined by Frisch et al. [27], and we show that it is indeed the case.

In Chapter 11, we extend our calculus with overloaded functions, typecases, and non-determinism, to obtain a calculus modeling the functional core of ℂDuce. We present how the interpretation domain can be extended to properly represent each of these features, before summarizing everything by giving a sound denotational semantics for the calculus. Everything is not perfect however, since our semantics requires the introduction of some restrictions on the type system of ℂDuce. Nevertheless, in the very last section, we show how our semantics and the semantics of ℂDuce can be reconciled.

Finally, in Chapter 12, we tackle our original goal of providing a set-theoretic interpretation of gradual types and a denotational semantics for a gradually-typed language. We start by gathering all the intuition we acquired thus far, and provide a set-theoretic interpretation of gradual types. We show that this new interpretation highlights strong properties about the behaviour of set-theoretic gradual types, and we use it to define and study semantic versions of subtyping on gradual types and precision. We conclude by providing a sound denotational semantics for a simply-typed cast language.

1.2.3. The semantics of gradual types

The semantics definitions of gradual subtyping and precision we obtain in Chapter 12 have strong consequences. First, they solve a particular problem of our approach presented in Chapter 5, which is related to the way we dealt with negation types. We show in particular that, with these new relations, $?$ and $\neg?$ are equivalent from both a subtyping and a precision point of view. We also prove a strong result about the representation of gradual types, which states that every gradual set-theoretic type can be represented using a single occurrence of the dynamic type.

Equipped with these new results, we go back to the semantics of the cast language presented in Chapter 5, with the goal of simplifying it. In Chapter 6, we introduce the *semantic gradual subtyping* and *semantic precision* relations, and we show that these relations can be used to solve some of the problems we encountered when designing the semantics of our cast language, since they allow us to manipulate gradual types semantically rather than syntactically. In particular, type operators (which compute the domain of a function or the type of the result of an application) can now be lifted easily from non-gradual types to gradual types, keeping all their properties in the process. We use these results to define a new semantics for the cast language of Chapter 5 which is arguably much simpler and intuitive than the first.

1.3. A brief timeline of this thesis

This work started as an internship, which led to an article on gradual set-theoretic types presented at *ICFP 2017* (Castagna and Lanvin [13]). When we started this work, we believed that adding support for set-theoretic types to existing gradual type systems would be a few months work, at most. However, as demonstrated by the very existence of this manuscript, things were not so easy.

We quickly realized that most of our intuitions about the behaviour of gradual set-theoretic type were wrong (we initially firmly believed that $\mathsf{Int} \wedge ?$ and Int were equivalent, which led to many unsound type systems). Nevertheless, after many tweaks and fixes, we managed to obtain a sound gradually-typed language supporting full-fledged set-theoretic types, which is presented

in the aforementioned paper. However, the type system and operational semantics were very ad-hoc, inflexible, and difficult to generalize. This motivated us to try and find a more general interpretation of gradual set-theoretic types.

This new interpretation constitutes most of Part I, and has been presented at *POPL 2019* (Castagna et al. [18]). This new approach proved to be much more flexible than the first, and almost entirely subsumes it. Therefore, the material from our first paper is mostly absent from this manuscript, apart from some very specific results (for example, Theorem 5.25). Our *POPL 2019* paper focused on three major points: declarative typing, type inference, and operational semantics. In this presentation, I concentrate on declarative typing and operational semantics, which are the parts I was involved the most in. I will only give a brief overview of the ideas involved in the inference systems, whose development is mostly due to Tommaso Petrucciani and is presented in his thesis (Petrucciani [56]).

We then took a small detour to study two other problems. The first concerned the development of an abstract machine for a gradually-typed cast language with set-theoretic types, focusing on its runtime space efficiency and an efficient implementation of casts. This work has been presented at *IFL 2019* (Castagna et al. [17]). The second problem was quite orthogonal and concerned occurrence typing. This is joint work with Giuseppe Castagna, Mickaël Laurent, and Kim Nguyễn, and is, at the time of writing, unpublished. Both these works are absent from this manuscript, since the difficulties encountered in their development are outside the main scope of this thesis. Nevertheless, they gave us some important insights into the behaviour of gradual set-theoretic types, which certainly influenced several parts of this presentation.

Finally, we were not entirely satisfied with several aspects of our previous approaches, especially with our very peculiar treatment of negation types. This led us to believe there was a crucial missing part in our understanding of gradual set-theoretic types. To solve this problem, we decided to take a few steps back and formalize, from a denotational point of view, a simple languages with set-theoretic types, to which we then added gradual types. This study, presented in Part II, is still in its early stages. Nevertheless, it has already proved very fruitful, and led to a deeper and better understanding of the various concepts underlying gradual set-theoretic types.

1.4. Outline

The chapter following this introduction is Chapter 2, which introduces the semantic subtyping approach and summarizes the existing work we will use throughout the thesis. Most notably, it presents set-theoretic types, their interpretation, and the subtyping relation it induces.

Most of the rest of the manuscript is then divided in two parts.

Part I This part focuses on our declarative approach to embed gradual typing in existing type systems.

> *Chapter 3* We introduce the work, focusing on the benefits of polymorphic gradual typing with set-theoretic types and type inference. We then briefly introduce our approach and ideas.
>
> *Chapter 4* We start by applying our approach to an ML-like language with let-polymorphism but no subtyping. We describe, step by step, the source language, its declarative type system, the cast language and its semantics, the compilation procedure, and the type inference algorithm.

Chapter 5 We extend our approach to support set-theoretic types, which requires defining a suitable subtyping relation on gradual set-theoretic types. We follow the same presentation as in the preceding chapter. We briefly present a cast language and its operational semantics, although we relegate most of the presentation to the appendix due to its complexity.

Chapter 6 To solve some problems of the approach presented in Chapter 5, we introduce various results from Part II. We show some powerful result about the representation of set-theoretic gradual types, and use them to give a much simpler semantics for the cast calculus of the previous chapter.

Chapter 7 We conclude this part by discussing our results, comparing them with related work, and pointing some interesting directions for future work.

Part II This second part focuses on our denotational study of several languages with set-theoretic types.

Chapter 8 We introduce the work and our motivations, and we summarize our contributions.

Chapter 9 We start by a simple λ-calculus, and try to define its denotational semantics by directly using existing work. This highlights several problems which we explain.

Chapter 10 We solve the problems exposed in the preceding chapter by adapting our approach, and defining a new interpretation of types. We show that the subtyping relation is unchanged, and our semantics is sound and adequate.

Chapter 11 We extend our approach to support typecases, overloading, and non-determinism. This allows us to define a formal denotational semantics for the functional core of ℂDuce , although we impose some restrictions on the type system that we later show how to weaken.

Chapter 12 We use the formalism we established thus far to tackle our original goal of defining a denotational semantics for a gradually-typed language. Although we restrict the denotational semantics to a simply-typed cast language, we give a full set-theoretic interpretation of gradual types which yields several powerful results and ties up the two parts of this manuscript.

Chapter 13 We conclude this part by discussing our results and our approach, and we point out the main aspects that can be improved.

We conclude the main content of this manuscript by a conclusion in which we summarize our results and the most important directions for future work.

1.5. Notational conventions

Powerset. Given a set S, we write $\mathscr{P}(S)$ for the *powerset* of S, that is, the set of all subsets of S: $\mathscr{P}(S) = \{S' \mid S' \subseteq S\}$. Additionally, we write $\mathscr{P}_f(S)$ for its *finite powerset*, that is, the set of its finite subsets.

Vectors. We write vectors and tuples using a superscript arrow ($\vec{\cdot}$). We use this notation with type variables ($\vec{\alpha}, \vec{\beta}$ or $\vec{X}, \vec{Y}$) and types ($\vec{t}$, $\vec{\tau}$). When considering a vector of type variables, we suppose that all variables are distinct. Therefore, we may implicitly convert between vectors of type variables and sets of type variables.

Free variables. Given a vector (or a set) of type variables $\vec{\alpha}$, we use $\beta \sharp \vec{\alpha}$ as an equivalent notation for $\beta \notin \vec{\alpha}$. We lift this relation to vectors of variables, so that $\vec{\beta} \sharp \vec{\alpha}$ stands for $\vec{\beta} \cap \vec{\alpha} = \emptyset$. We also use this notation with entities other than sets of variables, in which case the notation refers to the set of type variables occurring in these entities (which will often be defined using vars or a similar notation). For example, given a vector of type $\vec{t}$, we write $\vec{\alpha} \sharp \vec{t}$ if $\vec{\alpha} \cap \mathsf{vars}(\vec{t}) = \emptyset$, where $\mathsf{vars}(\vec{t})$ denotes the set of all type variables appearing in one or more of the types of $\vec{t}$. When two entities or more appear on one side of the operator $\sharp$ separated by commas, we consider the union of the type variables they contain. For example, $\vec{\alpha}, \vec{\beta} \sharp \vec{t}, \vec{t}'$ stands for $(\vec{\alpha} \cup \vec{\beta}) \cap (\mathsf{vars}(\vec{t}) \cup \mathsf{vars}(\vec{t}')) = \emptyset$.

Substitutions. We choose to use the postfix notation $[r/s]$ for substitutions, where s is the replaced entity, and r is the replacement. We will use this notation with type variables and types (for example, $[t/\alpha]$), and with language variables and values (for example, $[v/x]$). Given a vector of type variables $\vec{\alpha}$ and a vector of types $\vec{t}$ having the same length, we write $[\vec{t}/\vec{\alpha}]$ for the component-wise substitution of every element of $\vec{\alpha}$ by the corresponding type in $\vec{t}$.

Chapter 2.

Background

In this chapter, we introduce the background on semantic subtyping and set-theoretic types needed for the rest of this manuscript. Most of the results and definitions presented here come from the work of Frisch et al. [27], Castagna and Xu [14], and Gesbert et al. [32].

All along this thesis, we will extensively use and revisit the results presented in this section. In Part I, we will reuse them directly to give an interpretation of set-theoretic gradual types. In Part II, we will draw inspiration from these results to define new interpretations of types and terms.

Chapter Outline

Section 2.1 This section is a general introduction to set-theoretic types and semantic subtyping, and gives a rough presentation of the most important features of semantic subtyping.

Section 2.2 This section introduces set-theoretic types with type variables, and formalizes several important concepts such as type substitutions.

Section 2.3 In this section, we present semantic subtyping and the set-theoretic interpretation of types.

Section 2.4 We present several properties of semantic subtyping. We define disjunctive normal forms for set-theoretic types and use this notion to introduce several type operators.

2.1. Introduction

We have already given some examples highlighting how types with unions, intersection and negation connectives (which we call *set-theoretic types*) can be useful from a programming point of view. However, for set-theoretic types to be truly usable, we need to endow them with a suitable and intuitive subtyping relation.

From the standpoint of a programmer, it is arguably common and intuitive to think of types in terms of sets of values: for example, the type Int can be seen as the set of all values denoting integers. Then, an expression can be given type t if every value it can produce is of type t. Following this intuition gives set-theoretic types a natural interpretation that results from the interpretation of set-theoretic connectives. If Even is the type denoting the set of even integers and Nat the type denoting the set of natural numbers, then $\mathsf{Even} \wedge \mathsf{Nat}$ denotes their intersection, which is the set of non-negative even integers.

Subtyping conveys a notion of *safety*: a type t is a subtype of a type t' if any expression of type t can be *safely* passed to a context expecting an expression of type t'. Along with the

above set-theoretic interpretation, this highlights a strong relation between subtyping and set-containment. It also results from this intuition that the subtyping relation on set-theoretic types must follow several natural distributivity rules, such as De Morgan's laws. For example, it must treat $t \wedge (t_1 \vee t_2)$ and $(t \wedge t_1) \vee (t \wedge t_2)$ as equivalent. Some distributivity rules also apply to type constructors: for example, $(t \times t_1) \vee (t \times t_2)$ must be equivalent to $t \times (t_1 \vee t_2)$ since both types denote the same set of values. Likewise for function types, $(t \to t_1) \vee (t \to t_2)$ must be equivalent to $t \to (t_1 \vee t_2)$.

This shows that giving a syntactic definition of subtyping, which is often the preferred approach, would be extremely complex and error prone. Many rules would be needed to ensure that subtyping satisfies all the distributivity properties we want, and proving the formal properties of subtyping would require large case disjunctions. Therefore, we study and present an alternative way of defining subtyping, which consists in first defining a set-theoretic model of types, and then interpreting subtyping between two types as inclusion between their interpretations.

While this seems a simple endeavour, there are some technical details one must be aware of. In particular, one must be very careful not to introduce any circularity in the definitions: the type system depends on the subtyping relation, which depends on the set-theoretic interpretation of types, which depends itself on the type system, to deduce whether a value belongs to a type. To break this circularity, the model usually corresponds to an untyped denotational semantics where types are interpreted as ideals, which precludes the set-theoretic interpretation of negation types (since the complement of an ideal is not an ideal).

The approach we present here, dubbed *semantic subtyping*, takes a middle ground between a syntactic definition of subtyping and a semantic definition based on a full-fledged denotational semantics of the language. Instead of starting from a subtyping relation to arrive to a model, it starts by defining a model, which is not, in principle, related to the terms of a language, and builds a subtyping relation from this model. While this may seem counter-intuitive, it is not necessary to relate the interpretation of types with their actual meaning in the language, we just need to ensure that the induced subtyping relation is safe (to ensure the type soundness of the language) and follows the properties we want. Of course, the closer the model and the values of the language are, the more precise the subtyping relation will be. Therefore, while the model of types is, in principle, detached from the values of the language, its definition tries to model the behaviour of values.

2.1.1. Set-theoretic interpretation

As we explained, to define subtyping using the semantic subtyping approach, we start by defining an *interpretation domain*, denoted $\mathscr{D}$, which roughly follows the definition of the values of a language. Then, we define a interpretation of types (the set of which is denoted Types) as sets of elements of this domain $[\![.]\!] : \text{Types} \to \mathscr{P}(\mathscr{D})$. Finally, the subtyping relation is defined using set-containment over this interpretation: $t_1 \leq t_2 \stackrel{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$.

As anticipated, the interpretation of set-theoretic types must follow from the mathematical interpretation of the set-theoretic operations. That is, if $\mathbb{0}$ denotes the empty type and $\mathbb{1}$ the top type (that is, every well-typed expression can be given type $\mathbb{1}$), then the following equalities must hold:

$$\begin{array}{rclcrcl} [\![\mathbb{0}]\!] & = & \emptyset & & [\![\mathbb{1}]\!] & = & \mathscr{D} \\ [\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] & & [\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\ [\![\neg t]\!] & = & \mathscr{D} \setminus [\![t]\!] & & & & \end{array}$$

Fixing these equalities leaves the interpretation of type constructors and constants to be defined, which is the focus of the next subsections.

Remark 2.1.
These properties also ensure that the interpretation of types inherit all the distributivity properties of the set-theoretic operations, as well as De Morgan's laws. For example, it holds that $[\![\mathbb{1}]\!] = [\![\neg\mathbb{0}]\!]$*, and* $[\![t_1 \wedge t_2]\!] = [\![\neg(\neg t_1 \vee \neg t_2)]\!]$*. Therefore, it is customary to define* $\mathbb{1}$ *as syntactic sugar for* $\neg\mathbb{0}$*, and* $t_1 \wedge t_2$ *as syntactic sugar for* $\neg(\neg t_1 \vee \neg t_2)$*, only leaving* $\vee$*,* $\neg$ *and* $\mathbb{0}$ *to be included in the grammar of types.*

Provided we defined the interpretation of type constructors and constants, we can use the interpretation $[\![.]\!]$ to define subtyping, as we anticipated: $t_1 \leq t_2 \overset{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$. It remains to prove the properties of this subtyping relation needed to ensure the soundness of the type system it is included in. Once again, many properties follow from the properties of the corresponding set-theoretic operations. For example, the transitivity of subtyping follows from the transitivity of set-containment. Additionally, note that $S_1 \subseteq S_2$ holds if and only if $S_1 \cap \overline{S}_2 = \emptyset$. This ensures that $t_1 \leq t_2$ if and only if $t_1 \wedge \neg t_2 \leq \mathbb{0}$. Thus, many properties of semantic subtyping reduce to emptiness problems. This is also especially important for the decidability of the subtyping relation, since this shows that it can be decided as long as the emptiness of a type can also be.

2.1.2. Semantic subtyping for first-order languages

The approach we present here was first introduced by Hosoya et al. [38], when working on the XML processing language XDuce. They defined subtyping semantically by building a model of types, without building a full model of the underlying language.

The language in question is a monomorphic, first-order language. If we abstract away the XML-based constructs, this leaves us with a language that does not contain functions, but features pairs and constants as values. A syntactic definition of values would thus be $v ::= c \mid (v, v)$, where c ranges over a certain set of constants $\mathscr{C}$.

Types then feature base types b for constants, and a constructor $\times$ for pair types. Additionally, Hosoya, Vouillon and Pierce consider recursive set-theoretic types, therefore, types can also contain set-theoretic connectives. To summarize, types are defined coinductively by the following grammar, using the previously-defined syntactic sugar for $\wedge$ and $\mathbb{0}$:

$$t ::= b \mid t \times t \mid t \vee t \mid \neg t \mid \mathbb{0}$$

In this setting, Hosoya, Vouillon and Pierce showed that types can be immediately interpreted as sets of values of the language, without introducing any circularity. Base types are interpreted as the set of constants belonging to them (for example, $[\![b]\!] = \{\texttt{true}, \texttt{false}\}$), and pair types are interpreted using the Cartesian product ($[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$).

While we will show in the next section that it is not practical, in a higher-order setting, to interpret types immediately as sets of values, we still follow the same reasoning when it comes to the interpretation of constants and pairs. As we will formalize later on, we directly add the constants $\mathscr{C}$ of the language to the interpretation domain $\mathscr{D}$, and introduce a function $\mathbb{B}(.)$ that associates to every base type the set of constants that belong to it, such that $[\![b]\!] = \mathbb{B}(b)$.

2.1.3. Adding arrow types

Frisch et al. [27] extend this approach to a higher-order setting, by considering languages which

include first-class functions, and therefore arrow types. As anticipated, this requires some major changes, since interpreting types directly as sets of values of the language would cause a circularity issue: we could define the interpretation of arrow types as $[\![t_1 \to t_2]\!] = \{\lambda\, x.\ e \mid\vdash \lambda\, x.\ e : t_1 \to t_2\}$, but then the interpretation of types would depend on the definition of the relation $\vdash e : t$, which in turn depends on the definition of subtyping (which is classically added via a subsumption rule). However, since our goal is to use the definition of $[\![.]\!]$ to define subtyping, we are stuck in a dependency loop.

This problem does not occur in a first-order setting because, in a language whose values are restricted to constants and pairs, the restriction of the relation $\vdash e : t$ to values is straightforward. Deriving the type of λ-abstractions, however, requires the full type system since it involves the typing of arbitrary expressions.

Hence, we need to decouple the interpretation of types from values, and instead define an interpretation domain which mimics the behaviour of values, and which allows us to define an interpretation function $[\![.]\!]$ inducing a suitable subtyping relation. A first idea would be to interpret functions *extensionally*, as infinite directed graphs mapping inputs to outputs. This would be formalized as

$$[\![t_1 \to t_2]\!] \stackrel{\text{def}}{=} \{R \subseteq \mathscr{D} \times \mathscr{D} \mid \forall (d_1, d_2) \in R,\ d_1 \in [\![t_1]\!] \implies d_2 \in [\![t_2]\!]\}$$

Intuitively, a relation $\{(d_i, d_i') \mid i \in I\}$ represents a function which maps the element d_i to the element d_i' for every $i \in I$, and diverges on every other element d. For this relation to be in the interpretation of a type $t_1 \to t_2$, we simply require that if d_i belongs to the interpretation of t_1, then d_i' belongs to the interpretation of t_2. In other words, the function it represents maps every value of type t_1 into a value of type t_2, which is the expected behaviour. Note that we impose no requirement on the behaviour of the function on values that do not have type t_1. Relations can also be non-deterministic: the relation $\{(d, d_1), (d, d_2)\}$ maps the same element d to both d_1 and d_2, even if $d_1 \neq d_2$.

While this definition sounds intuitive, it causes a problem of cardinality. Relations are defined as infinite graphs on $\mathscr{D}$, that is, they are elements of $\mathscr{P}(\mathscr{D} \times \mathscr{D})$. For relations to also be elements of the domain (which is required by the definition of $[\![.]\!]$), $\mathscr{D}$ must satisfy the following inequality:

$$\mathscr{P}(\mathscr{D} \times \mathscr{D}) \subseteq \mathscr{D}$$

Provided $\mathscr{D}$ is non-empty (which is the case as soon as constants are added to the language), this inequality cannot hold, since $\mathscr{P}(\mathscr{D} \times \mathscr{D})$ and $\mathscr{D}$ necessarily have different cardinalities.

The solution proposed by Frisch et al. [27] is to restrict relations to *finite* relations. If we restrict $\mathscr{P}(\mathscr{D} \times \mathscr{D})$ to finite sets, which we denote by $\mathscr{P}_f(\mathscr{D} \times \mathscr{D})$, then the above equation can hold. This means that functions do not exactly correspond to elements of $\mathscr{D}$ since any function that can produce an infinite number of results (such as the identity function) cannot be represented by a single element of $\mathscr{D}$. However, it can still be represented by an infinite number of finite relations of $\mathscr{D}$, which corresponds, in essence, to its *finite approximations*. This also makes sense intuitively from a programming point of view: any terminating program can only produce a finite number of intermediary values, and its functions can only be applied to a finite number of values. Therefore, in principle, the behaviour of any function in a terminating program can be approximated using finite relations.

Subtyping also behaves as expected with this restriction, thanks to the fact that for every two sets A and B, $\mathscr{P}(A) \subseteq \mathscr{P}(B) \iff \mathscr{P}_f(A) \subseteq \mathscr{P}_f(B)$ holds. Therefore, restricting relations to finite relations does not change the behaviour of subtyping. This leads to the following definition

of the interpretation of arrow types:

$$[\![t_1 \to t_2]\!] \stackrel{\text{def}}{=} \{R \in \mathscr{P}_f(\mathscr{D} \times \mathscr{D}) \mid \forall (d_1, d_2) \in R,\ d_1 \in [\![t_1]\!] \implies d_2 \in [\![t_2]\!]\}$$

However, one problem remains. Following this definition, we have:

$$[\![\mathbb{1} \to \mathbb{1}]\!] \stackrel{\text{def}}{=} \{R \in \mathscr{P}_f(\mathscr{D} \times \mathscr{D}) \mid \forall (d_1, d_2) \in R,\ d_1 \in [\![\mathbb{1}]\!] \implies d_2 \in [\![\mathbb{1}]\!]\}$$

Since $d_2 \in [\![\mathbb{1}]\!]$ always holds, this yields $[\![\mathbb{1} \to \mathbb{1}]\!] = \mathscr{P}_f(\mathscr{D} \times \mathscr{D})$. In other words, the interpretation of any type $t_1 \to t_2$ is contained in the interpretation of $\mathbb{1} \to \mathbb{1}$. By definition of subtyping, this means that any function can be given type $\mathbb{1} \to \mathbb{1}$ (as long as it can be given an arrow type), and can thus be applied to any argument. This is, obviously, unsound in most settings: the function $\lambda x.\ x\ x$ can be applied to 0, reducing to 0 0, which is a stuck term.

The solution to this second issue is to introduce a special, distinguished symbol, which we denote Ω. This symbol represents a runtime type error occurring when a function is applied to an element outside its domain. As such, this particular symbol only appears in the right hand side of relations, and does not belong to the domain $\mathscr{D}$. In the formal definition we will give in Section 2.3, we will define the set $\mathscr{D}_\Omega$ as $\mathscr{D} \cup \{\Omega\}$, ranged over by ∂, and define relations as elements of $\mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega)$. The interpretation of arrow types then becomes:

$$[\![t_1 \to t_2]\!] \stackrel{\text{def}}{=} \{R \in \mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega) \mid \forall (d, \partial) \in R,\ d \in [\![t_1]\!] \implies \partial \in [\![t_2]\!]\}$$

With this new definition, $\partial \in [\![\mathbb{1}]\!]$ holds if and only if $\partial \neq \Omega$. Therefore, the interpretation of $\mathbb{1} \to \mathbb{1}$ is not equal to $\mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega)$ (the set of all relations) anymore, since it does not contain the relations belonging to $\mathscr{P}_f(\mathscr{D} \times \{\Omega\})$. On the other hand, the interpretation of $\mathsf{Int} \to \mathsf{Int}$ now contains, for example, the relation $\{(\mathtt{true}, \Omega)\}$, which ensures that $\mathsf{Int} \to \mathsf{Int}$ is not a subtype of $\mathbb{1} \to \mathbb{1}$.

This approach allows us to define a subtyping relation on monomorphic set-theoretic types which enjoys the properties we wanted. Frisch et al. [27] also show that this relation is decidable, and describe an algorithm, which has then be implemented as part of the language CDuce . Interestingly, they also show that while types are not directly interpreted as sets of values to define subtyping, we can later define a second interpretation of types $[\![.]\!]^{\mathscr{V}}$ as $[\![t]\!]^{\mathscr{V}} \stackrel{\text{def}}{=} \{v \mid\!\vdash v : t\}$, and prove that $[\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!]^{\mathscr{V}} \subseteq [\![t_2]\!]^{\mathscr{V}}$. This proves that the subtyping relation they define using $\mathscr{D}$ truly corresponds to a relation on sets of values of the language.

2.1.4. Adding type variables

We have now explained how to use the semantic subtyping approach in a system with constant, product, and arrow types. However, we are still limited to a monomorphic setting. The next step is to add polymorphism, and allow types to contain type variables. We only consider prenex parametric polymorphism, as types supporting first-class polymorphism have not been studied yet using semantic subtyping.

There are two important aspects to type variables. The first is that type variables are only related to themselves (by reflexivity), and then follow the usual set-theoretic principles. For example, $\alpha \leq \alpha \vee t$ holds, as does $\alpha \wedge t \leq \alpha$. The second aspect is that subtyping must be stable by type substitution: if $t_1 \leq t_2$ holds, then $t_1\theta \leq t_2\theta$ must also hold for every type substitution θ.

The solution proposed by Castagna and Xu [14] is to integrate type substitutions directly into the interpretation $[\![.]\!]$. They parameterize the interpretation by an *assignment*, which they

denote $\eta : \mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D})$, and which maps variables to subsets of $\mathscr{D}$. The interpretation of a type depends on this assignment, so that the interpretation function is now $[\![.]\!]_{(.)} : \text{Types} \to (\mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D})) \to \mathscr{P}(\mathscr{D})$. The interpretation of most types does not change: if t is a closed type, then $[\![t]\!]_\eta = [\![t]\!]$ for every assignment η (where the interpretation in the right hand side is the monomorphic version we presented in the previous subsections). The difference comes from the interpretation of type variables, which is defined as $[\![\alpha]\!]_\eta = \eta(\alpha)$. Since subtyping must be preserved by arbitrary type substitutions, the definition of subtyping is now:

$$t_1 \leq t_2 \overset{\text{def}}{\iff} \forall \eta : \mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D}).\ [\![t_1]\!]_\eta \subseteq [\![t_2]\!]_\eta$$

However, this definition has several problems, which Castagna and Xu [14] discuss. The first problem is algorithmic: due to the universal quantification on assignments, it is not known whether the relation is decidable, and they conjecture it to be NEXPTIME-complete if it is. The second problem is that this relation is not always intuitive. As an example, Castagna and Xu [14] present the statement:

$$t \times \alpha \leq (t \times \neg t) \vee (\alpha \times t)$$

where t is closed. Here, the variable α appears in two unrelated positions in the two types at hand, therefore, following the intuition that α is only comparable with itself, one would expect this statement to be false. However, according to the above definition, if t is taken to be a singleton type then the statement holds.

To understand why this happens, consider a type t such that $[\![t]\!]_\eta = \{d\}$ independently of η. For every assignment η, we either have $d \in [\![\alpha]\!]_\eta$ or $[\![\alpha]\!]_\eta \subseteq \mathscr{D} \setminus \{d\}$. Thus, the following judgment holds for every assignment η, proving that $t \times \alpha \leq (t \times \neg t) \vee (\alpha \times t)$:

$$\{d\} \times \eta(\alpha) \subseteq (\{d\} \times (\mathscr{D} \setminus \{d\})) \cup (\eta(\alpha) \times \{d\})$$

Castagna and Xu [14] argue that the problem comes from the fact that for every assignment η, either $[\![t \wedge \alpha]\!]_\eta$ is empty, or $[\![t \wedge \neg\alpha]\!]_\eta$ is empty; but neither of the two interpretations is *always* empty for *every* assignment. Therefore, they introduce a property they call *convexity* as a solution to this problem, akin to *parametricity* as studied by Reynolds [62]. An interpretation $[\![.]\!]_{(.)}$ is said to be *convex* if it satisfies the following statement for every set of types $\{t_1, \ldots, t_n\} \subseteq \text{Types}$:

$$\forall \eta.\ \exists i \in \{1..n\}.\ [\![t_i]\!]_\eta = \emptyset \iff \exists i \in \{1..n\}.\ \forall \eta.\ [\![t_i]\!]_\eta = \emptyset$$

For the reasons we stated above, this property cannot be satisfied by any interpretation that contains singleton types (types whose interpretation is a singleton that is independent of the assignment η). Therefore, to achieve convexity, Castagna and Xu [14] propose to change the interpretation of closed types (and, in particular, singleton types) to interpret them into infinite sets. While this breaks even further the correspondence between the interpretation domain and the values of the language, this ensures the subtyping relation avoids problematic judgments such as the one we presented.

The interpretation domain and (convex) interpretation function we will use throughout the manuscript comes from the work of Gesbert et al. [32], in which they present an algorithm to decide polymorphic semantic subtyping. They follow the idea of Castagna and Xu [14], and interpret closed and base types into infinite sets. To do this, they attach sets of *labels* to the elements of the domain $\mathscr{D}$, where each label is associated to a type variable. Formally, the domain

is now defined by the following grammar:

$$\mathscr{D} \ni d ::= c^L \mid (d, d)^L \mid \{(d, \partial), \ldots, (d, \partial)\}^L$$

where L ranges over $\mathscr{P}_f(\mathscr{V}^\alpha)$. The interpretation of types is then modified so that closed types contain elements independently of their labels:

$$\begin{aligned} [\![b]\!]_\eta &= \{c^L \mid c \in \mathbb{B}(b)\} \\ [\![t_1 \times t_2]\!]_\eta &= \{(d_1, d_2)^L \mid d_1 \in [\![t_1]\!] \wedge d_2 \in [\![t_2]\!]\} \end{aligned}$$

Gesbert et al. [32] go even further by proving that it is possible to remove the dependency of the interpretation on an assignment η, by interpreting variables as the set of elements having a matching label:

$$[\![\alpha]\!] = \{d \in \mathscr{D} \mid \alpha \in \mathsf{tags}(d)\}$$

where $\mathsf{tags}(d)$ denotes the outermost set of labels attached to an element d. This interpretation makes the subtyping relation easier to define and decide, since it removes the universal quantification introduced by the parameter η. Gesbert et al. [32] define another relation using quantification where, as before, $[\![\alpha]\!]_\eta = \eta(\alpha)$, and prove that the two are actually equivalent. Throughout this manuscript, we will use the first interpretation (which does not depend on η) since it is easier to manipulate, but we will still study the second in Section 2.4 since it is essential to prove that subtyping is preserved by type substitutions.

2.2. Set-theoretic types

We now bring together the various concepts presented throughout the previous section, and formalize the types we will use throughout this manuscript.

2.2.1. Syntax

We consider types that feature *base types*, *type variables*, products, arrows, and set-theoretic connectives (see Definition 2.2 for the formal definition). Therefore, we suppose that there exist three sets corresponding to base types, type variables, and language constants:

$\mathscr{V}^\alpha \ni \alpha, \beta, \gamma$	type variables
$\mathscr{C} \ni c$	language constants
$\mathscr{B} \ni b$	base types

The set $\mathscr{V}^\alpha$ is supposed to be countably infinite, and $\mathscr{V}^\alpha \cap \mathscr{B} = \emptyset$. Moreover, as anticipated in Subsection 2.1.2, we assume the existence of two functions

$$\mathbb{B}(.) : \mathscr{B} \to \mathscr{P}(\mathscr{C}) \qquad b_{(.)} : \mathscr{C} \to \mathscr{B}$$

such that $\mathbb{B}(.)$ associates to every base type the set of constants that belong to it, and $b_{(.)}$ associates to every constant c its most precise type b_c.

Moreover, we assume the existence of a *singleton type* in $\mathscr{B}$ associated to every constant. A singleton type is a type such that a single constant c belongs to it, and is *de facto* the most precise type for this constant. As such, we have $\mathbb{B}(b_c) = \{c\}$ for every constant $c \in \mathscr{C}$. When the constant c is determined and when there is no ambiguity, we may use the constant itself instead

of b_c to denote its singleton type. For example, we may consider $\texttt{true} \vee \texttt{false}$ as a type equivalent to Bool.

Assuming the above definitions, the types we consider are defined as follows.

Definition 2.2. *The set* Types *of* set-theoretic types *is the set of terms defined coinductively by the following grammar:*

$$
\begin{array}{lll}
t ::= \alpha & & \textit{type variable} \\
\quad \mid b & & \textit{base type} \\
\quad \mid t \times t & & \textit{product} \\
\quad \mid t \to t & & \textit{arrow} \\
\quad \mid t \vee t & & \textit{union} \\
\quad \mid \neg t & & \textit{negation} \\
\quad \mid \mathbb{0} & & \textit{empty}
\end{array}
$$

that moreover satisfy the following two conditions:

- (regularity) *the term has a finite number of different sub-terms;*
- (contractivity) *every infinite branch of the term contains an infinite number of occurrences of the* $\times$ *or* $\to$ *type constructors.*

As anticipated, the above definition does not introduce a top type or intersection and difference connectives. They are defined as abbreviations using the following equalities:

$$
\begin{array}{lll}
t_1 \wedge t_2 \stackrel{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2) & & \text{intersection} \\
t_1 \setminus t_2 \stackrel{\text{def}}{=} t_1 \wedge \neg t_2 & & \text{difference} \\
\mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0} & & \text{any}
\end{array}
$$

We refer to b, $\times$, and $\to$ as *type constructors* and to $\vee$, $\neg$, $\wedge$, and $\setminus$ as *type connectives*.

We suppose the existence of a base type $\mathbb{1}_{\mathscr{B}} \in \mathscr{B}$ that is the top type of base types: $\forall b \in \mathscr{B}$, $\mathbb{B}(b) \subseteq \mathbb{B}(\mathbb{1}_{\mathscr{B}})$.

The negation connective is given a higher precedence than the other connectives, which are themselves given a higher precedence than constructors. As such, following the usual convention that the constructor $\to$ is right-associative, $t_1 \to t_2 \wedge t_1' \to t_2'$ is actually equivalent to $t_1 \to ((t_2 \wedge t_1') \to t_2')$. The intersection of two arrow types must be parenthesized as follows $(t_1 \to t_2) \wedge (t_1' \to t_2')$.

Note that types are defined *coinductively* instead of *inductively*, so they can be infinite trees, provided they satisfy the two conditions presented in Definition 2.2. This gives a definition of equi-recursive types that is equivalent to the arguably more common definition that involves explicit binders for recursion.

The contractivity condition is crucial because it removes ill-formed types such as $t = t \vee t$ (which does not carry any information about the set denoted by the type) or $t = \neg t$ (which cannot represent any set). It also ensures that the binary relation $\triangleright \subseteq \text{Types}^2$ defined by $t_1 \vee t_2 \triangleright t_i$, $\neg t \triangleright t$ is Noetherian (that is, strongly normalizing). This gives an induction principle on Types that we will use without any further reference to the relation. The induction principle derived from the relation "$\triangleright$" states that we can use induction on type connectives but not on type constructors.

This is well-founded because contractivity ensures that there are finitely many type connectives between two type constructors.

The regularity condition, on the other hand, is only needed to ensure the decidability of the subtyping relation, which relies on the finite representation of types.

Additionally, we write $\mathsf{vars}(t)$ to denote the set of type variables that occur in a type t. The function $\mathsf{vars}(.)$ cannot be defined inductively since types are defined coinductively, however, it still satisfies the following equalities:

$$\begin{array}{rclcrcl}
\mathsf{vars}(\alpha) & = & \{\alpha\} & & \mathsf{vars}(b) & = & \emptyset \\
\mathsf{vars}(t_1 \times t_2) & = & \mathsf{vars}(t_1) \cup \mathsf{vars}(t_2) & & \mathsf{vars}(t_1 \to t_2) & = & \mathsf{vars}(t_1) \cup \mathsf{vars}(t_2) \\
\mathsf{vars}(t_1 \vee t_2) & = & \mathsf{vars}(t_1) \cup \mathsf{vars}(t_2) & & \mathsf{vars}(\neg t) & = & \mathsf{vars}(t) \\
\mathsf{vars}(\mathbb{0}) & = & \emptyset & &
\end{array}$$

We say that a type t is closed if $\mathsf{vars}(t) = \emptyset$, and we say that a variable α is free in a type t, which we denote by $\alpha \sharp t$, if $\alpha \notin \mathsf{vars}(t)$.

2.2.2. Type substitutions

Throughout this manuscript, we will regularly use the concept of type substitutions, which we denote by θ. Their definition is fairly standard.

Definition 2.3 (Type substitution)**.** *A* type substitution *is a mapping* $\theta : \mathscr{V}^\alpha \to \mathsf{Types}$ *from type variables to types which is the identity everywhere except on a finite set of type variables, which we call its* domain*, denoted* $\mathsf{dom}(\theta)$*. Formally,* $\mathsf{dom}(\theta) = \{\alpha \in \mathscr{V}^\alpha \mid \theta(\alpha) \neq \alpha\}$*.*
We write $t\theta$ *for the application of the type substitution* θ *to the type* t*.*

The application of a type substitution to a set-theoretic type follows the intuitive equalities (which cannot be taken as an inductive definition due to recursive types):

$$\begin{array}{rclcrcl}
\alpha\theta & = & \theta(\alpha) & & b\theta & = & b \\
(t_1 \times t_2)\theta & = & (t_1\theta) \times (t_2\theta) & & (t_1 \to t_2)\theta & = & (t_1\theta) \to (t_2\theta) \\
(t_1 \vee t_2)\theta & = & (t_1\theta) \vee (t_2\theta) & & (\neg t)\theta & = & \neg(t\theta) \\
\mathbb{0}\theta & = & \mathbb{0} & &
\end{array}$$

Given a type t and a variable α, we use the notation $[t/\alpha]$ to denote the substitution θ such that $\mathsf{dom}(\theta) \subseteq \{\alpha\}$ and $\theta(\alpha) = t$. This notion is extended to vectors of types and type variables by defining it pointwise. Given two vectors $\vec{t}$ of types and $\vec{\alpha}$ of type variables of equal lengths n, we use $[\vec{t}/\vec{\alpha}]$ to denote the substitution θ such that $\mathsf{dom}(\theta) \subseteq \vec{\alpha}$ and $\alpha_i\theta = t_i$ for every $i \in \{1..n\}$.

We will also commonly refer to the set of variables introduced by a type substitution θ, which we denote $\mathsf{vars}(\theta)$, following the same notation as for the variables occurring in a type:

$$\mathsf{vars}(\theta) \stackrel{\text{def}}{=} \bigcup_{\alpha \in \mathsf{dom}(\theta)} \mathsf{vars}(\alpha\theta)$$

Given two type substitutions θ_1 and θ_2, we use $\theta_1 \circ \theta_2$ to denote their composition, defined by $(\theta_1 \circ \theta_2)(\alpha) \stackrel{\text{def}}{=} \alpha\theta_2\theta_1$. Moreover, if θ_1 and θ_2 are disjoint (that is, $\mathsf{dom}(\theta_1) \cap \mathsf{dom}(\theta_2) = \emptyset$), we

denote their union by $\theta_1 \cup \theta_2$, which is defined as:

$$(\theta_1 \cup \theta_2)(\alpha) \stackrel{\text{def}}{=} \begin{cases} \alpha\theta_1 & \text{if } \alpha \in \mathsf{dom}(\theta_1) \\ \alpha\theta_2 & \text{if } \alpha \in \mathsf{dom}(\theta_2) \\ \alpha & \text{otherwise} \end{cases}$$

In Part I, we will regularly refer to these definitions, extending them to various sets of type variables and types.

2.3. Semantic subtyping

As we explained in the previous sections, in semantic subtyping, we interpret types as subsets of an *interpretation domain*. While this domain corresponds roughly to the values of a language, we have shown that some care must be taken when interpreting functional values, for cardinality reasons. Therefore, we represent functions as finite relations. As anticipated, we also include tags to interpret type variables, and include a distinguished symbol Ω to represent type errors. The following definition summarizes the various steps presented in Section 2.1.

Definition 2.4 (Interpretation domain)**.** *The interpretation domain $\mathscr{D}$ is the set of finite terms d produced inductively by the following grammar:*

$$d ::= c^L \mid (d, d)^L \mid \{(d, \partial), \ldots, (d, \partial)\}^L$$
$$\partial ::= d \mid \Omega$$

where c ranges over the set $\mathscr{C}$ of constants, L ranges over $\mathscr{P}_f(\mathscr{V}^\alpha)$, and where Ω is such that $\Omega \notin \mathscr{D}$.
We also write $\mathscr{D}_\Omega = \mathscr{D} \cup \{\Omega\}$.

Throughout the manuscript, we will commonly denote finite relations $\{(d_1, \partial_1), \ldots, (d_n, \partial_n)\}^L$ by R^L (or simply R in a monomorphic setting), and use $\{(d_i, \partial_i) \mid i \in I\}^L$ as a more formal notation, implicitly assuming that I and L are both finite.

In the following definitions, to interpret type variables, we will need to refer to the set of tags attached to an element of $\mathscr{D}$. To ease the formalism, we define the following function $\mathsf{tags}(.)$, which extracts the outermost set of types variables attached to an element:

$$\mathsf{tags}(c^L) = \mathsf{tags}((d_1, d_2)^L) = \mathsf{tags}(\{(d_i, \partial_i) \mid i \in I\}^L) = L$$

The next step is to define the interpretation of types as sets of elements of $\mathscr{D}$, via a function $\mathscr{D}. : \mathsf{Types} \to \mathscr{P}(\mathscr{D})$. We have already explained in Section 2.1 the intuition behind the interpretation of each constructor and each connective. Formally, the interpretation must satisfy the

following equalities, where the sets of labels L are arbitrary:

$$\begin{aligned}
[\![\alpha]\!] &= \{d \mid \alpha \in \mathsf{tags}(d)\} \\
[\![b]\!] &= \{c^L \mid c \in \mathbb{B}(b)\} \\
[\![t_1 \times t_2]\!] &= \{(d_1, d_2)^L \mid d_1 \in [\![t_1]\!] \wedge d_2 \in [\![t_2]\!]\} \\
[\![t_1 \to t_2]\!] &= \{\{(d_i, \partial_i) \mid i \in I\}^L \mid \forall i \in I.\, d_i \in [\![t_1]\!] \implies \partial_i \in [\![t_2]\!]\} \\
[\![t_1 \vee t_2]\!] &= [\![t_1]\!] \cup [\![t_2]\!] \\
[\![\neg t]\!] &= \mathscr{D} \setminus [\![t]\!] \\
[\![\mathbb{0}]\!] &= \emptyset
\end{aligned}$$

However, the presence of recursive types do not allow us to use these equalities as an inductive definition of $[\![.]\!]$. Instead, we define a predicate on $\mathscr{D}_\Omega \times \mathsf{Types}$ to decide whether an element belongs to the interpretation of a type. We then use this predicate to define the function $[\![.]\!]$ satisfying the above equalities.

Since this predicate is defined on pairs of $\mathscr{D}_\Omega \times \mathsf{Types}$, we can rely on the aforementioned induction principle enabled by the contractivity property of types. Additionally, while types are interpreted as subsets of $\mathscr{D}$, the predicate is defined on $\mathscr{D}_\Omega$ to slightly simplify the case of arrow types.

Definition 2.5 (Set-theoretic interpretation of types)**.** *We define a binary predicate $(\partial : t)$ ("the element ∂ belongs to the type t") where $\partial \in \mathscr{D}_\Omega$ and $t \in \mathsf{Types}$, by induction on the pair (∂, t) ordered lexicographically. The predicate is defined as follows:*

$$\begin{aligned}
(d : \alpha) &= \alpha \in \mathsf{tags}(d) \\
(c^L : b) &= c \in \mathbb{B}(b) \\
((d_1, d_2)^L : t_1 \times t_2) &= (d_1 : t_1) \wedge (d_2 : t_2) \\
(\{(d_i, \partial_i) \mid i \in I\}^L : t_1 \to t_2) &= \forall i \in I.\, (d_i : t_1) \implies (\partial_i : t_2) \\
(d : t_1 \vee t_2) &= (d : t_1) \vee (d : t_2) \\
(d : \neg t) &= \neg(d : t) \\
(\partial : t) &= \mathit{false} \qquad \mathit{otherwise}
\end{aligned}$$

We define the set-theoretic interpretation *of set-theoretic types $[\![.]\!] : \mathsf{Types} \to \mathscr{P}(\mathscr{D})$ as $[\![t]\!] \stackrel{\text{def}}{=} \{d \in \mathscr{D} \mid (d : t)\}$.*

Finally, as anticipated, subtyping is defined using set-containment and the above definition of the interpretation of types.

Definition 2.6 (Subtyping)**.** *We define the* subtyping *relation $\leq$ and* subtyping equivalence *relation $\simeq$ as $t_1 \leq t_2 \stackrel{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$ and $t_1 \simeq t_2 \stackrel{\text{def}}{\iff} (t_1 \leq t_2)$ and $(t_2 \leq t_1)$.*

Remark 2.7.

The monomorphic version of semantic subtyping, which we will not state here, can be obtained straightforwardly by simply removing type variables α from the syntax of types given in Definition 2.2, and removing the sets of labels L from the interpretation domain and the interpretation of types. The types we consider in Part II are monomorphic set-theoretic types and their inter-

pretation will therefore draw inspiration from the monomorphic version of semantic subtyping. ⌟

2.4. Properties of semantic subtyping

This section is dedicated to the study of the subtyping relation defined in the previous section. We prove several results that will be needed in the rest of the manuscript. First of all, we introduce the aforementioned interpretation of types of Gesbert et al. [32] and their version of quantification-based semantic subtyping, which is proven to be equivalent to that of Definition 2.6. This then makes it possible to prove that subtyping is preserved by type substitutions, which will be crucial to ensure the soundness of the various systems presented in the thesis.

We also introduce some results from Frisch et al. [27] and Castagna and Xu [14] proving that types can be put in disjunctive normal forms with specific properties, and use these forms to define and study the properties of several type operators that will be useful for this work.

2.4.1. Equivalence with quantified subtyping

As anticipated, we now introduce the parameterized interpretation of types and quantification-based subtyping relation of Gesbert et al. [32], and prove that it is equivalent to the definition of subtyping we gave in Definition 2.6. While the latter is arguably easier to work with, it may seem less intuitive than the former when reasoning about polymorphic types. Moreover, the quantification-based definition makes it easier to reason about type substitutions, and the result of equivalence will then ensure that subtyping as defined in Definition 2.6 is stable under type substitution.

In this new interpretation, the interpretation domain is unchanged. However, as anticipated, the interpretation function is now parameterized with an *assignment*, which is a function $\eta : \mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D})$ mapping variables to subsets of the interpretation domain. As before, we cannot immediately give an inductive definition of the interpretation function $[\![.]\!]^q_{(.)}$ due to the presence of recursive types. Thus, we start by the definition of a ternary predicate on domain elements and types, parameterized by an assignment. All the notions introduced here, related to *quantification-based* subtyping, are distinguished from the notions presented in the previous chapter by a superscript ^{q}.

Definition 2.8. *We define a ternary predicate $(\partial :_\eta t)^q$ where $\partial \in \mathscr{D}_\Omega$, $t \in$ Types, and $\eta : \mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D})$ by induction on the pair (∂, t) ordered lexicographically. The predicate is defined as follows:*

$$
\begin{array}{rcll}
(d :_\eta \alpha)^q & = & d \in \eta(\alpha) & \\
(c^L :_\eta b)^q & = & c \in \mathbb{B}(b) & \\
((d_1, d_2)^L :_\eta t_1 \times t_2)^q & = & (d_1 :_\eta t_1)^q \wedge (d_2 :_\eta t_2)^q & \\
(\{(d_i, \partial_i) \mid i \in I\}^L :_\eta t_1 \to t_2)^q & = & \forall i \in I.\ (d_i :_\eta t_1)^q \implies (\partial_i :_\eta t_2)^q & \\
(d :_\eta t_1 \vee t_2)^q & = & (d :_\eta t_1)^q \vee (d :_\eta t_2)^q & \\
(d :_\eta \neg t)^q & = & \neg (d :_\eta t)^q & \\
(\partial :_\eta t)^q & = & \mathit{false} & \mathit{otherwise}
\end{array}
$$

We define the interpretation $[\![t]\!]^q_\eta$ of a set-theoretic type t as:

$$[\![t]\!]^q_\eta \stackrel{\text{def}}{=} \{d \in \mathscr{D} \mid (d :_\eta t)^q\}$$

We define the quantification-based subtyping relation $\leq^q$ on set-theoretic types as:

$$t_1 \leq^q t_2 \iff \forall \eta : \mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D}).\ [\![t_1]\!]^q_\eta \subseteq [\![t_2]\!]^q_\eta.$$

We now prove that the two subtyping relations $\leq$ and $\leq^q$ coincide. To achieve this, we first introduce the canonical assignment $\overline{\eta} : \mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D})$ defined by $\overline{\eta}(\alpha) = \{d \mid \alpha \in \text{tags}(d)\}$. Using this canonical assignment, the two interpretation functions coincide as stated by the following lemma:

Lemma 2.9. *For every type $t \in$ Types, $[\![t]\!] = [\![t]\!]^q_{\overline{\eta}}$.*

Proof. The statement is proven by a straightforward induction on the pair (d, t). □

The above lemma proves that the relation $\leq^q$ is stronger than $\leq$, since if subtyping holds for every assignment η, then it must in particular hold for $\overline{\eta}$. To prove the converse, we first prove a lemma due to Gesbert et al. [32], stating that if the interpretation of a type is empty for the assignment $\overline{\eta}$ then it must be empty for every other assignment η.

Lemma 2.10. *Let $V \in \mathscr{P}_f(\mathscr{V}^\alpha)$, and $T = \{t \in \text{Types} \mid \text{vars}(t) \subseteq V\}$. For every $t \in T$, the following holds:*

$$[\![t]\!]^q_{\overline{\eta}} = \emptyset \implies \forall \eta : \mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D}).\ [\![t]\!]^q_\eta = \emptyset$$

Proof. We prove the result by contraposition, by proving:

$$\forall t \in T.\ (\exists \eta.\ [\![t]\!]^q_\eta \neq \emptyset) \implies [\![t]\!]^q_{\overline{\eta}} \neq \emptyset$$

To prove this result, we prove the stronger statement:

$$\forall t \in T.\ \forall d \in \mathscr{D}.\ (d :_\eta t)^q \iff (F^\eta_V(d) :_{\overline{\eta}} t)^q$$

where the function $F^\eta_V(.)$ is defined as follows:

$$F^\eta_V(d) = \begin{cases} c^{l^\eta_V(d)} & \text{if } d = c^L \\ (F^\eta_V(d_1), F^\eta_V(d_2))^{l^\eta_V(d)} & \text{if } d = (d_1, d_2)^L \\ \{(F^\eta_\Omega(d_i), F^\eta_\Omega(\partial_i)) \mid i \in I\}^{l^\eta_V(d)} & \text{if } d = \{(d_i, \partial_i) \mid i \in I\}^L \end{cases}$$

$$l^\eta_V(d) = \{\alpha \in V \mid d \in \eta(\alpha)\}$$

Using the function $F^\eta_V(.)$ to change the labels of domain elements recursively according to η, we can associate to every element of $[\![t]\!]^q_\eta$ an element of $[\![t]\!]^q_{\overline{\eta}}$, and reciprocally.
The proof is done by induction on the pair (d, t) ordered lexicographically, and by case analysis on t.

- $\underline{t = \alpha.}$ We have

$$\begin{aligned}(d :_\eta \alpha)^q &\iff d \in \eta(\alpha)\\ (F_V^\eta(d) :_{\overline{\eta}} \alpha)^q &\iff F_V^\eta(d) \in \overline{\eta}(\alpha) \iff \alpha \in \mathsf{tags}(F_V^\eta(d))\\ &\iff \alpha \in l_V^\eta(d) \iff (\alpha \in V) \wedge (d \in \eta(\alpha))\end{aligned}$$

  and the result follows since $\alpha \in T$ implies $\alpha \in V$.

- $\underline{t = b.}$ If d is not of the form c^L, then the result is immediate. Otherwise, $(c^L :_\eta b)^q \iff c \in \mathbb{B}(b) \iff (F_V^\eta(c^L) :_{\overline{\eta}} b)^q$.

- $\underline{t = t_1 \times t_2.}$ If d is not of the form $(d_1, d_2)^L$, then the result is immediate. Otherwise, we have

$$\begin{aligned}((d_1, d_2)^L :_\eta t_1 \times t_2)^q &\iff (d_1 :_\eta t_1)^q \wedge (d_2 :_\eta t_2)^q\\ (F_V^\eta((d_1, d_2)^L) :_{\overline{\eta}} t_1 \times t_2)^q &\iff (F_V^\eta(d_1) :_{\overline{\eta}} t_1)^q \wedge (F_V^\eta(d_2) :_{\overline{\eta}} t_2)^q\end{aligned}$$

  and for $i \in \{1, 2\}$, $(d_i :_\eta t_i)^q \iff (F_V^\eta(d_i) :_{\overline{\eta}} t_i)^q$ holds by IH.

- $\underline{t = t_1 \to t_2.}$ If d is not of the form $\{(d_i, \partial_i) \mid i \in I\}^L$, then the result is immediate. Otherwise, we have

$$\begin{aligned}(\{(d_i, \partial_i) \mid i \in I\}^L :_\eta t_1 \to t_2)^q &\iff (\forall i \in I.\, (d_i :_\eta t_1)^q \implies (\partial_i :_\eta t_2)^q)\\ (F_V^\eta(\{(d_i, \partial_i) \mid i \in I\}^L) :_{\overline{\eta}} t_1 \to t_2)^q &\iff (\forall i \in I.\, (F_V^\eta(d_i) :_{\overline{\eta}} t_1)^q \implies (F_V^\eta(\partial_i) :_{\overline{\eta}} t_2)^q)\end{aligned}$$

  and for $i \in I$, both $(d_i :_\eta t_1)^q \iff (F_V^\eta(d_i) :_{\overline{\eta}} t_1)^q$ and $(\partial_i :_\eta t_2)^q \iff (F_V^\eta(\partial_i) :_{\overline{\eta}} t_2)^q$ hold by IH.

- $\underline{t = t_1 \vee t_2.}$ We have

$$\begin{aligned}(d :_\eta t_1 \vee t_2)^q &\iff (d :_\eta t_1)^q \vee (d :_\eta t_2)^q\\ (F_V^\eta(d) :_{\overline{\eta}} t_1 \vee t_2)^q &\iff (F_V^\eta(d) :_{\overline{\eta}} t_1)^q \vee (F_V^\eta(d) :_{\overline{\eta}} t_2)^q\end{aligned}$$

  and the result follows by IH.

- $\underline{t = \neg t'.}$ We have

$$\begin{aligned}(d :_\eta \neg t')^q &\iff \neg(d :_\eta t')^q\\ (F_V^\eta(d) :_{\overline{\eta}} \neg t')^q &\iff \neg(F_V^\eta(d) :_{\overline{\eta}} t')^q\end{aligned}$$

  and the result follows by IH.

- $\underline{t = \mathbb{0}.}$ Immediate since $(d :_\eta \mathbb{0})^q$ can never hold.

□

Finally, using the above lemma and the fact that $t_1 \leq t_2$ is equivalent to $t_1 \setminus t_2 \leq \mathbb{0}$ (and similarly for $\leq^q$), we can prove that the two subtyping relations are equivalent.

Proposition 2.11. *For all types* $t_1, t_2 \in \mathsf{Types}$, $t_1 \leq t_2 \iff t_1 \leq^q t_2$.

Proof. By definition, $t_1 \leq t_2 \iff [\![t_1 \setminus t_2]\!] = \emptyset$, and $t_1 \leq^q t_2 \iff \forall \eta.\ [\![t_1 \setminus t_2]\!]^q_\eta = \emptyset$. By Lemma 2.9, the first statement becomes $t_1 \leq t_2 \iff [\![t_1 \setminus t_2]\!]^q_{\overline{\eta}} = \emptyset$. The implication $t_1 \leq^q t_2 \implies t_1 \leq t_2$ is then immediate. The converse follows from Lemma 2.10, taking $V = \mathsf{vars}(t_1 \setminus t_2)$. □

2.4.2. Stability of subtyping by type substitution

Having proven that the two aforementioned subtyping relations are equivalent, we can now prove that subtyping as defined in Definition 2.6 is preserved by type substitutions. To do this, we first prove the following lemma, introduced by Castagna and Xu [14] in their proof that $\leq^q$ is preserved by type substitutions.

Lemma 2.12. *For every $t \in \mathsf{Types}$, every substitution $\theta : \mathscr{V}^\alpha \to \mathsf{Types}$, and every assignment $\eta : \mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D})$, if η' is defined by $\eta'(\alpha) = [\![\alpha\theta]\!]^q_\eta$, then $[\![t\theta]\!]^q_\eta = [\![t]\!]^q_{\eta'}$.*

Proof. The statement is shown by straightforward induction on the pair (d, t). □

The above Lemma allows us to prove that $\leq^q$ is preserved by type substitutions, which then gives the same result for $\leq$ thanks to Proposition 2.11.

Proposition 2.13. *For every types $t_1, t_2 \in \mathsf{Types}$, if $t_1 \leq t_2$ then $t_1\theta \leq t_2\theta$ for every type substitution θ.*

Proof. By Proposition 2.11, we have $t_1 \leq^q t_2$. By Definition 2.8, this proves $\forall \eta.\ [\![t_1 \setminus t_2]\!]^q_\eta = \emptyset$. Now consider an arbitrary $\theta : \mathscr{V}^\alpha \to \mathsf{Types}$ and an assignment $\eta : \mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D})$. Consider η' defined as $\eta'(\alpha) = [\![\alpha\theta]\!]^q_\eta$. By Lemma 2.12, since $[\![t_1 \setminus t_2]\!]^q_{\eta'} = \emptyset$, we deduce that $[\![(t_1 \setminus t_2)\theta]\!]^q_\eta = \emptyset$. This proves that $t_1\theta \leq^q t_2\theta$, and the result follows from Proposition 2.11. □

2.4.3. Normal forms and type operators

In this subsection, we study several type operators that we will use to define the operational semantics of our gradually-typed languages, and to prove their soundness.

When defining a algorithmic type system or the operational semantics of a language, one often needs to compute the domain or codomain of an expression. With simple types, this task is trivial: since an expression that returns a function necessarily has a type of the form $t_1 \to t_2$, its domain t_1 and codomain t_2 can be easily computed syntactically. With set-theoretic types, this is more difficult, since the algorithmic type of an expression can be an arbitrary succession of intersections and unions of arrow types.

To compute the domain and result type of an arbitrary set-theoretic type t, we first show that t can be put in an equivalent disjunctive normal form satisfying specific properties of uniformity. Then, we define the various operators syntactically on its disjunctive normal form, and prove that they satisfy soundness properties, namely, that the types computed this way are the most precise types that can be derived declaratively.

Remark 2.14.
We distinguish the notions of codomain and result type for set-theoretic types, since the type of the result of an application depends on the type of the argument, while the codomain does not. For example, the codomain of $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$ *is* $\mathsf{Int} \vee \mathsf{Bool}$*, since a function of this type can return both integers and booleans. However, the result type of* $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$ *applied to* Int *is* Int*, since a function of this type will always return integers when applied to integers.*

Disjunctive normal forms for types

The first step is to prove that types can be put in disjunctive normal form, that is, they are equivalent to a union of intersections of *atoms* or negations of atoms, where an atom is either a product, an arrow, a base type, or a type variable.

Formally, we define the following sets:

$$\mathscr{A}_{prod} \stackrel{\text{def}}{=} \{t_1 \times t_2 \mid t_1, t_2 \in \mathsf{Types}\}$$
$$\mathscr{A}_{fun} \stackrel{\text{def}}{=} \{t_1 \to t_2 \mid t_1, t_2 \in \mathsf{Types}\}$$

and we use the metavariable a to range over $\mathscr{A}_{prod} \cup \mathscr{A}_{fun} \cup \mathscr{B} \cup \mathscr{V}^{\alpha}$, the set of *atoms*.

We now define disjunctive normal forms for set-theoretic types.

Definition 2.15 (Disjunctive normal form)**.** *A disjunctive normal form (DNF) is a type* $t \in \mathsf{Types}$ *such that*

$$t \equiv \bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)$$

where I *is a finite set of indices, and* P_i *and* N_i *are finite sets of atoms for every* $i \in I$*.*

We can then prove that every type is equivalent to a disjunctive normal form, which follows from the distributivity properties of union and intersection connectives.

Proposition 2.16 (Existence of DNF)**.** *For every type* $t \in \mathsf{Types}$*, there exists a disjunctive normal form* $\mathscr{N}(t)$ *such that* $t \simeq \mathscr{N}(t)$*.*

Proof. We define two mutually recursive functions on types $\mathscr{N}$ and $\mathscr{N}'$, by mutual induction. This induction is well-founded since no recursive use of the functions appears below

type constructors.

$$
\begin{aligned}
\mathcal{N}(a) &= a \\
\mathcal{N}(t_1 \vee t_2) &= \mathcal{N}(t_1) \vee \mathcal{N}(t_2) \\
\mathcal{N}(\neg t) &= \mathcal{N}'(t) \\
\mathcal{N}(\mathbb{0}) &= \mathbb{0} \\
\mathcal{N}'(a) &= \neg a \\
\mathcal{N}'(t_1 \vee t_2) &= \bigvee_{i \in I, j \in J} \left(\bigwedge_{a \in P_i \cup P_j} a \wedge \bigwedge_{a \in N_i \cup N_j} \neg a \right) \\
&\qquad \text{where } \mathcal{N}'(t_1) = \bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right) \text{ and } \mathcal{N}'(t_2) = \bigvee_{i \in J} \left(\bigwedge_{a \in P_j} a \wedge \bigwedge_{a \in N_j} \neg a \right) \\
\mathcal{N}'(\neg t) &= \mathcal{N}(t) \\
\mathcal{N}'(\mathbb{0}) &= \mathbb{1}
\end{aligned}
$$

Following the distributivity properties of the union, intersection and negation connectives, it is straightforward to verify by mutual induction that $\mathcal{N}(t) \simeq t$ and $\mathcal{N}'(t) \simeq \neg t$, and that $\mathcal{N}(t)$ is a disjunctive normal form. □

Before defining the operators on disjunctive normal forms, we need to go a bit further and introduce uniformity conditions for normal forms. We prove that it is possible to choose the normal form of a type so that every intersection only contains *one* kind of atom (besides type variables). We call such normal forms *uniform disjunctive normal forms* (UDNF).

Definition 2.17 (Uniform disjunctive normal form)**.** *A uniform disjunctive normal form (UDNF) is a disjunctive normal form*

$$t \equiv \bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)$$

such that for every $i \in I$, one of the following conditions holds:

- $P_i \cap \mathscr{B} \neq \emptyset$ *and* $(P_i \cup N_i) \cap (\mathscr{A}_{prod} \cup \mathscr{A}_{fun}) = \emptyset$
- $P_i \cap \mathscr{A}_{prod} \neq \emptyset$ *and* $(P_i \cup N_i) \cap (\mathscr{B} \cup \mathscr{A}_{fun}) = \emptyset$
- $P_i \cap \mathscr{A}_{fun} \neq \emptyset$ *and* $(P_i \cup N_i) \cap (\mathscr{B} \cup \mathscr{A}_{prod}) = \emptyset$

As before, we prove that every type is equivalent to a uniform disjunctive normal form. This result follows from the fact that $\mathbb{1} \simeq \mathbb{1}_{\mathscr{B}} \vee (\mathbb{0} \to \mathbb{1}) \vee (\mathbb{1} \times \mathbb{1})$, which ensures that the intersections in a disjunctive normal form can be made uniform by taking their intersection with $\mathbb{1}$, and distributing the mentioned union.

Proposition 2.18 (Existence of UDNF)**.** *For every type $t \in$ Types, there exists a uniform disjunctive normal form $\mathcal{N}_U(t)$ such that $t \simeq \mathcal{N}_U(t)$.*

Proof. By writing

$$\mathscr{N}(t) = \bigvee_{i \in I} \underbrace{\left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)}_{\mathscr{I}_i}$$

we define

$$\mathscr{N}_U(t) \stackrel{\text{def}}{=} \bigvee_{\substack{i \in I \\ \mathscr{I}_i \wedge \mathbb{1}_{\mathscr{B}} \not\leq \mathbb{0}}} \mathscr{I}_i^{\text{base}} \vee \bigvee_{\substack{i \in I \\ \mathscr{I}_i \wedge (\mathbb{1} \times \mathbb{1}) \not\leq \mathbb{0}}} \mathscr{I}_i^{\text{prod}} \vee \bigvee_{\substack{i \in I \\ \mathscr{I}_i \wedge (\mathbb{0} \to \mathbb{1}) \not\leq \mathbb{0}}} \mathscr{I}_i^{\text{fun}}$$

where for every $i \in I$,

$$\begin{aligned}
\mathscr{I}_i^{\text{base}} &\stackrel{\text{def}}{=} \mathbb{1}_{\mathscr{B}} \wedge \bigwedge_{a \in P_i \cap (\mathscr{B} \cup \mathscr{V}^\alpha)} a \wedge \bigwedge_{a \in N_i \cap (\mathscr{B} \cup \mathscr{V}^\alpha)} \neg a \\
\mathscr{I}_i^{\text{prod}} &\stackrel{\text{def}}{=} (\mathbb{1} \times \mathbb{1}) \wedge \bigwedge_{a \in P_i \cap (\mathscr{A}_{prod} \cup \mathscr{V}^\alpha)} a \wedge \bigwedge_{a \in N_i \cap (\mathscr{A}_{fun} \cup \mathscr{V}^\alpha)} \neg a \\
\mathscr{I}_i^{\text{fun}} &\stackrel{\text{def}}{=} (\mathbb{0} \to \mathbb{1}) \wedge \bigwedge_{a \in P_i \cap (\mathscr{A}_{fun} \cup \mathscr{V}^\alpha)} a \wedge \bigwedge_{a \in N_i \cap (\mathscr{A}_{prod} \cup \mathscr{V}^\alpha)} \neg a
\end{aligned}$$

It is clear that $\mathscr{N}_U(t)$ is a uniform disjunctive normal form by definition. We prove that $\mathscr{N}_U(t) \simeq \mathscr{N}(t)$, which ensures that $\mathscr{N}_U(t) \simeq t$ by Proposition 2.16.
Since $\mathbb{1} \simeq \mathbb{1}_{\mathscr{B}} \vee (\mathbb{0} \to \mathbb{1}) \vee (\mathbb{1} \times \mathbb{1})$, we have, for every $i \in I$:

$$\mathscr{I}_i \simeq \mathscr{I}_i \wedge \mathbb{1} \simeq (\mathscr{I}_i \wedge \mathbb{1}_{\mathscr{B}}) \vee (\mathscr{I}_i \wedge (\mathbb{0} \to \mathbb{1})) \vee (\mathscr{I}_i \wedge (\mathbb{1} \times \mathbb{1}))$$

All that remains is proving the following results:

$$\begin{aligned}
\mathscr{I}_i \wedge \mathbb{1}_{\mathscr{B}} \not\leq \mathbb{0} &\implies \mathscr{I}_i \wedge \mathbb{1}_{\mathscr{B}} \simeq \mathscr{I}_i^{\text{base}} \\
\mathscr{I}_i \wedge \mathbb{1} \times \mathbb{1} \not\leq \mathbb{0} &\implies \mathscr{I}_i \wedge \mathbb{1} \times \mathbb{1} \simeq \mathscr{I}_i^{\text{prod}} \\
\mathscr{I}_i \wedge \mathbb{0} \to \mathbb{1} \not\leq \mathbb{0} &\implies \mathscr{I}_i \wedge \mathbb{0} \to \mathbb{1} \simeq \mathscr{I}_i^{\text{fun}}
\end{aligned}$$

For every $a \in \mathscr{A}_{prod} \cup \mathscr{A}_{fun}$, we have $\mathbb{1}_{\mathscr{B}} \wedge a \leq \mathbb{0}$. This guarantees that if $\mathscr{I}_i \wedge \mathbb{1}_{\mathscr{B}} \not\leq \mathbb{0}$, then necessarily $P_i \subseteq \mathscr{B} \cup \mathscr{V}^\alpha$. Moreover, this also ensures that $\mathbb{1}_{\mathscr{B}} \wedge \bigwedge_{a \in N_i \cap (\mathscr{A}_{prod} \cup \mathscr{A}_{fun})} \neg a \simeq \mathbb{1}_{\mathscr{B}}$. Hence the first implication. The other two can be proven using the same reasoning. □

This very last result ensures that, for example, a type $t \leq \mathbb{0} \to \mathbb{1}$ can always be written equivalently as a disjunctive normal form containing only arrows, negation of arrows, and type variables. This will make the definition of operators much easier in the following paragraphs.

Type operators

Having defined (uniform) disjunctive normal forms, we can now define the various type operators we will use throughout this manuscript. We define three of them: the domain operator, the result type operator, and the projection operator.

The first we define is the domain operator, which computes the domain of a function type (that is, a type $t \leq \mathbb{0} \to \mathbb{1}$).

Definition 2.19 (Domain operator)**.** *For every type $t \leq \mathbb{0} \to \mathbb{1}$, we define its* domain $\mathsf{dom}(t)$ *as:*

$$\mathsf{dom}(t) \stackrel{\text{def}}{=} \bigwedge_{i \in I} \bigvee_{s_i \to t_i \in P_i} s_i$$

where

$$\mathscr{N}_U(t) \simeq \bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)$$

To compute the domain of a normal form, only the positive arrows are used. The intuition is that negative arrows do not characterize the input of functions, only their outputs. For example, if a function is of type $(\mathbb{1} \to \mathbb{1}) \wedge \neg(\mathsf{Int} \to \mathsf{Bool})$, then it can be applied to any argument. In particular, the negation type $\neg(\mathsf{Int} \to \mathsf{Bool})$ does not forbid the function to be applied to an integer, it only states that, when applied to *some* integers (but not necessarily all), the function returns something that is not a boolean.

The definition of the domain operator then follows the following intuition: the domain of an intersection of arrows is the union of their domains, and the domain of a union of arrows is the intersection of their domains. Intuitively, a function of type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$ can be applied to both integers and booleans, while a function of type $(\mathsf{Even} \to \mathsf{Even}) \vee (\mathsf{Nat} \to \mathsf{Nat})$ can only be safely applied to integers that are both even and non-negative (i.e., $\mathsf{Even} \wedge \mathsf{Nat}$), because we cannot be sure whether it is of one type or the other.

The next operator is the result type operator, which is much more complex. As anticipated, since the type of the result of an application depends on the type of the argument, this operator depends on both the type of the function and the type of the argument.

Definition 2.20 (Result type operator)**.** *For every type $t \leq \mathbb{0} \to \mathbb{1}$ and every type s such that $s \leq \mathsf{dom}(t)$, we define the* result type *of t applied to s, noted $t \circ s$ as:*

$$t \circ s \stackrel{\text{def}}{=} \bigvee_{i \in I} \bigvee_{\substack{Q \subsetneq P_i \\ s \not\leq \bigvee_{s_i \to t_i \in Q} s_i}} \bigwedge_{s_i \to t_i \in P_i \setminus Q} t_i$$

where

$$\mathscr{N}_U(t) = \bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)$$

To understand this operator, first consider a simple intersection of arrows $t \equiv \bigwedge_{p \in P} s_p \to t_p$ (for the same reasons as before, negation types do not play any role when computing the result type of an application). If the type of the argument s intersects the domain of several arrows, that is $s \wedge s_{p_1} \wedge \ldots \wedge s_{p_n} \not\leq \mathbb{0}$ for $p_1, \ldots, p_n \in P$, then the application *may* return a result in the intersection of all arrows, that is, $t_{p_1} \wedge \ldots \wedge t_{p_n}$, provided the argument resolves to a value that is indeed in the intersection of the aforementioned domains. However, the argument may also resolve to a value that is outside the intersection of these domains. Thus, we need to consider all the possible intersections between the type s and subsets of $\{s_p \mid p \in P\}$, and take the union of all the possible results. This is what the operator does: by removing all subsets Q that do not entirely contain s, it obtains all the subsets $P_i \setminus Q$ of P_i that have a non-empty intersection with s, and computes the result.

Finally, to generalize this result to a union of intersections of arrows, we just need to consider that the result of a union is the union of the results. For example, an expression of type $(\mathsf{Nat} \to$

Nat) ∨ (Even → Even) applied to an argument of type Nat ∧ Even will produce a result of type Nat ∨ Even depending on whether the function resolves to a function of type Nat → Nat or a function of type Even → Even.

The last operator we define is the projection operator, which computes the projection of a type $t \leq \mathbb{1} \times \mathbb{1}$.

Definition 2.21 (Projection operator)**.** *For every type $t \leq \mathbb{1} \times \mathbb{1}$, and every $j \in \{1, 2\}$, we define the* j-th projection *of t, noted $\pi_j(t)$ as:*

$$\pi_j(t) \stackrel{\text{def}}{=} \bigvee_{i \in I} \bigwedge_{t_1^i \times t_2^i \in P_i} t_j^i$$

where

$$\mathcal{N}_U(t) = \bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)$$

Compared to the previous operators, the definition of the projection operator is quite straightforward. To understand it, simply remark that the projection of an intersection of products is the intersection of their projections, and similarly for an union of products. The former actually comes immediately from the interpretation of types: $(t_1 \times t_2) \wedge (t_1' \times t_2')$ is equivalent to $(t_1 \wedge t_1') \times (t_2 \wedge t_2')$. The latter is quite intuitive: if an expression has type $(\mathsf{Int} \times \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool})$, then its first projection will either reduce to an integer (if the expression reduces to a value of type Int × Int) or to a boolean (if the expression reduces to a value of type Bool × Bool). Hence, its first projection is of type Int ∨ Bool.

Naturally, proving the soundness of systems involving these operators will require some properties about these operators. We prove that each operator computes the most precise type (corresponding to the operator) that can be derived declaratively. For example, if an expression of type t can be given type $t_1 \to t_2$ by subsumption, then necessarily $t_1 \leq \mathsf{dom}(t)$. That is, a declarative type system with standard rules and a subsumption rule will never deduce an arrow type with a domain more precise than the type computed by the domain operator.

As the proof of these properties require very involved manipulation of normal forms, we refer the reader to Frisch et al. [27] for the proofs, and only state the properties.

Proposition 2.22. *For all types $t, t' \in \mathsf{Types}$, if $t \leq \mathbb{0} \to \mathbb{1}$ then $t \leq \mathsf{dom}(t) \to \mathbb{1}$ and if $t \leq t' \to \mathbb{1}$ then $t' \leq \mathsf{dom}(t)$.*

Proposition 2.23. *For all types $t, t', s \in \mathsf{Types}$, if $t \leq \mathbb{0} \to \mathbb{1}$ and $s \leq \mathsf{dom}(t)$ then $t \leq s \to t \circ s$. Moreover, if $t \leq s \to t'$ then $t \circ s \leq t'$.*

Proposition 2.24. *For all types $t, t_1, t_2 \in \mathsf{Types}$, if $t \leq \mathbb{1} \times \mathbb{1}$ then $t \leq \pi_1(t) \times \pi_2(t)$ and if $t \leq t_1 \times t_2$ then $\pi_1(t) \leq t_1$ and $\pi_2(t) \leq t_2$.*

Part I.

A declarative approach to gradual typing

Chapter 3.

Introduction

The first part of this thesis is devoted to the study of several gradual type systems, from a Hindley-Milner type system with implicit polymorphism but without subtyping, to a system supporting full-fledged set theoretic types and semantic subtyping. Initially, this work started as an attempt to add polymorphism and type inference to a set-theoretic gradual type system we presented in Castagna and Lanvin [13]. However, this led to a novel approach in which polymorphism facilitated the introduction of gradual typing into existing type systems, independently of the presence of set-theoretic types.

As such, we consider two type systems: an ML-like type system and a set-theoretic type system, to which we add gradual typing following this new approach. We study each system declaratively using structural rules, before defining the associated semantics and compilation. Then, we study the algorithmic aspects of typing, by designing type inference algorithms for both systems.

3.1. Gradual typing, set-theoretic types, and polymorphism

Combining gradual typing with union and intersection types yields many practical benefits, especially in a polymorphic setting. For a preview of what can be done in this setting, consider the following ML-like code snippet adapted from Siek and Vachharajani [67]:

```
let mymap (condition) (f) (x : ?) =
    if condition then Array.map f x else List.map f x
```

According to the value of the argument `condition`, the function `mymap` applies either the array version or the list version of `map` to the other two arguments. This example cannot be typed using only simple types: the type of `x` and the return type of `mymap` change depending on the value of `condition`. By annotating `x` with the gradual type `?`, the type reconstruction system for gradual types of Siek and Vachharajani [67] can type this piece of code with $\mathsf{Bool} \to (\alpha \to \beta) \to ? \to ?$. That is, type reconstruction recognizes that the parameter `condition` must be bound to a Boolean value, and the compilation process adds dynamic checks to ensure that the value bound to `x` will be, according to the case, either an array or a list whose elements are of a type compatible with the actual input type of `f`.

This type however is still imprecise. For example, if we pass a value that is neither an array nor a list (e.g., an integer) as the last argument to `mymap`, then this application is well-typed, even though the execution will always fail, independently of the value of `condition`. Likewise, the type gives no useful information about the result of `mymap`, even though it will clearly be either a β-list or a β-array. These problems can be remedied by using set-theoretic types:

```
let mymap (condition) (f) (x : (α array ∨ α list) ∧ ?) =
    if condition then Array.map f x else List.map f x
```

The union indicates that a value of this type is *either* an array *or* a list, both of α-elements. The intersection indicates that `x` has *both* type $(\alpha\,\texttt{array} \vee \alpha\,\texttt{list})$ *and* type `?`. Intuitively, this type annotation means that the function `mymap` accepts for `x` a value of any type (which is indicated by `?`), as long as this value is also either an array *or* a list of α elements, with α being the domain of the `f` argument. The use of the intersection of a union type with "`?`" to type a parameter corresponds to a programming style in which the programmer asks the system to *statically* enforce that the function will be applied only to arguments in the union type and delegates to the system any *dynamic* check regarding the use of the parameter in the body of the function. The system presented in Section 5.4 can deduce for this definition the type:

$$\mathsf{Bool} \to (\alpha \to \beta) \to ((\alpha\,\texttt{array} \vee \alpha\,\texttt{list}) \wedge \texttt{?}) \to (\beta\,\texttt{array} \vee \beta\,\texttt{list})$$

This type forces the last argument of `mymap` to be either an array or a list of elements whose type is the input type of the argument bound to `f`. Note that the return type of `mymap` is no longer gradual (as it was with the previous definition), since the union type allows us to define it without any loss of precision, as well as to capture the correlation with the return type of the argument bound to `f`. The derivation of this type is used by the compiler to insert dynamic type-checks that ensure type soundness. In particular, the compilation process described in Section 4.2 inserts in the body of `mymap` the casts that dynamically check that the first occurrence of `x` in the body of the function is bound to an array of elements of the appropriate type, and that the second occurrence of `x` is bound to a list of such elements, producing a code like the following:

```
let mymap (condition) (f) (x : (α array ∨ α list) ∧ ?) =
    if condition then Array.map f (x<α array>) else List.map f (x<α list>)
```

where *e*<*t*> is a type-cast expression that dynamically checks whether the result of *e* has type *t*.

This kind of type discipline is out of reach of current systems. To obtain it, in this part of the thesis we explore a new idea to interpret gradual types, namely, that the unknown type `?` acts like a type variable, but a peculiar one since each occurrence of `?` in a typing constraint can be considered as a placeholder for a possibly distinct type variable.

3.2. Our approach

Most current presentations of gradual type systems rely on either *consistency* or *consistent subtyping*, the latter being the combination of consistency and subtyping on static types. Consistency, usually denoted $\sim$, is a reflexive and symmetric but non-transitive relation such that $\tau_1 \sim \tau_2$ holds whenever τ_1 and τ_2 are syntactically equal everywhere except where `?` occurs. It is often defined using the following inference rules:

$$\frac{}{\texttt{?} \sim \tau} \qquad \frac{}{\tau \sim \texttt{?}} \qquad \frac{\tau_1 \sim \tau_1' \quad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}$$

According to these rules, it is clear that the transitive closure of $\sim$ is the total relation on gradual types, since every type τ is consistent with `?`, which is itself consistent with every type. This means that, as opposed to subtyping, consistency cannot be added to a type system via a structural rule such as:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'}\ \tau \sim \tau'$$

since such a type system would be able to type every program, even those that do not contain ?. Therefore, consistency (or consistent subtyping) is usually embedded directly into elimination rules. For example, the rule for applications is replaced by the two following rules:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \; \tau_1 \sim \tau_2 \qquad \frac{\Gamma \vdash e_1 : ? \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : ?}$$

The second rule is needed because ? is consistent with $\tau_2 \to ?$ for every type τ_2, therefore a value of type ? can be applied to a value of type τ_2.

Such a presentation is not specific to gradual typing: when studying algorithmic type systems for languages with subtyping, it is customary to embed the subtyping relation into elimination rules, to mimic the behaviour of a static type checker. However, the use of a subsumption rule provides an easy and concise way of adding subtyping into an existing type system, and to gain insight into how a type system behaves in presence of subtyping.

In this work, we show that such an approach is also possible with gradual typing, and we describe what is, to our knowledge, the first presentation of a gradual type system that relies entirely on a single structural rule to introduce gradual typing. When combined with semantic subtyping, this gives us some insight into the interaction between gradual types and set-theoretic types, which sometimes proves to be counter-intuitive.

At the core of our approach is the notion that every occurrence of ? behaves as a type variable, possibly distinct from all the others. We formalize it by defining an operation of *discrimination* which replaces each occurrence of ? in a gradual type by a type variable. Our semantics for gradual types relies on this operation, for two major reasons: first, it allows us to formalize the notion of substituting an occurrence of ? with a type, which is the main idea behind consistency. Second, by applying discrimination we map a polymorphic gradual type into a set of polymorphic static types, one for each possible replacement of occurrences of the dynamic type by a type variable. Then, we can use pre-existing notions on static types (for example, the semantic subtyping interpretation) to interpret, indirectly, our initial gradual type.

Using discrimination, we define a relation we call *precision* and denote $\preccurlyeq$, which is a preorder on gradual types. Given two types τ_1 and τ_2, $\tau_1 \preccurlyeq \tau_2$ when τ_2 is *more precise* than τ_1, that is, if it was obtained from τ_1 by replacing some occurrences of ? by some gradual types. This is a relation that occurs frequently in the gradual typing literature, sometimes reversed. Siek and Vachharajani [67] call this relation "less or equally informative" while Garcia [29] and others use the inverse relation $\sqsubseteq$ which they also call "precision".

Since this relation is a preorder, it can be used in a structural rule. We argue that adding the rule

$$[\mathrm{T}_{\mathsf{Mater}}] \; \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'} \; \tau \preccurlyeq \tau'$$

to any static type system is enough to endow it with gradual typing. In particular, we show that any type derivation in a system where the consistency relation is embedded into elimination rules can be obtained in our system by applying $[\mathrm{T}_{\mathsf{Mater}}]$ using a precise strategy. We refer to this rule as *materialization*, by analogy with the *subsumption* rule for subtyping.

Thanks to this approach, adding subtyping is now only a matter of also adding a subsumption rule for subtyping. This contrasts with the complexity of existing approaches where consistency (or precision) and subtyping must be combined to define consistent subtyping. However, this

requires to extend the subtyping relation from static types to gradual types. In the absence of set-theoretic types, we achieve this by following an existing approach (presented notably by Siek and Taha [66]) in which ? is simply treated as a new base type that is incompatible with every other type for subtyping. That is, the subtyping relation $\leq$ is defined such that $? \leq ?$ holds (by reflexivity), but not $? \leq \mathsf{Int}$ or $\mathsf{Int} \leq ?$. This contrasts with precision where ? acts as a bottom type, which is to be expected: the two relations have very different purposes. The role of subtyping is to determine whether an expression can *always be safely used* in a given context, while the role of precision is to determine whether *there is a possibility* that using an expression in a given context is safe.

However, defining a subtyping relation for gradual set-theoretic types is more challenging. The dynamic type cannot be considered as a separate base type: for reasons we will explain in Chapter 5, $? \setminus ?$ must not be empty, which would be the case if ? were interpreted as a base type. Instead, we use once again the notion of discrimination to interpret gradual set-theoretic types as set-theoretic types with variables, which we restrict to ensure that two occurrences of ? are not replaced with the same type variable if they do not occur under the same number of negations. Using this interpretation, deciding subtyping on gradual types reduces to deciding subtyping on static types, which can be decided using semantic subtyping.

This approach has another advantage: since precision is expressed in terms of type substitutions and gradual subtyping reduces to static subtyping, we can describe type inference for gradual typing by directly reusing existing algorithms for static type systems, where an additional unification step takes care of solving precision constraints.

Finally, our approach sheds some light on the logical meaning of gradual typing. It is well-known that there is a strong correspondence between systems with subtyping and systems without subtyping but with explicit coercions: every usage of the subsumption rule in the former corresponds to the insertion of an explicit coercion in the latter. Our definition of precision yields an analogous correspondence between a gradually-typed language and the cast calculus in which the language is compiled: every usage of the materialization rule in the former corresponds to the insertion of an explicit cast in the latter. As such, the cast language looks like an important ingredient for a Curry-Howard isomorphism for gradual typing disciplines. An intriguing direction for future work is to study the logic associated with these expressions.

3.3. Overview

As a first step, in Chapter 4, we add gradual typing to ML-like languages. We start by giving the definition of precision for simple gradual types, and use it to present a declarative static semantics for a gradually-typed version of ML. As customary for gradually-typed languages, we give the dynamic semantics by compiling well-typed terms into a cast calculus, which we prove to be sound. We then study the algorithmic aspects of typing, by defining a constraint-based type inference algorithm that we prove to be sound and complete. Finally, we briefly discuss the extension of the system with subtyping, by considering a simple subtyping relation without set-theoretic types. However, the presence of subsumption makes type inference more difficult since, in particular, constraint resolution involves computing intersections and unions of types. Therefore, we postpone the algorithmic aspects of this system to the next chapter.

In Chapter 5, we introduce set-theoretic types, whose interpretation we base on the approach of semantic subtyping. We add union, negation, and intersection (which are encoded by De Morgan's laws) to types, as well as recursive types (which are needed for the inference algorithm). We describe two main challenges. The first consists in defining a suitable subtyping relation for

gradual set-theoretic types. We achieve this by using our interpretation of ? as type variables, and lift the subtyping relation from static types to gradual types. The second challenge is the extension of the cast calculus with set-theoretic types. We propose a first solution which, albeit complex, enjoys the usual type safety properties, and is a conservative extension of the approach presented in Chapter 4.

In Chapter 6, we propose a second approach, based on recent results that will be developed in the second part of this thesis. By introducing new relations, which we call *semantic gradual subtyping* and *semantic precision*, we obtain strong results about the representation of set-theoretic gradual types, which greatly ease reasoning about them. The semantics we obtain using this approach is quite different from the first, but is much simpler, and enjoys the same soundness properties.

We follow with a brief conclusion in Chapter 7, in which we discuss our main results, compare our approach to existing work, and highlight some interesting directions for future work.

Apart from the semantics presented in Chapter 6, the work presented in this part of the thesis was the subject of an article (Castagna et al. [18]) in which we follow the same presentation, providing all the details about the semantics of the cast calculi and the inference algorithms. In this manuscript, we only give a brief overview of the challenges we encountered when defining the inference algorithms, and provide a quick rundown of our solutions. We refer the interested reader to the appendix for the full definitions of the inference systems, and to the cited work for the proofs of all results.

The reason for this omission is twofold: first, the goal of this thesis is to study the semantics of gradual types and gradually-typed languages, and type inference is a rather independent problem. Second, the type inference algorithms have been developed in collaboration with and thoroughly explained by Petrucciani [56], and presenting them here would be redundant at best.

Chapter 4.

Gradual typing for Hindley-Milner systems

"La gran victoria de hoy fue el resultado de pequeñas victorias que pasaron desapercibidas."

Paulo Coelho

In this chapter, we present our interpretation of gradual types as types with polymorphic type variables. As an example, we use this approach to add gradual typing to a language with ML-style polymorphism, both declaratively and algorithmically.

Chapter Outline

Section 4.1 We introduce our source language, a standard λ-calculus equipped with pairs and a let construct. We then present the *precision* relation, and use it to define a declarative gradual type system for our source language. We follow with a comparison of our type system to several existing type systems.

Section 4.2 We present the syntax and semantics of the target language (or cast language) associated with our source language. This is done by adding casts and explicit type substitutions to our source language. We also show how precision can be used to provide a very simple declarative compilation system for our languages. Finally, we present several properties of our calculi, revisiting commonly proven properties of gradual languages, namely type soundness, blame safety, and gradual guarantee.

Section 4.3 We complete our presentation with the definition of a type inference algorithm, which allows us to decide whether a term of the source language is well-typed or not, and to compile it to a term of the cast language by adding the necessary checks. We state soundness and completeness properties for this algorithm with respect to the type system presented in the preceding sections.

Section 4.4 Finally, we hint at a way to add subtyping to our language. While this is easily done declaratively, we show that type inference becomes considerably more difficult.

4.1. Source language

This chapter starts with the definition of the source language, its declarative type system, and the presentation of some of its properties.

4.1.1. Syntax and types

The source language manipulates both static simple types STypes and gradual simple types GTypes, which are defined inductively by the follow grammar:

$$\begin{array}{rrll} \mathsf{STypes} \ni t & ::= & \alpha \mid b \mid t \times t \mid t \to t & \text{static simple types} \\ \mathsf{GTypes} \ni \tau & ::= & ? \mid \alpha \mid \tau \times \tau \mid \tau \to \tau & \text{gradual simple types} \end{array}$$

where b ranges over a set $\mathscr{B}$ of *basic types* (e.g., $\mathscr{B} = \{\mathsf{Int}, \mathsf{Bool}\}$), and α, β, and γ range over a countable set $\mathscr{V}^{\alpha}$ of *type variables*. The static types STypes (ranged over by t), are the types of an ML-like language: type variables, basic types, products, and arrows. Gradual types GTypes (ranged over by τ) add the unknown type ? to them.

The expressions of the source language $\mathsf{Terms}^{\mathsf{HM}}$ are those defined inductively by the following grammar:

$$\mathsf{Terms}^{\mathsf{HM}} \ni e ::= x \mid c \mid \lambda x.\, e \mid \lambda x{:}\tau.\, e \mid e\, e \mid (e, e) \mid \pi_i\, e \mid \mathsf{let}\ \vec{\alpha}\ x = e\ \mathsf{in}\ e$$

where x ranges over a countable set of variables Vars, and c ranges over a set $\mathscr{C}$ of constants. This grammar corresponds to a fairly standard λ-calculus with constants, pairs (e, e), projections for the elements of a pair $\pi_i\, e$ (where $i \in \{1, 2\}$), plus a let construct. There are two aspects to point out.

One is that there are two forms of λ-abstractions: annotated abstractions $\lambda x{:}\tau.\, e$ and unannotated ones $\lambda x.\, e$. In the former, the annotation τ fixes the type of the argument, whereas in the latter the type can be chosen during typing (and will, in practice, be computed by type inference). Furthermore, the type τ in the explicitly annotated λ-abstraction is gradual, while we require that the inferred type of the parameter of an unannotated abstraction be a static type t (cf. Figure 4.1, Rule $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Abstr}}]$). This restriction is common in the gradual typing literature, for example in [30], and is necessary to properly reject some ill-typed programs. For example, without this restriction, we could type $\lambda x.\, (x + 1, \neg x)$ since it would be possible to infer the type ? for x, so as to deduce for $\lambda x.\, (x + 1, \neg x)$ the type $? \to \mathsf{Int} \times \mathsf{Bool}$. But $\lambda x.\, (x + 1, \neg x)$ is not a well-typed term in ML, therefore, by the principles of gradual typing (see Theorem 1 of Siek et al. [69]) it must be rejected unless its parameter is explicitly annotated by a type in which ? occurs (here, it must be annotated by ? itself).

The second non-standard element of this syntax is that the let binding is decorated with a vector $\vec{\alpha}$ of type variables, as in $\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2$. This *decoration* (we reserve the word *annotation* for types annotating parameters in λ-abstractions) serves as a binder for the type variables that appear in annotations *occurring in* e_1. For instance, $\mathsf{let}\ \alpha\ z = \lambda x{:}\alpha.\, x\ \mathsf{in}\ e$ and $\mathsf{let}\ z = \lambda x.\, x\ \mathsf{in}\ e$ are equivalent, while the expression $\mathsf{let}\ z = \lambda x{:}\alpha.\, x\ \mathsf{in}\ e$ implies that α was introduced in the annotation of an outer expression such as $\lambda y{:}\alpha.\, \mathsf{let}\ z = \lambda x{:}\alpha.\, x\ \mathsf{in}\ e$. The normal let from ML can be recovered as the case where $\vec{\alpha}$ is empty which would be the case if, as in ML, function parameters never had type annotations.

As is customary, we consider expressions modulo α-renaming of bound variables. In $\lambda x.\, e$ and $\lambda x{:}\tau.\, e$, x is bound in e; in $\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2$, x is bound in e_2 and the variables of $\vec{\alpha}$ are bound in e_1. We denote by $\vec{\alpha} \sharp e$ the fact that the variables of $\vec{\alpha}$ do not occur free in e. It is also customary to refer to the source language as the *gradually-typed language*.

$$[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Cst}}]\ \frac{}{\Gamma \vdash c : b_c} \qquad [\mathrm{T}^{\mathsf{HM}}_{\mathsf{Var}}]\ \frac{}{\Gamma \vdash x : \tau\,[\vec{t}/\vec{\alpha}]}\ \Gamma(x) = \forall\vec{\alpha}.\tau$$

$$[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Proj}}]\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\, e : \tau_i} \qquad [\mathrm{T}^{\mathsf{HM}}_{\mathsf{Pair}}]\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad [\mathrm{T}^{\mathsf{HM}}_{\mathsf{App}}]\ \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\, e_2 : \tau}$$

$$[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Abstr}}]\ \frac{\Gamma, x : t \vdash e : \tau}{\Gamma \vdash \lambda x.\, e : t \to \tau} \qquad [\mathrm{T}^{\mathsf{HM}}_{\mathsf{AAbstr}}]\ \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x{:}\tau'.\, e : \tau' \to \tau}$$

$$[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Let}}]\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \forall\vec{\alpha}, \vec{\beta}.\tau_1 \vdash e_2 : \tau}{\Gamma \vdash (\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2) : \tau}\ \vec{\alpha}, \vec{\beta} \sharp \Gamma \textit{ and } \vec{\beta} \sharp e_1$$

$$[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Mater}}]\ \frac{\Gamma \vdash e_1 : \tau'}{\Gamma \vdash e_2 : \tau}\ \tau' \preccurlyeq \tau$$

FIGURE 4.1. Declarative type system of the source language

4.1.2. Precision and type system

One of the main aspects of our approach is that adding gradual typing to a standard Hindley-Milner type system can be done declaratively in a very simple way. We now illustrate this by presenting the declarative type system of the source language.

Type system

We use the standard notion for type schemes and type environments. A type scheme has the form $\forall\vec{\alpha}.\tau$, where $\vec{\alpha}$ is a vector of distinct variables. We identify type schemes with an empty $\vec{\alpha}$ with gradual types. A type environment Γ is a finite function from variables to type schemes.

The type system is defined by the rules in Figure 4.1, using standard statements of the form $\Gamma \vdash e : \forall\vec{\alpha}.\tau$ (the quantification being omitted whenever $\vec{\alpha}$ is empty).

The first eight rules are almost those of a standard Hindley-Milner type system. In $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Cst}}]$, we use b_c to denote the basic type for a constant c (e.g., $b_3 = \mathsf{Int}$). One important aspect to note is that, for reasons discussed earlier, the types used to instantiate the type scheme in $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Var}}]$ and the type used for the domain in $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Abstr}}]$ must all be static types, as forced by the use of the metavariable t.

The other non-standard aspect is the rule for $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Let}}]$. To type let $\vec{\alpha}\ x = e_1$ in e_2, we type e_1 with some type τ_1; then, we type e_2 in the expanded environment in which x has type $\forall\vec{\alpha},\vec{\beta}.\tau_1$. The first side condition ($\vec{\alpha},\vec{\beta} \sharp \Gamma$) asks that all the variables we generalize do not occur free in Γ; this is standard when implementing let-polymorphism. The second condition ($\vec{\beta} \sharp e_1$) states that the type variables $\vec{\beta}$ must not occur free in e_1. This means that the type variables that are explicitly introduced by the programmer (by using them in annotations) can only be generalized at the level of a let binding by explicitly specifying them in the decoration. In contrast, type variables introduced by the type system (i.e., the fresh variables in the static type t in Rule $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Abstr}}]$) can be generalized at any let (implicitly, that is, by the type system), provided they do not occur in the environment. Note that we recover the standard Hindley-Milner rule for let bindings when expressions do not contain annotations and decorations are empty.

As anticipated, the type system does not need to deal with gradual types explicitly except in one rule. Indeed, the first eight rules do not check anything regarding gradual types (they only impose restrictions that some types must be static). The last rule, $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Mater}}]$ which we call *materialization*, is a subsumption-like rule that allows us to make any gradual type more precise by replacing occurrences of ? with arbitrary gradual types. This is accomplished by the *precision* relation $\preccurlyeq$, which we define next.

Precision

Intuitively, $\tau_1 \preccurlyeq \tau_2$ holds when τ_2 can be obtained from τ_1 by replacing some occurrences of ? with arbitrary gradual types, possibly different for every occurrence. This relation can be easily defined by the following inductive rules, which add the reflexive case for type variables to the rules of Siek and Vachharajani [67]:[1]

$$\frac{}{? \preccurlyeq \tau} \qquad \frac{}{\alpha \preccurlyeq \alpha} \qquad \frac{}{b \preccurlyeq b}$$

$$\frac{\tau_1 \preccurlyeq \tau_1' \quad \tau_2 \preccurlyeq \tau_2'}{(\tau_1, \tau_2) \preccurlyeq (\tau_1', \tau_2')} \qquad \frac{\tau_1 \preccurlyeq \tau_1' \quad \tau_2 \preccurlyeq \tau_2'}{\tau_1 \to \tau_2 \preccurlyeq \tau_1' \to \tau_2'}$$

However, this definition is intrinsically tied to the syntax of types. Instead, we want the definition of precision to remain valid also when we extend the language of types we use. Therefore, we give a definition based on our view, anticipated earlier, of occurrences of ? as type variables.

First, let us define a new sort of types, named *type frames*, as follows:

$$\mathsf{TFrames} \ni T \quad ::= \quad X \mid \alpha \mid b \mid t \times t \mid t \to t \quad \text{simple type frames}$$

where X ranges over a set $\mathscr{V}^X$ of *frame variables* disjoint from $\mathscr{V}^\alpha$. Type frames are like gradual types except that, instead of ?, they have frame variables. We write TFrames for the set of all type frames.

Additionally, we write $\mathsf{vars}(T)$ to denote the set of variables in $\mathscr{V}^X \cup \mathscr{V}^\alpha$ occurring in a type frame T, and we denote by $\mathsf{vars}^X(T)$ and $\mathsf{vars}^\alpha(T)$ the sets $\mathsf{vars}(T) \cap \mathscr{V}^X$ and $\mathsf{vars}(T) \cap \mathscr{V}^\alpha$ respectively. We also extend $\mathsf{vars}(.)$ to gradual types, static types, type schemes, as well as type environments, to denote the set of free type variables occurring in them.

Given a type frame T, we write $T^\dagger$ for the gradual type obtained by replacing all frame variables in T with ?. The reverse operation, which we call *discrimination*, is the function $\star(.) : \mathsf{GTypes} \to \mathsf{TFrames}$ defined as follows:

> **Definition 4.1** (Discrimination of a gradual type)**.** *Given a gradual type $\tau \in \mathsf{GTypes}$, the set $\star(\tau)$ of its discriminations is defined as:* $\star(\tau) \stackrel{\text{def}}{=} \{T \in \mathsf{TFrames} \mid T^\dagger = \tau\}$.

The definition of precision, stated formally below, says that τ_2 is more precise than τ_1 if it can be obtained from τ_1 by first replacing all occurrences of ? with arbitrary variables in $\mathscr{V}^X$, and then applying a substitution which replaces those variables with gradual types.

[1] Henglein [34] defines an equivalent relation for monomorphic types (called "subtyping") but with different rules.

Definition 4.2 (Precision). *We define the precision relation on gradual types $\tau_1 \preccurlyeq \tau_2$ ("τ_2 is more precise than τ_1") as follows:*

$$\tau_1 \preccurlyeq \tau_2 \overset{\text{def}}{\iff} \exists T \in \star(\tau_1), \theta : \mathscr{V}^X \to \mathsf{GTypes}.\ T\theta = \tau_2$$

In the above definition, $\theta : \mathscr{V}^X \to \mathsf{GTypes}$ is a type substitution (i.e., a mapping that is the identity on a cofinite set of variables) from frame variables to gradual types. We use $\mathsf{dom}(\theta)$ to denote the set of variables for which θ is not the identity (i.e., $\mathsf{dom}(\theta) = \{X \mid X\theta \neq X\}$).

As a reference to the materialization rule, we may also say that τ_1 *materializes into* τ_2 whenever $\tau_1 \preccurlyeq \tau_2$. We may also refer to τ_2 as a *materialization* of τ_1 in this case.

It is not difficult to prove that the precision relation of Definition 4.2 and the one deduced by the inductive rules that we have given in the previous page are equivalent, and that they are inverses of the precision relation introduced by Garcia [29] and of naive subtyping [75].

4.1.3. Comparison to existing relations

In their approach on *abstracting gradual typing*, Garcia et al. [31] propose to add gradual typing to existing type systems using abstract interpretation. The cornerstone of their work is the definition of the set of concretizations of a gradual type. They define this concretization as a function $\gamma : \mathsf{GTypes} \to \mathscr{P}(\mathsf{STypes})$ which syntactically replaces *all* occurrences of ? in a gradual type by (possibly distinct) static types. Formally, the concretization function γ is defined by Garcia et al. [31] syntactically as follows:

$$\begin{array}{rcl}
\gamma : \mathsf{GTypes} & \to & \mathsf{STypes} \\
\gamma(b) & = & \{b\} \\
\gamma(\alpha) & = & \{\alpha\} \\
\gamma(?) & = & \mathsf{STypes} \\
\gamma(\tau_1 \to \tau_2) & = & \{t_1 \to t_2 \mid t_1 \in \gamma(\tau_1), t_2 \in \gamma(\tau_2)\} \\
\gamma(\tau_1 \times \tau_2) & = & \{t_1 \times t_2 \mid t_1 \in \gamma(\tau_1), t_2 \in \gamma(\tau_2)\}
\end{array}$$

In our setting, it is straightforward to prove the following result, stating that the concretizations of a gradual type are the static types that can be obtained from it by precision:

Proposition 4.3. *For every gradual type $\tau \in \mathsf{GTypes}$, $\gamma(\tau) = \{t \in \mathsf{STypes} \mid \tau \preccurlyeq t\}$.*

Proof. Straightforward by induction on τ. □

The precision relation can also be obtained from the concretization operation, by syntactically comparing the concretizations of two types. Formally, we have the following result:

Proposition 4.4. *For all gradual types $\tau_1, \tau_2 \in \mathsf{GTypes}$, $\tau_1 \preccurlyeq \tau_2 \iff \gamma(\tau_2) \subseteq \gamma(\tau_1)$.*

Proof. Straightforward by induction on τ, using the inductive rules for $\preccurlyeq$. □

This last result allows us to define, using only precision, all the relations commonly used in the gradual typing literature, following the approach of Garcia et al. [31]. For example, consistency,

a relation on which most of the existing work on gradual typing is based, relates two gradual types whenever they only differ syntactically where ? occurs.

Formally, consistency is defined for our types by the following inductive rules:

$$\frac{}{? \sim \tau} \qquad \frac{}{\tau \sim ?} \qquad \frac{}{\alpha \sim \alpha} \qquad \frac{}{b \sim b} \qquad \frac{\tau_1 \sim \tau_1' \quad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'} \qquad \frac{\tau_1 \sim \tau_1' \quad \tau_2 \sim \tau_2'}{\tau_1 \times \tau_2 \sim \tau_1' \times \tau_2'}$$

Garcia et al. [31] have shown that τ_1 and τ_2 are consistent with each other if and only if they share a concretization in common. Based on Proposition 4.4, we deduce a similar result for our system:

Proposition 4.5. *For every gradual types $\tau_1, \tau_2 \in \mathsf{GTypes}$,*

$$\tau_1 \sim \tau_2 \overset{\text{def}}{\iff} \exists \tau \in \mathsf{GTypes},\ \tau_1 \preccurlyeq \tau \text{ and } \tau_2 \preccurlyeq \tau$$

Proof. Straightforward by induction on the pair (τ_1, τ_2), following the inductive definition of consistency. □

This result has already been remarked by Siek and Vachharajani [67] using their precision relation, which is the inverse of our precision.

The second commonly-used relation in gradual type systems is called *consistent subtyping*, and is often noted $\widetilde{\leq}$. In a system where static types are equipped with a subtyping relation, consistent subtyping is a relation formalizing whether a gradual typing *may be* used in place of another (provided the necessary dynamic checks are inserted). For example, in a context expecting an expression of type ?, any expression of any type can be used. Conversely, in a context expecting an expression of a type t, an expression of type ? can always be used, provided we insert a dynamic check to ensure that the result of this expression is indeed of type t. This shows that consistent subtyping is a non-transitive relation whose transitive closure is the full relation on gradual types: ? is both a consistent subtype and a consistent supertype of every type. For this reason, it cannot be used in a subsumption rule like the precision relation, and it is often embedded directly in other typing rules. For example, the typing rule for applications may become:

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau}\ (\tau_2 \mathrel{\widetilde{\leq}} \tau_1)$$

Garcia et al. [31] define consistent subtyping as the existence of two concretizations that verify the subtyping relation on static types. In our system, provided static types are equipped with a subtyping relation $\leq_S$, we can define consistent subtyping as follows:

Definition 4.6. *The* consistent subtyping *relation on gradual types, noted $\widetilde{\leq}$, is defined as:*

$$\tau_1 \mathrel{\widetilde{\leq}} \tau_2 \overset{\text{def}}{\iff} \exists t_1, t_2 \in \mathsf{STypes}.\ \tau_1 \preccurlyeq t_1,\ \tau_2 \preccurlyeq t_2,\ t_1 \leq_S t_2$$

4.1.4. Relationship with existing type systems

We said that our type system is *declarative*. This is because all auxiliary relations (here precision) are handled by structural rules (here $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Mater}}]$) added to an existing set of logical and identity

$$
\begin{array}{rcll}
\text{STypes} \ni t & ::= & b \mid t \to t & \text{static simple types} \\
\text{GTypes} \ni \tau & ::= & ? \mid b \mid \tau \to \tau & \text{gradual simple types} \\
\text{Terms} \ni e & ::= & x \mid c \mid \lambda x{:}\tau.\, e \mid e\, e & \text{expressions}
\end{array}
$$

$$
[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Cst}}]\ \frac{}{\Gamma \vdash_{\mathsf{M}} c : b_c} \qquad [\mathrm{T}^{\mathsf{HM}}_{\mathsf{Var}}]\ \frac{}{\Gamma \vdash_{\mathsf{M}} x : t}\ \Gamma(x) = t
$$

$$
[\mathrm{T}^{\mathsf{HM}}_{\mathsf{App}}]\ \frac{\Gamma \vdash_{\mathsf{M}} e_1 : \tau' \to \tau \quad \Gamma \vdash_{\mathsf{M}} e_2 : \tau'}{\Gamma \vdash_{\mathsf{M}} e_1\, e_2 : \tau} \qquad [\mathrm{T}^{\mathsf{HM}}_{\mathsf{AAbstr}}]\ \frac{\Gamma, x : \tau' \vdash_{\mathsf{M}} e : \tau}{\Gamma \vdash_{\mathsf{M}} \lambda x{:}\tau'.\, e : \tau' \to \tau}
$$

$$
[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Mater}}]\ \frac{\Gamma \vdash_{\mathsf{M}} e_1 : \tau'}{\Gamma \vdash_{\mathsf{M}} e_2 : \tau}\ \tau' \preccurlyeq \tau
$$

FIGURE 4.2. Monomorphic restriction of the implicative fragment of our system

rules.[2] In a declarative system, every term may have different types and derivations; removing the structural rules corresponds to finding an algorithmic system that for every well-typed term chooses one particular derivation and, thus, one type of the declarative system. This is usually obtained by moving the checks of the auxiliary relations into the elimination rules: this yields a system that is easier to implement but less understandable. And this is exactly what most gradual type systems do, via the introduction of consistency and consistent subtyping. It is possible to show that the set of typable terms of our declarative system is the same as the set of typable terms of the existing gradual type systems that use consistency.

In particular, the relation between our system and the gradual type system of Siek and Taha [65] can be stated formally. Let $\vdash_{\mathsf{ST}}$ denote the typing judgments of Siek and Taha [65] and let $\vdash_{\mathsf{M}}$ denote the monomorphic restriction of the implicative fragment of our system, that is, our gradual types without type variables and the typing rules of the simply-typed λ-calculus plus materialization: see Figure 4.2 for the complete definition. Then we have the following result:

Proposition 4.7. *If* $\Gamma \vdash_{\mathsf{ST}} e : \tau$ *then* $\Gamma \vdash_{\mathsf{M}} e : \tau$. *Conversely, if* $\Gamma \vdash_{\mathsf{M}} e : \tau$, *then there exists a type* τ' *such that* $\Gamma \vdash_{\mathsf{ST}} e : \tau'$ *and* $\tau' \preccurlyeq \tau$.

Proof. Both implications can be shown by a straightforward induction on the corresponding typing derivation. The most enlightening case lies in the proof that $\Gamma \vdash_{\mathsf{ST}} e : \tau$ implies $\Gamma \vdash_{\mathsf{M}} e : \tau$, for the rule [GAPP2] of Siek and Taha [65]:

$$
[\text{GAPP2}]\ \frac{\Gamma \vdash_{\mathsf{ST}} e_1 : \tau' \to \tau \qquad \Gamma \vdash_{\mathsf{ST}} e_2 : \tau_2}{\Gamma \vdash_{\mathsf{ST}} e_1\, e_2 : \tau}\ \tau_2 \sim \tau'
$$

This rule is derivable in our system, because, by Proposition 4.5, if $\tau_2 \sim \tau'$ then there exists some type τ_3 such that $\tau_2 \preccurlyeq \tau_3$ and $\tau' \preccurlyeq \tau_3$. Then, using $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Mater}}]$ twice, we have $\Gamma \vdash_{\mathsf{M}} e_1 : \tau_3 \to \tau$ and $\Gamma \vdash_{\mathsf{M}} e_2 : \tau_3$. Finally, $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{App}}]$ proves that $\Gamma \vdash_{\mathsf{M}} e_1\, e_2 : \tau$. □

[2] In logic, logical rules refer to a particular connective (here, a type constructor, that is, either $\to$, or $\times$, or b), while identity rules (e.g., axioms and cuts) and structural rules (e.g., weakening and contraction) do not.

$$
\begin{array}{rrll}
\text{STypes} \ni t & ::= & \alpha \mid b \mid t \to t & \text{static simple types} \\
\text{GTypes} \ni \tau & ::= & ? \mid \alpha \mid b \mid \tau \to \tau & \text{gradual simple types} \\
\text{Terms} \ni e & ::= & x \mid c \mid \lambda x.\, e \mid \lambda x{:}\tau.\, e \mid e\, e & \text{expressions}
\end{array}
$$

$$
[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Cst}}]\ \frac{}{\Gamma \vdash_{\mathsf{P}} c : b_c} \qquad [\mathrm{T}^{\mathsf{HM}}_{\mathsf{Var}}]\ \frac{}{\Gamma \vdash_{\mathsf{P}} x : \tau\,[\vec{t}/\vec{\alpha}]}\ \Gamma(x) = \forall\vec{\alpha}.\tau
$$

$$
[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Abstr}}]\ \frac{\Gamma, x : t \vdash_{\mathsf{P}} e : \tau}{\Gamma \vdash_{\mathsf{P}} \lambda x.\, e : t \to \tau} \qquad [\mathrm{T}^{\mathsf{HM}}_{\mathsf{AAbstr}}]\ \frac{\Gamma, x : \tau' \vdash_{\mathsf{P}} e : \tau}{\Gamma \vdash_{\mathsf{P}} \lambda x{:}\tau'.\, e : \tau' \to \tau}
$$

$$
[\mathrm{T}^{\mathsf{HM}}_{\mathsf{App}}]\ \frac{\Gamma \vdash_{\mathsf{P}} e_1 : \tau' \to \tau \quad \Gamma \vdash_{\mathsf{P}} e_2 : \tau'}{\Gamma \vdash_{\mathsf{P}} e_1\, e_2 : \tau} \qquad [\mathrm{T}^{\mathsf{HM}}_{\mathsf{Mater}}]\ \frac{\Gamma \vdash_{\mathsf{P}} e_1 : \tau'}{\Gamma \vdash_{\mathsf{P}} e_2 : \tau}\ \tau' \preccurlyeq \tau
$$

FIGURE 4.3. Polymorphic restriction of the implicative fragment of our system

The (polymorphic) implicative fragment of our system (i.e., our system without products), denoted by $\vdash_{\mathsf{P}}$ and presented in Figure 4.3, is yet another well-known gradual type system, because it coincides with the ITGL type system of Garcia and Cimini [30], denoted by $\vdash_{\mathsf{IT}}$, as stated by the following result:

Proposition 4.8. *If* $\Gamma \vdash_{\mathsf{IT}} e : \tau$ *then* $\Gamma \vdash_{\mathsf{P}} e : \tau$. *Conversely, if* $\Gamma \vdash_{\mathsf{P}} e : \tau$, *then there exists a type* τ' *such that* $\Gamma \vdash_{\mathsf{IT}} e : \tau'$ *and* $\tau' \preccurlyeq \tau$.

Proof. The proof is mostly the same as the proof of Proposition 4.7, the main difference being the presence of unannotated λ-abstractions. However, our typing rule $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Abstr}}]$ is identical to the rule [Uλ] of Garcia and Cimini [30]. □

In other words, the relationship between our new declarative approach (i.e., with the $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Mater}}]$ rule) and the standard ones that use consistency (e.g., Siek and Taha [65] and Garcia and Cimini [30]) is analogous to the usual relationship between a declarative type system with subtyping (i.e., with a subsumption rule) and an algorithmic type system.

4.1.5. Static gradual guarantee

The static gradual guarantee is a property introduced by Siek et al. [69] to formalize the soundness of gradual type systems. In essence, it states that making the type annotations of a program less precise preserves its type.

The presence of $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Mater}}]$ in the type system of our source language yields the static gradual guarantee property without much effort. To show this, we lift the precision relation to terms by relating type annotations via precision. This is a standard construction whose rules are given in Figure 4.4.

Since types only appear in annotated λ-abstractions, most rules are straightforward and only check that sub-expressions are in a precision relation. The rule for annotated λ-abstractions simply adds the check that the explicit type annotations are also in a precision relation.

The static gradual guarantee is stated as follows:

$$\frac{}{x \preccurlyeq x} \qquad \frac{}{c \preccurlyeq c} \qquad \frac{e \preccurlyeq e'}{\lambda x.\, e \preccurlyeq \lambda x.\, e'} \qquad \frac{e \preccurlyeq e' \qquad \tau \preccurlyeq \tau'}{\lambda x{:}\tau.\, e \preccurlyeq \lambda x.\, \tau' e'} \qquad \frac{e_1 \preccurlyeq e_1' \quad e_2 \preccurlyeq e_2'}{e_1\, e_2 \preccurlyeq e_1'\, e_2'}$$

$$\frac{e_1 \preccurlyeq e_1' \quad e_2 \preccurlyeq e_2'}{(e_1, e_2) \preccurlyeq (e_1', e_2')} \qquad \frac{e \preccurlyeq e'}{\pi_i\, e \preccurlyeq \pi_i\, e'} \qquad \frac{e_1 \preccurlyeq e_1' \quad e_2 \preccurlyeq e_2'}{\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2 \preccurlyeq \mathsf{let}\ \vec{\alpha}\ x = e_1'\ \mathsf{in}\ e_2'}$$

FIGURE 4.4. Precision of terms in $\mathsf{Terms}^{\mathsf{HM}}$

Theorem 4.9 (Static gradual guarantee). *If $\emptyset \vdash e : \tau$ and $e' \preccurlyeq e$, then $\emptyset \vdash e' : \tau$.*

Due to the presence of type schemes in environments and typing rules, we cannot immediately prove this theorem. We first need to extend the notion of precision to type schemes and environments, and then show a weakening property. Given two type schemes $S_1 = \forall \vec{\alpha}_1.\tau_1$ and $S_2 = \forall \vec{\alpha}_2.\tau_2$, we write $S_1 \preccurlyeq S_2$ if for every instance $\tau_2' = \tau_2\,[\vec{t}_2/\vec{\alpha}_2]$ of S_2 there exists an instance $\tau_1' = \tau_1\,[\vec{t}_1/\vec{\alpha}_1]$ of S_1 such that $\tau_1' \preccurlyeq \tau_2'$. It is straightforward to show that, if $\vec{\alpha}_1 = \vec{\alpha}_2 = \emptyset$, then this is equivalent to precision on gradual types. We then extend this definition to type environments, and we write $\Gamma_1 \preccurlyeq \Gamma_2$, when for every $x \in \mathsf{dom}(\Gamma_1)$, $\Gamma_1(x) \preccurlyeq \Gamma_2(x)$.

We first state three results related to precision and free type variables.

Lemma 4.10. *For every type scheme $S = \forall \vec{\alpha}.\tau$. The following results hold:*

- *for every instance $\tau\,[\vec{t}/\vec{\alpha}]$ of S, $\mathsf{vars}(S) \subseteq \mathsf{vars}(\tau\,[\vec{t}/\vec{\alpha}])$;*
- *there exists an instance $\tau\,[\vec{t}/\vec{\alpha}]$ of S such that $\mathsf{vars}(S) = \mathsf{vars}(\tau\,[\vec{t}/\vec{\alpha}])$.*

Proof. For the first point, notice that $\mathsf{vars}(S) = \mathsf{vars}(\tau) \setminus \vec{\alpha}$ and that $\mathsf{vars}(\tau\,[\vec{t}/\vec{\alpha}]) = (\mathsf{vars}(\tau) \setminus \vec{\alpha}) \cup \mathsf{vars}(\vec{t})$.
For the second point, the same remark shows that any vector $\vec{t}$ of closed types (i.e., such that $\mathsf{vars}(\vec{t}) = \emptyset$) suffices. □

Lemma 4.11. *For every types $\tau_1, \tau_2 \in \mathsf{GTypes}$, if $\tau_1 \preccurlyeq \tau_2$ then $\mathsf{vars}(\tau_1) \subseteq \mathsf{vars}(\tau_2)$.*

Proof. By hypothesis, since $\tau_1 \preccurlyeq \tau_2$, there exists $T_1 \in \mathsf{TFrames}$ and $\theta : \mathscr{V}^X \to \mathsf{GTypes}$ such that ${T_1}^\dagger = \tau_1$ and $T_1\theta = \tau_2$. Since θ only acts on frame variables and $\mathsf{vars}(\tau_1) \subseteq \mathscr{V}^\alpha$, it must hold that $\mathsf{vars}(\tau_1) \subseteq \mathsf{vars}(\tau_2)$. □

Lemma 4.12. *The following results hold:*

- *for every type schemes S_1, S_2, if $S_1 \preccurlyeq S_2$ then $\mathsf{vars}(S_1) \subseteq \mathsf{vars}(S_2)$;*
- *for every type environments Γ_1, Γ_2, if $\Gamma_1 \preccurlyeq \Gamma_2$ then $\mathsf{vars}(\Gamma_1) \subseteq \mathsf{vars}(\Gamma_2)$.*

Proof. Let $S_1 = \forall \vec{\alpha}_1.\tau_1$ and $S_2 = \forall \vec{\alpha}_2.\tau_2$ two type schemes such that $S_1 \preccurlyeq S_2$. By Lemma 4.10, there exists an instance $\tau_2' = \tau_2\,[\vec{t}_2/\vec{\alpha}_2]$ of S_2 such that $\mathsf{vars}(\tau_2') = \mathsf{vars}(S_2)$. By definition of the precision of type schemes, there exists an instance $\tau_1' = \tau_1\,[\vec{t}_1/\vec{\alpha}_1]$ of S_1 such that $\tau_1' \preccurlyeq \tau_2'$. By Lemma 4.11, $\mathsf{vars}(\tau_1') \subseteq \mathsf{vars}(\tau_2')$, and by Lemma 4.10, $\mathsf{vars}(S_1) \subseteq \mathsf{vars}(\tau_1')$. Therefore, we deduce that $\mathsf{vars}(S_1) \subseteq \mathsf{vars}(S_2)$.

For type environments, the result is an immediate corollary of the first point. □

We now state the weakening result that will allow us to prove the static gradual guarantee.

Lemma 4.13. *For every term* $e \in \mathsf{Terms}^{\mathsf{HM}}$ *and every type environments* Γ_1, Γ_2*, if* $\Gamma_1 \preccurlyeq \Gamma_2$ *and* $\Gamma_2 \vdash e : \tau$ *then* $\Gamma_1 \vdash e : \tau$*.*

Proof. By induction on the derivation on $\Gamma_2 \vdash e : \tau$ and case analysis on the last rule applied.

- $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Cst}}]$. Immediate since the rule does not depend on the environment.
- $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Var}}]$. By inversion of the typing rules, $\tau = \tau_2\,[\vec{t}_2/\vec{\alpha}_2]$ where $\Gamma_2(x) = \forall\vec{\alpha}_2.\tau_2$. By definition of $\preccurlyeq$ on environments, $\Gamma_1(x) \preccurlyeq \Gamma_2(x)$. Thus, by noting $\Gamma_1(x) = \forall\vec{\alpha}_1.\tau_1$, we can find an instance $\tau' = \tau_1\,[\vec{t}_1/\vec{\alpha}_1]$ of $\Gamma_1(x)$ such that $\tau' \preccurlyeq \tau$. By $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Var}}]$, we have $\Gamma_1 \vdash x : \tau'$, and by $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Mater}}]$, we deduce $\Gamma_1 \vdash x : \tau$.
- $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Proj}}]$, $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Pair}}]$, $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{App}}]$, $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Mater}}]$. All these cases are immediately proven by application of the induction hypothesis.
- $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Abstr}}]$, $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{AAbstr}}]$. Also proven immediately by induction hypothesis, by noting that for every type $\tau \in \mathsf{GTypes}$, since $\tau \preccurlyeq \tau$, then it holds that $(\Gamma_1, x : \tau) \preccurlyeq (\Gamma_2, x : \tau)$.
- $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Let}}]$. . By inversion, we have $\Gamma_2 \vdash \mathsf{let}\,\vec{\alpha}\,x = e_1\,\mathsf{in}\,e_2 : \tau$ where:

  $$\Gamma_2 \vdash e_1 : \tau_1 \qquad \Gamma_2, x : \forall\vec{\alpha}, \vec{\beta}.\tau_1 \vdash e_2 : \tau \qquad \vec{\alpha}, \vec{\beta} \sharp \Gamma_2 \text{ and } \vec{\beta} \sharp e_1$$

  By induction hypothesis, $\Gamma_1 \vdash e_1 : \tau_1$. By reflexivity, $\Gamma_2, x : \forall\vec{\alpha}, \vec{\beta}.\tau_1 \preccurlyeq \Gamma_1, x : \forall\vec{\alpha}, \vec{\beta}.\tau_1$ and by induction hypothesis, $\Gamma_1, x : \forall\vec{\alpha}, \vec{\beta}.\tau_1 \vdash e_2 : \tau$. By Lemma 4.12, $\mathrm{vars}(\Gamma_1) \subseteq \mathrm{vars}(\Gamma_2)$, which then yields that $\vec{\alpha}, \vec{\beta} \sharp \Gamma_1$. Thus, we have verified all the premises of Rule $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Let}}]$ which proves that $\Gamma_1 \vdash \mathsf{let}\,\vec{\alpha}\,x = e_1\,\mathsf{in}\,e_2 : \tau$.

□

And finally, using this result, we can prove the static gradual guarantee as stated above.

Proof of Theorem 4.9. We prove the stronger claim that, for every Γ, if $\Gamma \vdash e : \tau$ and $e' \preccurlyeq e$ then $\Gamma \vdash e' : \tau$. The proof is done by induction on the derivation $\Gamma \vdash e : \tau$ and by case analysis on the last rule applied. All cases are straightforward by application of the induction hypothesis, except for Rule $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{AAbstr}}]$.
In that case, by inversion, we have $e = \lambda x{:}\tau_1.\,e_1$ and $\tau = \tau_1 \to \tau_2$ where $\Gamma, x : \tau_1 \vdash e_1 : \tau_2$. Since $e' \preccurlyeq e$, we have $e' = \lambda x{:}\tau_1'.\,e_1'$ with $\tau_1' \preccurlyeq \tau_1$ and $e_1' \preccurlyeq e_1$. By induction hypothesis, $\Gamma, x : \tau_1 \vdash e_1' : \tau_2$. By Lemma 4.13, we deduce that $\Gamma, x : \tau_1' \vdash e_1' : \tau_2$. By $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Aabstr}}]$ we deduce that $\Gamma \vdash e' : \tau_1' \to \tau_2$, and the result follows by application of $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Mater}}]$. □

4.2. Cast language

As customary with gradual typing, the semantics of the gradually-typed language is given by translating its well-typed expressions into a cast language (also called target language), which we define next.

4.2.1. Syntax

The syntax of the cast language is defined as follows:

$$\mathsf{Terms}^{\langle \mathsf{HM} \rangle} \ni E ::= x \mid c \mid \lambda^{\tau \to \tau} x.\, E \mid E\, E \mid (E, E) \mid \pi_i\, E \mid \mathsf{let}\, x = E\, \mathsf{in}\, E \mid \Lambda \vec{\alpha}.\, E \mid E[\vec{t}] \mid E\langle \tau \Rightarrow_p \tau \rangle$$

This is an explicitly-typed λ-calculus similar to the source language with a few differences and the addition of explicit casts.

There is now just one kind of λ-abstraction, $\lambda^{\tau_1 \to \tau_2} x.\, E$, which is annotated with its complete arrow type. While this is not strictly necessary in this calculus, this will become important when adding set-theoretic types in the next chapter, since the semantics of cast applications will need to access the type of their argument.

Additionally, let-expressions no longer bind type variables; instead, there are explicit type abstractions $\Lambda \vec{\alpha}.\, E$ and applications $E[\vec{t}]$. For example, the source language expression $\mathsf{let}\, \alpha\, z = \lambda x{:}\alpha.\, \lambda y.\, x\, \mathsf{in}\, z\, 42$, of type $\beta \to \mathsf{Int}$, is translated into the cast calculus as $\mathsf{let}\, z = \Lambda \alpha \beta.\, \lambda^{\alpha \to \beta \to \alpha} x.\, \lambda^{\beta \to \alpha} y.\, x\, \mathsf{in}\, z[\mathsf{Int}, \beta]\, 42$. Despite the presence of type abstractions, the cast calculus does not support first-class polymorphism; the syntax of types remains unchanged from Section 4.1 and does not include universally quantified types.

Finally, the important additions to the calculus are explicit casts of the form $E\langle \tau \Rightarrow_p \tau' \rangle$ where, as usual in the gradual typing literature, p ranges over a set of polarized blame labels. Such an expression dynamically checks whether E, of static type τ, produces a value of type τ'; if the cast fails, then the label p is used to blame the cast. These casts are inserted during compilation to perform runtime checks in dynamically-typed code: for instance, the function $\lambda x{:}?.\, x + 1$ will be compiled into $\lambda^{? \to \mathsf{Int}} x.\, x\langle ? \Rightarrow_\ell \mathsf{Int} \rangle + 1$, which checks at runtime whether the function parameter is bound to an integer value (and if not blames the label ℓ). Blame labels are pointers that indicate precisely where a cast was originally inserted, and are inserted alongside casts during compilation. When a cast fails, this allows the program to report precise error messages.

We suppose given a set $\mathscr{L}$ of blame labels, ranged over by ℓ. As customary, blame labels in casts are polarized, and we follow the standard convention of using ℓ to range over positive labels and $\overline{\ell}$ for negative ones.

The polarity of a label indicates whether the failure at point ℓ is due to the context (blame $\overline{\ell}$) or to the expression in that context (blame ℓ). We write p to denote a blame label independently of its polarity, and use the involutory operation $\overline{p}$ to reverse the polarity of a label.

4.2.2. Type system

The typing rules for the cast language are presented in Figure 4.5. Most of the typing rules are identical to the rules presented in the previous section for the source language. Type environments still associate variables to type schemes of the form $\forall \vec{\alpha}.\tau$ (rule $[\mathrm{T}^{\langle \mathsf{HM} \rangle}_{\mathsf{Var}}]$), and we use the standard rules for the introduction $[\mathrm{T}^{\langle \mathsf{HM} \rangle}_{\mathsf{TAbstr}}]$ and elimination $[\mathrm{T}^{\langle \mathsf{HM} \rangle}_{\mathsf{TApp}}]$ of type abstractions.

The main addition is our typing rules for casts, which are more precise than the current literature, since they capture invariants that are typically captured by a separate safe-for relation that is used to establish the Blame Theorem [75]. A cast $E\langle \tau' \Rightarrow_p \tau \rangle$ is well-typed if the cast expression has type τ' and if the cast goes from this type to either a more precise (positive label) or a less precise (negative label) gradual type τ. See rules $[\mathrm{T}^{\langle \mathsf{HM} \rangle}_{\mathsf{Cast+}}]$ and $[\mathrm{T}^{\langle \mathsf{HM} \rangle}_{\mathsf{Cast-}}]$, respectively.

The central idea here is that we correlate the polarity of a label with the direction of a cast: an upcast will have a negative label while a downcast will have a positive one. In the existing gradual typing literature, proving the blame safety theorem usually involves two subtyping rela-

$$[\mathrm{T}^{\langle \mathrm{HM} \rangle}_{\mathrm{Cst}}]\ \frac{}{\Gamma \vdash c : b_c} \qquad [\mathrm{T}^{\langle \mathrm{HM} \rangle}_{\mathrm{Var}}]\ \frac{}{\Gamma \vdash x : \tau\,[\vec{t}/\vec{\alpha}]}\ \Gamma(x) = \forall \vec{\alpha}.\tau$$

$$[\mathrm{T}^{\langle \mathrm{HM} \rangle}_{\mathrm{Proj}}]\ \frac{\Gamma \vdash E : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\, E : \tau_i} \qquad [\mathrm{T}^{\langle \mathrm{HM} \rangle}_{\mathrm{Pair}}]\ \frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash (E_1, E_2) : \tau_1 \times \tau_2} \qquad [\mathrm{T}^{\langle \mathrm{HM} \rangle}_{\mathrm{App}}]\ \frac{\Gamma \vdash E_1 : \tau' \to \tau \quad \Gamma \vdash E_2 : \tau'}{\Gamma \vdash E_1\, E_2 : \tau}$$

$$[\mathrm{T}^{\langle \mathrm{HM} \rangle}_{\mathrm{Abstr}}]\ \frac{\Gamma, x : \tau \vdash E : \tau'}{\Gamma \vdash \lambda^{\tau \to \tau'} x.\, E : \tau \to \tau'} \qquad [\mathrm{T}^{\langle \mathrm{HM} \rangle}_{\mathrm{Let}}]\ \frac{\Gamma \vdash E_1 : \forall \vec{\alpha}.\tau_1 \quad \Gamma, x : \forall \vec{\alpha}.\tau_1 \vdash E_2 : \tau}{\Gamma \vdash \mathsf{let}\, x = E_1\, \mathsf{in}\, E_2 : \tau}\ \vec{\alpha} \sharp \Gamma$$

$$[\mathrm{T}^{\langle \mathrm{HM} \rangle}_{\mathrm{TAbstr}}]\ \frac{\Gamma \vdash E : \tau}{\Gamma \vdash \Lambda \vec{\alpha}.\, E : \forall \vec{\alpha}.\tau}\ \vec{\alpha} \sharp \Gamma \qquad [\mathrm{T}^{\langle \mathrm{HM} \rangle}_{\mathrm{TApp}}]\ \frac{\Gamma \vdash E : \forall \vec{\alpha}.\tau}{\Gamma \vdash E[\vec{t}] : \tau\,[\vec{t}/\vec{\alpha}]}$$

$$[\mathrm{T}^{\langle \mathrm{HM} \rangle}_{\mathrm{Cast+}}]\ \frac{\Gamma \vdash E : \tau'}{\Gamma \vdash E\langle \tau' \Rightarrow_l \tau \rangle : \tau}\ \tau' \preccurlyeq \tau \qquad [\mathrm{T}^{\langle \mathrm{HM} \rangle}_{\mathrm{Cast-}}]\ \frac{\Gamma \vdash E : \tau'}{\Gamma \vdash E\langle \tau' \Rightarrow_{\overline{l}} \tau \rangle : \tau}\ \tau \preccurlyeq \tau'$$

FIGURE 4.5. Typing rules for the cast language

tions, called *positive subtyping* (written $\leq^+$) and *negative subtyping* (written $\leq^-$), characterizing respectively casts that cannot yield positive blame and casts that cannot yield negative blame. The factoring theorem for naive subtyping from Wadler and Findler [75] shows that our precision relation implies negative subtyping, and implies the converse of positive subtyping. In other words, if $\tau' \preccurlyeq \tau$ then $\tau' <:^- \tau$ and $\tau <:^+ \tau'$. This ensures that a cast that satisfies rule $[\mathrm{T}^{\langle \mathrm{HM} \rangle}_{\mathrm{Cast+}}]$ is safe for $\overline{\ell}$, and that a cast that satisfies rule $[\mathrm{T}^{\langle \mathrm{HM} \rangle}_{\mathrm{Cast-}}]$ is also safe for $\overline{\ell}$. In other words, the type system ensures that a well-typed expression can never produce negative blame: this is the blame safety property of our calculus, which will be stated in Corollary 4.19.

This contrasts with existing approaches, where the direction of a cast[3] and the polarity of its blame label are unrelated, and in which defining *safe casts* thus require the definition of additional relations. In our system, safe casts are simply casts with negative labels, which greatly simplifies the statement and proof of blame safety. As we will see in Subsection 4.2.4, this is possible thanks to the fact that every well-typed term of the source language can be compiled into a well-typed term of the cast language using only downcasts.

4.2.3. Semantics

The main difficulty when defining the semantics of a cast calculus is to determine when a cast should fail. To solve this problem, Wadler and Findler [75] introduce the notion of *ground types* to compare types in casts, with the idea that *incompatibility between ground types is the source of all blame.* A ground type contains very little information: it only serves to know whether the underlying value is a function (its ground type being $? \to ?$), a pair ($? \times ?$), or a constant (in which case its ground type is simply its base type b). This allows us to perform purely syntactic checks when deciding whether a cast should fail or not: casting an expression whose ground type is $? \to ?$ (that is, a function) to $? \times ?$ fails because a function cannot be cast to a pair.

Precision allows us to define the ground type of a type τ as being the type that materializes in

[3] In most systems, the direction of a cast is not even properly defined, since compilation can insert casts such as $\langle ? \to \mathsf{Int} \Rightarrow_p \mathsf{Int} \to ? \rangle$, in which no type is more precise than the other.

τ while containing as little information as possible about τ, without being equal to ?:

Proposition 4.14 (Ground type existence)**.** *For every type $\tau \in \mathsf{GTypes} \setminus \{?\}$, there exists a unique type $\tau' \in \mathsf{GTypes}$ satisfying the following conditions:*

- $\tau' \preccurlyeq \tau$ *and* $\tau' \neq ?$
- $\forall \tau'' \in \mathsf{GTypes}, \tau'' \preccurlyeq \tau' \implies \tau'' = ?$ *or* $\tau'' = \tau'$

Proof. By induction on τ.

- $\underline{b}$. The only type τ' such that $\tau' \preccurlyeq b$ and $\tau' \neq ?$ is $\tau' = b$.
- $\underline{\alpha}$. Similar to b.
- $\underline{\tau_1 \times \tau_2}$. Let $\tau' = ? \times ?$. By definition of $\preccurlyeq$, we have $\tau' \preccurlyeq \tau_1 \times \tau_2$, and it is clear that $\tau' \neq ?$. Now let $\tau'' \in \mathsf{GTypes}$ such that $\tau'' \preccurlyeq \tau'$ and $\tau'' \neq ?$. By inversion of the definition of $\preccurlyeq$, since $\tau'' \preccurlyeq ? \times ?$, we have $\tau'' = \tau_1'' \times \tau_2''$ where $\tau_1'' \preccurlyeq ?$ and $\tau_2'' \preccurlyeq ?$. By definition of $\preccurlyeq$, this imposes that $\tau_1'' = \tau_2'' = ?$, which proves that $\tau'' = \tau'$.
- $\underline{\tau_1 \to \tau_2}$. Similar to the previous case, defining τ' as $? \to ?$.

□

Based on this proposition, we define the *ground type operator* that associates to every type τ (different from ?) its ground type. This also gives us a definition of ground types as being the types that are left unchanged by an application of this operator.

Definition 4.15 (Ground types)**.** *For every type $\tau \in \mathsf{GTypes} \setminus \{?\}$, we define the ground type of τ, noted $\mathsf{gnd}(\tau)$, as the unique gradual type that verifies the conditions of Proposition 4.14. Types τ such that $\mathsf{gnd}(\tau) = \tau$ are called* ground types *and are ranged over by ρ.*

Following the proof of Proposition 4.14, we obtain that ground types follow the usual inductive definition from Wadler and Findler [75]:

$$\rho ::= b \mid \alpha \mid ? \times ? \mid ? \to ?$$

and that the ground type operator $\mathsf{gnd}(.)$ obeys the usual equations:

$$\begin{array}{rclcrcl} \mathsf{gnd}(b) & = & b & & \mathsf{gnd}(\alpha) & = & \alpha \\ \mathsf{gnd}(\tau_1 \times \tau_2) & = & ? \times ? & & \mathsf{gnd}(\tau_1 \to \tau_2) & = & ? \to ? \end{array}$$

Now that ground types are defined, we can present the operational semantics of the cast calculus. The cast calculus features a strict call-by-value reduction semantics defined in a small-step style by the reduction rules presented in Figure 4.6. The semantics is defined in terms of values (ranged over by V) and evaluation contexts (ranged over by $\mathscr{E}$).

Values of the cast language are defined by the following grammar:

$$\mathsf{Values}^{\langle \mathsf{HM} \rangle} \ni V ::= c \mid \lambda^{\tau \to \tau} x.E \mid (V, V) \mid V\langle \tau \to \tau \Rightarrow_p \tau \to \tau \rangle \mid V\langle \tau \times \tau \Rightarrow_p \tau \times \tau \rangle \mid V\langle \rho \Rightarrow_p ? \rangle$$

where, additionally, no casts are identity casts. This definition of values follows the definition given by Siek et al. [68], where there are three value forms with casts. Values cast from an arrow

Cast reductions.

$$
\begin{array}{rrcll}
[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{ExpandL}}] & V\langle \tau \Rightarrow_p ? \rangle & \rightsquigarrow & V\langle \tau \Rightarrow_p \mathrm{gnd}(\tau) \rangle \langle \mathrm{gnd}(\tau) \Rightarrow_p ? \rangle & \text{if } \tau \neq ? \text{ and } \tau \neq \mathrm{gnd}(\tau) \\
[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{ExpandR}}] & V\langle ? \Rightarrow_p \tau \rangle & \rightsquigarrow & V\langle ? \Rightarrow_p \mathrm{gnd}(\tau) \rangle \langle \mathrm{gnd}(\tau) \Rightarrow_p \tau \rangle & \text{if } \tau \neq ? \text{ and } \tau \neq \mathrm{gnd}(\tau) \\
[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{CastId}}] & V\langle \tau \Rightarrow_p \tau \rangle & \rightsquigarrow & V & \\
[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{Collapse}}] & V\langle \rho \Rightarrow_p ? \rangle \langle ? \Rightarrow_q \rho \rangle & \rightsquigarrow & V & \\
[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{Blame}}] & V\langle \rho \Rightarrow_p ? \rangle \langle ? \Rightarrow_q \rho' \rangle & \rightsquigarrow & \mathsf{blame}\ q & \text{if } \rho \neq \rho'
\end{array}
$$

Standard reductions.

$$
\begin{array}{rrcll}
[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{CApp}}] & V\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle\, V' & \rightsquigarrow & (V\,(V'\langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle)) \langle \tau_2 \Rightarrow_p \tau_2' \rangle & \\
[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{App}}] & (\lambda^{\tau_1 \to \tau_2} x.\, E)\, V & \rightsquigarrow & E\,[V/x] & \\
[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{CProj}}] & \pi_i\,(V\langle \tau_1 \times \tau_2 \Rightarrow_p \tau_1' \times \tau_2' \rangle) & \rightsquigarrow & (\pi_i\, V) \langle \tau_i \Rightarrow_p \tau_i' \rangle & \\
[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{Proj}}] & \pi_i\,(V_1, V_2) & \rightsquigarrow & V_i & \\
[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{TApp}}] & (\Lambda \vec{\alpha}.\, E)[\vec{t}] & \rightsquigarrow & E\,[\vec{t}/\vec{\alpha}] & \\
[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{Let}}] & \mathsf{let}\, x = V\, \mathsf{in}\, E & \rightsquigarrow & E\,[V/x] & \\
[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{Ctx}}] & \mathscr{E}\,[E] & \rightsquigarrow & \mathscr{E}\,[E'] & \text{if } E \rightsquigarrow E' \\
[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{CtxBlame}}] & \mathscr{E}\,[E] & \rightsquigarrow & \mathsf{blame}\ p & \text{if } E \rightsquigarrow \mathsf{blame}\ p
\end{array}
$$

FIGURE 4.6. Semantics of the cast calculus

type to another arrow type, of the form $V\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle$, cannot be reduced unless they are applied to an argument (which will allow the cast to be split, as presented in $[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{CApp}}]$). The same goes for values cast to a pair type, which can only be reduced when they are applied a projection. Finally, values of the form $V\langle \rho \Rightarrow_p ? \rangle$ are, in essence, values that are "boxed" and annotated with their ground type. They can only be reduced by being "unboxed", that is, cast to some usable type $\tau \neq ?$. The ground type of τ will then be checked against ρ, to decide whether the cast should fail or not.

Evaluation contexts implement a standard right-most outer-most weak reduction strategy:

$$\mathscr{E} ::= [\,] \mid E\,\mathscr{E} \mid \mathscr{E}\,V \mid \mathscr{E}[\vec{t}] \mid (E, \mathscr{E}) \mid (\mathscr{E}, V) \mid \pi_i\,\mathscr{E} \mid \mathsf{let}\, x = \mathscr{E}\, \mathsf{in}\, E \mid \mathscr{E}\langle \tau \Rightarrow_p \tau \rangle$$

The reduction rules of Figure 4.6 closely follow the presentation of Siek et al. [68]. They are divided into two groups: the reductions for the application of casts to a value and the reductions corresponding to the elimination of type constructors. For the former we use the technique by Wadler and Findler [75] which consists in checking whether a cast is performed between two types with the same toplevel constructor and failing when this is not the case. This amounts to introducing intermediate ground types whenever necessary, using the two rules $[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{ExpandL}}]$ and $[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{ExpandR}}]$, and then checking whether the succession of an upcast and a downcast involves the same ground type (rule $[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{Collapse}}]$) or not (rule $[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{Blame}}]$). As we explained before, in regards to an implementation, the rule $[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{ExpandL}}]$ corresponds to tagging a value with its type constructor (as done in Lisp implementations), while the rule $[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{Collapse}}]$ corresponds to untagging a value.

Most of the rules of the standard reductions group are taken from Siek et al. [68] too: we added the rules for type abstractions and applications, for projections, and for let bindings (all absent

in the cited work).

The soundness of this calculus is proven via progress and subject reduction. However, we do not give a direct proof of these properties: instead, they follow from the corresponding properties of the cast calculus presented later in Section 5.3, and the conservativity of the extension stated in Theorem 5.32.

As customary, progress states that every well-typed expression that is not a value can be reduced. Additionally, we prove that if an expression reduces to a blame, then this blame is necessarily negative (i.e., it blames a negative label). This will entail the property of *blame safety* as an immediate consequence.

Lemma 4.16 (Progress)**.** *For every term $E \in \mathsf{Terms}^{\langle \mathsf{HM} \rangle}$, if $\emptyset \vdash E : \forall \vec{\alpha}.\tau$ then one of the following holds:*

- *there exists $E' \in \mathsf{Terms}^{\langle \mathsf{HM} \rangle}$ such that $E \rightsquigarrow E'$;*
- *there exists $\ell \in \mathscr{L}$ such that $E \rightsquigarrow \mathsf{blame}\ \ell$;*
- *$E \in \mathsf{Values}^{\langle \mathsf{HM} \rangle}$.*

Subject reduction states that the type of an expression is preserved by reduction, and is formalized as follows:

Lemma 4.17 (Subject reduction)**.** *For every term $E, E' \in \mathsf{Terms}^{\langle \mathsf{HM} \rangle}$, if $\Gamma \vdash E : \forall \vec{\alpha}.\tau$ and $E \rightsquigarrow E'$ then $\Gamma \vdash E' : \forall \vec{\alpha}.\tau$.*

Finally, soundness is an immediate consequence of the above two lemmas, and states that every expression either reduces to a value, reduces to a positive blame, or diverges. The latter is necessary since, due to gradual types, diverging expressions can be well-typed. For example, if we write $\omega = \lambda^{? \to ?} x.\, x\langle ? \Rightarrow_{\ell_1} ? \to ? \rangle\, x$, then $\omega\langle ? \to ? \Rightarrow_{\ell_2} (? \to ?) \to ? \rangle \omega$ is well-typed and diverges.

Theorem 4.18 (Soundness)**.** *For every term $E \in \mathsf{Terms}^{\langle \mathsf{HM} \rangle}$, if $\emptyset \vdash E : \forall \vec{\alpha}.\tau$ then one of the following holds:*

- *there exists $V \in \mathsf{Values}^{\langle \mathsf{HM} \rangle}$ such that $E \rightsquigarrow^* V$;*
- *there exists $\ell \in \mathscr{L}$ such that $E \rightsquigarrow^* \mathsf{blame}\ \ell$;*
- *E diverges.*

The second important result for our calculus is *blame safety*, introduced by Wadler and Findler [75], which guarantees that the statically typed part of a program cannot be blamed. In our system, as we anticipated, the typing rules enforce the correspondence between the polarity of the label of a cast and its direction (with respect to precision). That is, a cast of the form $\langle \tau \Rightarrow_p \tau' \rangle$ where p is negative necessarily verifies $\tau' \preccurlyeq \tau$ and cannot be blamed since going to a less-precise type is always a safe operation. Since all this information is encoded in the typing rules, blame safety is an immediate corollary of Theorem 4.18, and can be stated without resorting to positive and negative subtyping as defined by Wadler and Findler [75].

Corollary 4.19 (Blame safety)**.** *For every term $E \in \mathsf{Terms}^{\langle \mathsf{HM} \rangle}$ and every blame label $\ell \in \mathscr{L}$, if $\emptyset \vdash E : \forall \vec{\alpha}.\tau$ then $E \not\rightsquigarrow^* \mathsf{blame}\ \overline{\ell}$.*

Non-trivial rules.

$$[\mathrm{C}^{\mathsf{HM}}_{\mathsf{Var}}]\ \frac{}{\Gamma \vdash x \rightsquigarrow x[\vec{t}] : \tau\,[\vec{t}/\vec{\alpha}]}\ \Gamma(x) = \forall\vec{\alpha}.\tau \qquad [\mathrm{C}^{\mathsf{HM}}_{\mathsf{Abstr}}]\ \frac{\Gamma, x : t \vdash e \rightsquigarrow E : \tau}{\Gamma \vdash \lambda x.\, e \rightsquigarrow \lambda^{t\to\tau} x.\, E : t \to \tau}$$

$$[\mathrm{C}^{\mathsf{HM}}_{\mathsf{Let}}]\ \frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau_1 \quad \Gamma, x : \forall\vec{\alpha}, \vec{\beta}.\tau_1 \vdash e_2 \rightsquigarrow E_2 : \tau}{\Gamma \vdash \mathsf{let}\, x\, \vec{\alpha} = e_1 \,\mathsf{in}\, e_2 \rightsquigarrow \mathsf{let}\, x = \Lambda\vec{\alpha}, \vec{\beta}.\, E_1 \,\mathsf{in}\, E_2 : \tau}\ \vec{\alpha}, \vec{\beta} \sharp \Gamma \text{ and } \vec{\beta} \sharp e_1$$

$$[\mathrm{C}^{\mathsf{HM}}_{\mathsf{Mater}}]\ \frac{\Gamma \vdash e \rightsquigarrow E : \tau'}{\Gamma \vdash e \rightsquigarrow E\langle \tau' \Rightarrow_\ell \tau\rangle}\ \tau' \preccurlyeq \tau$$

Composition and identity rules.

$$[\mathrm{C}^{\mathsf{HM}}_{\mathsf{Cst}}]\ \frac{}{\Gamma \vdash c \rightsquigarrow c : b_c} \qquad [\mathrm{C}^{\mathsf{HM}}_{\mathsf{Proj}}]\ \frac{\Gamma \vdash e \rightsquigarrow E : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\, e \rightsquigarrow \pi_i\, E : \tau_i} \qquad [\mathrm{C}^{\mathsf{HM}}_{\mathsf{Pair}}]\ \frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau_1 \quad \Gamma \vdash e_2 \rightsquigarrow E_2 : \tau_2}{\Gamma \vdash (e_1, e_2) \rightsquigarrow (E_1, E_2) : \tau_1 \times \tau_2}$$

$$[\mathrm{C}^{\mathsf{HM}}_{\mathsf{App}}]\ \frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau' \to \tau \quad \Gamma \vdash e_2 \rightsquigarrow E_2 : \tau'}{\Gamma \vdash e_1\, e_2 \rightsquigarrow E_1\, E_2 : \tau} \qquad [\mathrm{C}^{\mathsf{HM}}_{\mathsf{AAbstr}}]\ \frac{\Gamma, x : \tau' \vdash e \rightsquigarrow E : \tau}{\Gamma \vdash \lambda x{:}\tau'.\, e \rightsquigarrow \lambda^{\tau'\to\tau} x.\, E : \tau' \to \tau}$$

FIGURE 4.7. Compilation rules to the cast calculus

4.2.4. Compilation

The final ingredient of the declarative definition of the system is to show how to compile a well-typed expression of the source language into an expression of the cast calculus and prove that compilation preserves types. This result, combined with the soundness of the cast language, implies the soundness of the gradually-typed language: a well-typed expression is compiled into an expression that can only either return a value of the same type, or return a cast error, or diverge.

Compilation is driven by the derivation of the type for the source language expression. Conceptually, compilation is straightforward: every time the derivation uses the $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Mater}}]$ rule on some subexpression for a relation $\tau_1 \preccurlyeq \tau_2$, a cast $\langle \tau_1 \Rightarrow_\ell \tau_2 \rangle$ must be added to that subexpression. This solves a common ambiguity of gradually-typed languages: to compile an application such as $\lambda x{:}?.\, x\, 5$ one can either downcast the function to $\mathsf{Int} \to \mathsf{Int}$, or upcast the argument to $?$. Our approach yields a simple solution: we always downcast the less-precisely typed expression. If the types of the argument and of the function are not comparable, then they are at least consistent with each other, and we have shown in the proof of Proposition 4.7 that it is always possible to downcast both types to a common more precise type.

This is crucial to our blame safety property: since compilation only inserts downcasts and positive blame labels, this ensures that compiled terms are well-typed for the rules given in Figure 4.5, and that the blame safety property presented in Corollary 4.19 holds for our source language.

Technically, to produce a declarative compilation system, we enrich the judgements of typing derivations with a compilation part: $\Gamma \vdash e \rightsquigarrow E : \tau$ means that the source language expression e of type τ compiles to the cast language expression E. These judgements are derived by the same rules as those given for the source language in Figure 4.1 to whose judgements we add the

compilation part. The compilation rules are presented in Figure 4.7, and are separated into two categories: rules that are non-trivial modifications of the rules of Figure 4.1, and rules that simply compile and compose subexpressions.

$[\mathrm{C}^{\mathsf{HM}}_{\mathsf{Var}}]$ compiles occurrences of polymorphic variables by instantiating them with the needed types. $[\mathrm{C}^{\mathsf{HM}}_{\mathsf{Abstr}}]$ explicitly annotates the function with the type deduced by inference. The compilation of a let-construct abstracts the type variables that are generalized. Finally, the core of compilation is given by the $[\mathrm{C}^{\mathsf{HM}}_{\mathsf{Mater}}]$ rule, which, as explained before, corresponds to the insertion of an explicit cast (with a positive fresh label ℓ). All the remaining rules are straightforward modifications of the rules in Figure 4.1 insofar as their conclusions simply compose the compiled expressions in the premisses.

Compilation is defined for all well-typed expressions and preserves well-typing:

Theorem 4.20 (Compilation soundness)**.** *For every term $e \in \mathsf{Terms}^{\mathsf{HM}}$ and every type environment Γ, if $\Gamma \vdash e : \tau$ then there exists a term $E \in \mathsf{Terms}^{\langle \mathsf{HM} \rangle}$ such that $\Gamma \vdash e \rightsquigarrow E : \tau$ and $\Gamma \vdash E : \tau$.*

4.3. Type inference

In this section we show how to decide whether a given term is well-typed or not: we define a type inference algorithm that is sound and complete with respect to the declarative systems we presented earlier. As anticipated, we will not delve too much into the details, and instead refer to Petrucciani [56] for more details about the systems and proofs.

The algorithm is mostly based on the work of Pottier and Rémy [58] and of Castagna et al. [16], adapted for gradual typing. Our algorithm differs from that of Garcia and Cimini [30] in that ours literally reduces the inference problem to unification. To infer the type of an expression, we generate constraints that specify the conditions that must hold for the expression to be well-typed; then, we solve these constraints via unification to obtain a solution (a type substitution).

Our presentation proceeds as follows. We first introduce *type constraints* (§4.3.1) and show how to solve sets of type constraints using standard unification (§4.3.2). Then we show how to generate constraints for a given expression (§4.3.3). To keep constraint generation separated from solving, generation uses more complex *structured constraints* (this is essentially due to the presence of let-polymorphism) which are then solved by simplifying them into the simpler *type constraints* (§4.3.4). Finally, we state our soundness and completeness results for type inference.

4.3.1. Type constraints and solutions

A *type constraint* has either the form $(t_1 \mathrel{\dot{\leq}} t_2)$ or the form $(\tau \mathrel{\dot{\preccurlyeq}} \alpha)$, whose meaning we give below. *Type constraint sets* (ranged over by the metavariable D) are finite sets of type constraints. We write $\mathsf{vars}(D)$ for the set of type variables appearing in the type constraints in D. We write $\mathsf{vars}_{\dot{\preccurlyeq}}(D)$ for the set of type variables appearing in the gradual types in precision constraints in D: that is, $\mathsf{vars}_{\dot{\preccurlyeq}}(D) = \bigcup_{(\tau \mathrel{\dot{\preccurlyeq}} \alpha) \in D} \mathsf{vars}(\tau)$. When $\overline{\alpha} \subseteq \mathscr{V}^{\alpha}$ is a set of type variables and θ is a type substitution, we define $\overline{\alpha}\theta = \bigcup_{\alpha \in \overline{\alpha}} \mathsf{vars}(\alpha\theta)$.

We say that a type substitution $\theta : \mathscr{V}^{\alpha} \to \mathsf{GTypes}$ is a *solution* of a type constraint set D (with respect to a finite set $\Delta \subseteq \mathscr{V}^{\alpha}$), and we write $\theta \Vdash_{\Delta} D$, if:

- for every $(t_1 \mathrel{\dot{\leq}} t_2) \in D$, we have $t_1\theta = t_2\theta$;
- for every $(\tau \mathrel{\dot{\preccurlyeq}} \alpha) \in D$, we have $\tau\theta \preccurlyeq \alpha\theta$ and, for all $\beta \in \mathsf{vars}(\tau)$, $\beta\theta$ is a static type;

- $\mathsf{dom}(\theta) \cap \Delta = \emptyset$.

A subtyping type constraint ($t_1 \mathrel{\dot{\leq}} t_2$) forces the substitution to unify t_1 and t_2. We use $\mathrel{\dot{\leq}}$ instead of, say $\doteq$, to have uniform syntax with the later section on subtyping.

A precision type constraint ($\tau \mathrel{\dot{\preccurlyeq}} \alpha$) imposes two distinct requirements: the solution must make α a materialization of τ and must map all variables in τ to static types. These two conditions might be separated but in practice they must always be imposed together, and their combination simplifies the description of constraint solving. Note that the constraint ($\alpha \mathrel{\dot{\preccurlyeq}} \alpha$) forces $\alpha\theta$ to be static (since the other requirement, $\alpha\theta \preccurlyeq \alpha\theta$, is trivial).

Finally, the set Δ is used to force the solution *not* to instantiate certain type variables (those that belong to Δ).

4.3.2. Type constraint solving

We solve a type constraint set in three steps: we convert the type constraints to unification constraints between type frames (notably, by changing every occurrence of ? into a different frame variable); then we compute a unifier; finally, we convert the unifier into a solution (by renaming some variables and then changing frame variables back to ?).

We define this process as an algorithm $\mathsf{solve}_{(\cdot)}(\cdot)$ which, given a type constraint set D and a finite set $\Delta \subseteq \mathscr{V}^\alpha$, computes a set of type substitutions $\mathsf{solve}_\Delta(D)$. This set is either empty, indicating failure, or a singleton set containing the solution (which is unique up to variable renaming).[4]

We do not describe a unification algorithm explicitly; rather, we rely on properties satisfied by standard implementations (e.g., that by Martelli and Montanari [48]). We use unification on type frames: its input is a finite set $\overline{T^1 \doteq T^2}$ of equality constraints of the form $T^1 \doteq T^2$. We also include as input a finite set $\Delta \subseteq \mathscr{V}^\alpha$ that specifies the variables that unification must *not* instantiate (i.e., that should be treated as constants). We write $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2})$ for the result of the algorithm, which is either fail or a type substitution $\theta : \mathscr{V}^\alpha \cup \mathscr{V}^X \to \mathsf{STypes}$. We assume that unify satisfies the usual soundness and completeness properties and that it computes idempotent substitutions.

Unification is the main ingredient of our type constraint solving algorithm, but we need some extra steps to handle precision constraints, as described below.

Let D be a set of constraints. We write D as $\{(t_i^1 \mathrel{\dot{\leq}} t_i^2) \mid i \in I\} \cup \{(\tau_j \mathrel{\dot{\preccurlyeq}} \alpha_j) \mid j \in J\}$ by separating the two kinds of constraints. The algorithm $\mathsf{solve}_\Delta(D)$ is then defined as follows:

1. Let $\overline{T^1 \doteq T^2}$ be $\{(t_i^1 \doteq t_i^2) \mid i \in I\} \cup \{(T_j \doteq \alpha_j) \mid j \in J\}$ where the T_j are chosen to ensure:
   a) for each $j \in J$, $T_j^\dagger = \tau_j$;
   b) every frame variable X occurs in at most one of the T_j, at most once.
2. Compute $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2})$:
   a) if $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) = \mathsf{fail}$, return $\emptyset$;
   b) if $\mathsf{unify}_\Delta(\overline{T^1 \doteq T^2}) = \theta_0$, return $\{(\theta_0\theta_0')^\dagger|_{\mathscr{V}^\alpha}\}$ where:
      i. $\theta_0' = [\vec{\alpha}'/\vec{X}] \cup [\vec{X}'/\vec{\alpha}]$
      ii. $\vec{X} = \mathscr{V}^X \cap \mathsf{vars}_{\dot{\preccurlyeq}}(D)\theta_0$ and $\vec{\alpha} = \mathsf{vars}(D) \setminus (\Delta \cup \mathsf{dom}(\theta_0) \cup \mathsf{vars}_{\dot{\preccurlyeq}}(D)\theta_0)$
      iii. $\vec{\alpha}'$ and $\vec{X}'$ are vectors of fresh variables

[4] We use a set because, in the presence of subtyping, constraint solving can produce multiple incomparable solutions.

In step 1, we convert D to a set of type frame equality constraints. To do so, we convert all gradual types in precision constraints by replacing each occurrence of ? with a different frame variable. In step 2, we compute a unifier for these constraints. If a unifier θ_0 exists (step 2b), we use it to build our solution: however, we need a post-processing step to ensure that α and X variables are treated correctly. For example, a unifier could map α to X when $(\alpha \dot{\precsim} \alpha) \in D$: then, converting type frames back to gradual types would yield $\alpha := ?$, which is not a solution because α is mapped to a gradual type when a static type is required. This is because variables α appearing on the left of precision constraints are always introduced to type an unannotated λ-abstraction, and we want the inferred type to be static.

Therefore, to obtain the result we first compose θ_0 with a renaming substitution θ_0'; then, we apply $\dagger$ to change type frames back to gradual types, and we restrict the domain to $\mathcal{V}^\alpha$. The renaming θ_0 introduces fresh variables to replace some frame variables with type variables ($[\vec{\alpha}'/\vec{X}]$) and some type variables with frame variables ($[\vec{X}'/\vec{\alpha}]$). It has two purposes:

1. the first is to ensure that the variables in $\mathsf{vars}_{\dot{\precsim}}(D)$ are mapped to static types, which we need for $\theta \Vdash_\Delta D$ to hold. This is done by renaming all the frame variables that are present in the solution of the precision constraints (that is, $\mathcal{V}^X \cap \mathsf{vars}_{\dot{\precsim}}(D)\theta_0$) to type variables;

2. the second is to have the substitution introduce as few type variables as possible. This is done by taking all the type variables occurring in D that are present neither in Δ (forbidding their instantiation), in the domain of the solution θ_0, or in the solution of a precision constraint (forbidding them to be instanciated to a gradual type), and mapping them to a frame variable. This will, in effect, map these type variables to ? after application of the operator $.^\dagger$.

The algorithm $\mathsf{solve}_{(\cdot)}(\cdot)$ satisfies the following soundness property:

Proposition 4.21 (Soundness of solve)**.** *For every set of type constraints D and $\Delta \subseteq \mathcal{V}^\alpha$, if $\theta \in \mathsf{solve}_\Delta(D)$ then the following hold:*

- $\theta \Vdash_\Delta D$*;*
- $\mathsf{vars}(D)\theta \subseteq \mathsf{vars}_{\dot{\precsim}}(D)\theta \cup \Delta$.

The last property states that a solution θ returned by solve introduces as few variables as possible. In particular, the variables it introduces in D are only those in Δ and those that appear in the solutions of variables in $\mathsf{vars}_{\dot{\precsim}}(D)$ (whose solutions must be static). This is the role of the substitution $[\vec{X}'/\vec{\alpha}]$ in the description of solve. This avoids useless materializations of ? to type variables, and thus the insertion of useless casts at compilation.

For example, consider the expression $\mathsf{let}\, y = x \,\mathsf{in}\, E$ where x is given type ?. In the declarative system, y can be given type ?, or it can be typed as $\forall\alpha.\alpha$ by materializing ? into α and generalizing its type. However, in this case, the compiled expression would have a cast: $\mathsf{let}\, y = \Lambda\alpha.\, x\langle ? \Rightarrow_\ell \alpha\rangle \,\mathsf{in}\, E$. We prefer the compilation without this cast, which is why we replace as many α variables as possible with ?. This ensures that, for this example, y is given type ? by solve.

The algorithm is also complete, in the sense that if a solution to a set of constraints D exists, then solve will find a "similar solution":

Proposition 4.22 (Completeness of solve)**.** *For every set of type constraints D and $\Delta \subseteq \mathscr{V}^\alpha$, if there exists θ such that $\theta \Vdash_\Delta D$ then there exists two type substitutions θ' and θ'' such that:*

- $\theta' \in \mathsf{solve}_\Delta(D)$*;*
- *for every $\alpha \in \mathscr{V}^\alpha$, $\alpha\theta'(\theta \cup \theta'') \preccurlyeq \alpha(\theta \cup \theta'')$*
- *for every $\alpha \in \mathscr{V}^\alpha$ such that $\alpha\theta' \in \mathsf{STypes}$, $\alpha\theta'(\theta \cup \theta'') = \alpha(\theta \cup \theta'')$*

A standard statement of completeness would only feature the first and last properties. These two properties state that, if there exists a solution θ of the set of constraints D, then the algorithm will find a solution θ' equal to θ modulo some substitution θ''. However, we add the second weaker property to account for gradual types: the solution θ' produced by solve may introduce less precise types than the solution θ. This means that the types inferred by solve will always be the least precise possible, which in turn ensures that the casts inserted during compilation (based on the solution given by solve) fail as little as possible.

4.3.3. Structured constraints and constraint generation

In the absence of let-polymorphism, the type constraints we presented suffice to describe the conditions for a program to be well-typed (following the approach of Wand [76], augmented with precision constraints). With let-polymorphism, instead, we would need either to mix constraint generation and solving or to copy constraints for let-bound expressions multiple times. To avoid this, we use a kind of constraint that includes binding, following Pottier and Rémy [58].

As anticipated, we will not detail all the constraint generation system in this section, and instead refer the reader to Petrucciani [56] and to Figure A.1 in the Appendix. Here, we summarize the system briefly.

We first define *structured constraints* as the terms generated by the following grammar:

$$C ::= (t \dot{\leq} t) \mid (\tau \dot{\preccurlyeq} \alpha) \mid (x \dot{\preccurlyeq} \alpha) \mid \mathsf{def}\, x : \tau \,\mathsf{in}\, C \mid \exists \vec{\alpha}.\, C \mid C \wedge C \mid \mathsf{let}\, x : \forall \vec{\alpha}; \alpha[C]^{\vec{\alpha}} \,\mathsf{in}\, C$$

Structured constraints include type constraints and five other forms. A constraint $(x \dot{\preccurlyeq} \alpha)$ asks that the type scheme for x has an instance that materializes to the solution of α. Existential constraints $\exists \vec{\alpha}.\, C$ bind the type variables $\vec{\alpha}$ occurring in C; this simplifies freshness conditions, as in Pottier and Rémy [58]. $C \wedge C$ is simply the conjunction of two constraints, while def and let constraints are generated to type λ-abstractions and let-expressions.

We then define a function of the form $\langle\!\langle e : t \rangle\!\rangle$ which generates the structured constraint that must hold for e to be given type t. The idea is to introduce an existentially bound type variable (via a constraint of the form $\exists \alpha.\, C$) for every sub-expression of e, and introduce a precision constraint on every gradual type present in the annotation of a λ-abstraction, as well as on every variable.

For example, for a variable x to have type t, we generate the following constraint:

$$\langle\!\langle x : t \rangle\!\rangle = \exists \alpha.\, (x \dot{\preccurlyeq} \alpha) \wedge (\alpha \dot{\leq} t)$$

which states that there must be some α such that the type of x (which will be deduced later on during constraint rewriting) materializes into α, and this α must be a subtype of t. The solution for the variable α will then give us the type that x must be cast to for the expression to be well-typed.

For λ-abstractions, we introduce a variable for the type of the body and a variable for the type of the parameter, and use the constraint $\mathsf{def}\, x : \tau \,\mathsf{in}\, C$ to bind the type of the parameter. For annotated abstractions, this yields constraints of the form:

$$\langle\!\langle (\lambda x{:}\tau.\, e) : t \rangle\!\rangle = \exists \alpha_1, \alpha_2.\, (\mathsf{def}\, x : \tau \,\mathsf{in}\, \langle\!\langle e : \alpha_2 \rangle\!\rangle) \wedge (\tau \dot{\preccurlyeq} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \dot{\leq} t)$$

Here, we generate the constraints necessary to ensure that e is of type α_2, provided x has been given type τ. Then, since τ is a gradual type and subtyping constraints are only defined on static types, we ask that τ materializes to some type α_1 such that $\alpha_1 \to \alpha_2$ is a subtype of the expected type t.

This contrasts with unannotated abstractions, for which we bind x to α_1 instead:

$$\langle\!\langle (\lambda x.\, e) : t \rangle\!\rangle = \exists \alpha_1, \alpha_2.\, (\mathsf{def}\, x : \alpha_1 \,\mathsf{in}\, \langle\!\langle e : \alpha_2 \rangle\!\rangle) \wedge (\alpha_1 \dot{\preccurlyeq} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \dot{\leq} t)$$

As we already explained before, the constraint $\alpha_1 \dot{\preccurlyeq} \alpha_1$ forces the solution for α_1 to be static, thus ensuring that the inferred type for the parameter is static.

As a last peek inside the constraint generation rules, the following rule generates the constraint for an application:

$$\langle\!\langle e_1\, e_2 : t \rangle\!\rangle = \exists \alpha.\, \langle\!\langle e_1 : \alpha \to t \rangle\!\rangle \wedge \langle\!\langle e_2 : \alpha \rangle\!\rangle$$

This fairly-standard constraint introduces a variable α to unify the type of the argument with the domain of the function, and generates the corresponding constraints. As we precised before, all the remaining rules can be found in Figure A.1 in the appendix.

4.3.4. Constraint solving

While our definition of constraints is mostly based on the work of Pottier and Rémy [58], we approach constraint solving differently, following Castagna et al. [16].

Having described how to generate structured constraints from terms of the source language, and how to solve subtyping and precision constraints with the algorithm solve presented in Subsection 4.3.2, the only step left is to convert structured constraints into sets of type constraints. For this, we define a *constraint simplification* system that rewrites structured constraints into type constraints. However, this is not so simple, as the presence of let-polymorphism forces us to solve constraints and compute partial solutions during their simplification.

Once again, we only give a general intuition about the constraint simplification system, referring the reader to Petrucciani [56] and to Figure A.2 in the Appendix.

Constraint simplification is a relation $\Gamma; \Delta \vdash C \rightsquigarrow D$, where Γ is a type environment used to assign types to the variables in constraints of the form $(x \dot{\preccurlyeq} \alpha)$, and Δ is a finite subset of $\mathscr{V}^{\alpha}$ and is used to record variables that must not be instantiated. When simplifying constraints for a whole program, we take Γ and Δ to be initially empty. Γ will then be enriched with variables bound in $\mathsf{def}\, x : \tau \,\mathsf{in}\, C$ constraints, and Δ will be enriched with the variables whose instantiation is forbidden by let bindings (such as $\vec{\alpha}$ in the expression $\mathsf{let}\, x\, \vec{\alpha} = e \,\mathsf{in}\, e'$). Finally, C is the structured constraint to be simplified and D the result of simplification.

Constraint simplification is defined by a set of syntax-directed and deterministic rules. Subtyping and precision constraints are left unchanged. Variable constraints $(x \dot{\preccurlyeq} \alpha)$ are converted to precision constraints by simply replacing x with a fresh instance of its type scheme, as given by the environment Γ. To simplify a def constraint, we update the environment and simplify the

inner constraint:

$$\frac{(\Gamma, x : \tau); \Delta \vdash C \rightsquigarrow D}{\Gamma; \Delta \vdash \mathsf{def}\, x : \tau \,\mathsf{in}\, C \rightsquigarrow D}$$

For constraints $\exists \vec{\alpha}. C$, we simplify C after performing α-renaming, if needed, to ensure that $\vec{\alpha}$ is fresh. To simplify $C_1 \wedge C_2$, we simplify C_1 and C_2 and take the union of the resulting sets.

The rule for let constraints, produced from an expression $\mathsf{let}\, x\, \vec{\alpha} = e \,\mathsf{in}\, e'$ is of course the most complicated. To summarize briefly, it performs five steps. First, it simplifies the constraints generated from e, while remembering that the $\vec{\alpha}$ variables must not be instantiated. Second, it uses the solve algorithm to deduce a solution of these constraints, if one exists. Third, it generalizes the type given by the solution, producing a type scheme for the variable x. Fourth, it expands the environment Γ with this type scheme, and simplifies the constraints constructed from e'. Finally, it adds some additional constraints to ensure that the solution produced in the second step is compatible with the constraints generated in the fourth step.

As before, all the rules for constraint simplification can be found in the appendix, in Figure A.2.

4.3.5. Algorithmic compilation

To conclude the presentation of the algorithmic system, we define a compilation system. As we already hinted at before, the results of type inference can also be used to compile expressions. In particular, if e is an expression, and if we have a derivation $\mathscr{D}$ of $\Gamma; \Delta \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$ and a solution $\theta \Vdash_\Delta D$, we can compute a cast language expression E by "following" the derivation $\mathscr{D}$, and introducing a cast whenever we construct a precision constraint. For example, if $\mathscr{D}$ contains the rewriting of a variable precision constraint such as $\Gamma; \Delta \vdash (x \mathrel{\dot{\preccurlyeq}} \alpha) \rightsquigarrow (\tau \mathrel{\dot{\preccurlyeq}} \alpha)$ where $\Gamma(x) = \tau$, then the expression x will be compiled into the cast expression $x\langle \tau\theta \Rightarrow_\ell \alpha\theta \rangle$.

Formally, we define this as a function $\{\!|.|\!\}^{(.)}_{(.)}$ which, given a term e, a derivation $\mathscr{D}$, and a solution θ, produces a compiled term $\{\!|e|\!\}^{\mathscr{D}}_{\theta}$. The definition of this function, although straightforward, is very verbose, and can be found in the Appendix, in Figure A.3.

As for constraint solving, type inference is both sound and complete:

Theorem 4.23 (Soundness of type inference)**.** *Let $\mathscr{D}$ be a derivation of $\Gamma; \mathsf{vars}(e) \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Let θ be a type substitution such that $\theta \Vdash_{\mathsf{vars}(e)} D$. Then, we have $\Gamma\theta \vdash e : t\theta \rightsquigarrow \{\!|e|\!\}^{\mathscr{D}}_{\theta}$.*

Theorem 4.24 (Completeness of type inference)**.** *If $\Gamma \vdash e : \tau$, then, for every fresh type variable α, there exist D and θ such that $\Gamma; \mathsf{vars}(e) \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D$ and $[\tau/\alpha] \cup \theta \Vdash_{\mathsf{vars}(e)} D$.*

The latter result, combined with completeness of solve, ensures that inference can compute most general types for all expressions. In particular, starting from a program (i.e., a closed expression) e, we pick a fresh variable α and generate $\langle\!\langle e : \alpha \rangle\!\rangle$. Theorem 4.24 ensures that, if the program is well-typed, we can find a derivation $\mathscr{D}$ for $\emptyset; \emptyset \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D$, and that D has a solution. Since solve is complete, we can compute the principal solution θ of D. Then, $\alpha\theta$ is the most general type for the program and $\{\!|e|\!\}^{\mathscr{D}}_{\theta}$ is its compilation driven by the derivation $\mathscr{D}$.

4.4. Adding subtyping

We have shown that declaratively adding gradual types to an existing type system is as simple as adding a subsumption-like rule for precision. We now show that adding subtyping to the system of the previous section can be done in the same fashion. We just outline the main differences and the necessary additions without giving the details.

We aim at giving simple intuitions about subtyping in gradual type systems and thus prioritize simplicity. We therefore give a simple syntactic definition for subtyping based on static types, instead of a more complex but extension-robust semantic definition of it, which is postponed to the following chapter.

We also highlight the main reasons why extending the type inference algorithm with subtyping is challenging. As we will explain later on, this extension requires some form of union and intersection operations on types, which motivates the integration of set-theoretic types in the next chapter.

4.4.1. Declarative system

Subtyping

We suppose to start from a predefined subtyping preorder relation $\leq$ on $\mathscr{B}$ (e.g., Odd $\leq$ Int $\leq$ Real) and we extend it to the set GTypes of gradual types by the inductive application of the following inference rules:

$$\frac{}{? \leq ?} \qquad \frac{}{\alpha \leq \alpha} \qquad \frac{\tau_1 \leq \tau_1' \qquad \tau_2 \leq \tau_2'}{\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'} \qquad \frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

These rules are standard: covariance for products, co-contravariance for arrows. Just notice that, from the point of view of subtyping, the dynamic type ? is only related to itself, just like a type variable.

Type System

The extension of the source gradual language with subtyping could not be simpler: it suffices to add the standard subsumption rule to the declarative typing rules of Figure 4.1:

$$[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Sub}}] \ \frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau} \ \tau' \leq \tau$$

The definition of the dynamic semantics does not require any essential change, either. The cast calculus is the same as in Section 4.2, except that the $[\mathrm{T}^{\mathsf{HM}}_{\mathsf{Sub}}]$ rule above must be added to its typing rules and the two cast reduction rules that use type equality must be generalized to subtyping, namely:

$$\begin{array}{rlll} [\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{Collapse}}] & V\langle \rho \Rightarrow_p ? \rangle \langle ? \Rightarrow_q \rho' \rangle & \rightsquigarrow & V \qquad \text{if } \rho \leq \rho' \\ [\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{Blame}}] & V\langle \rho \Rightarrow_p ? \rangle \langle ? \Rightarrow_q \rho' \rangle & \rightsquigarrow & \mathsf{blame}\ q \quad \text{if } \rho \not\leq \rho' \end{array}$$

Compared to the rules presented in Figure 4.6, we simply allow the simplification of successive casts of the form $V\langle \rho \Rightarrow_p ? \rangle \langle ? \Rightarrow_q \rho' \rangle$ where $\rho \neq \rho'$, as long as $\rho \leq \rho'$.

The proof of type soundness for this new system is essentially the same as for the system without subtyping. The definition of the compilation of the source language into the "new"

cast calculus does not change either (subsumption is neutral for compilation). The proof that compilation preserves types stays essentially the same, since we have just added the subsumption rule to both systems.

4.4.2. Type inference

The changes required to add subtyping to the declarative system are minimal: define the subtyping relation, add the subsumption rule, and recheck the proofs since they need slight modifications. On the contrary, defining algorithms to decide the relations we just defined is more complicated. As we saw in Section 4.3, this amounts to (1) generating a set of constraints and (2) solving it.

Constraint generation is not problematic. The form of the constraints and the generation algorithm given in Section 4.3 already account for the extension with subtyping: hence, they do not need to be changed, neither here nor in the next section. For constraint generation, the rules presented in Figure A.1 are valid for this system too.

Constraint resolution, instead, is a different matter. In the previous section, constraints of the form $\alpha \dot{\leq} t$ were actually equality constraints (i.e., $\alpha \doteq t$) that could be solved by unification. The same constraints now denote subtyping, and their resolution requires the computation of intersections and unions. To see why, consider the following OCaml code snippet (that does not involve any gradual typing):

```
fun x -> if (fst x) then (1 + snd x) else x
```

We want our system to deduce for this definition the following type:

```
(Bool×Int) → ( Int | (Bool×Int) )
```

since, for an argument of type Bool × Int, the function can either return a value of type Int if its first projection is `true`, or a value of type Bool × Int otherwise.

To that end, a constraint generation system like the one we present in the next section would assign to the function the type $\alpha \to \beta$ and generate the following set of four constraints: $\{(\alpha \dot{\leq} \mathsf{Bool}\times\mathbb{1}), (\alpha \dot{\leq} \mathbb{1}\times\mathsf{Int}), (\mathsf{Int} \dot{\leq} \beta), (\alpha \dot{\leq} \beta)\}$, where, as defined in Chapter 2, $\mathbb{1}$ denotes the top type (that is, the supertype of all types). The first constraint is generated because `fst x` is used in a position where a Boolean is expected; the second comes from the use of `snd x` in an integer position; the last two constraints are produced to type the result of an `if_then_else` expression (with a supertype of the types of both branches). To compute the solution of two constraints of the form $\alpha \dot{\leq} t_1$ and $\alpha \dot{\leq} t_2$, the resolution algorithm must compute the greatest lower bound of t_1 and t_2 (or an approximation thereof); likewise for two constraints of the form $s_1 \dot{\leq} \beta$ and $s_2 \dot{\leq} \beta$ the best solution is the least upper bound of s_1 and s_2. This yields $\mathsf{Bool}\times\mathsf{Int}$ for the domain —i.e., the intersection of the upper bounds for α— and `(Int | (Bool×Int))` for the codomain—i.e., the union of the lower bounds for β.

In summary, to perform type reconstruction in the presence of subtyping, one must be able to compute unions and intersections of types. In some cases, as for the domain in the example above, the solution of these operations is a type of ML (or of the language at issue): then the operations can be meta-operators computed by the type-checker but not exposed to the programmer. In other cases, as for the codomain in the example, the solution is a type which might not already exist in the language: therefore, the only solution to type the expression precisely is to add the corresponding set-theoretic operations to the types of the language.

The full range of these options can be found in the literature. For instance, Pottier [57] defines intersection and union as meta-operations, and it is not possible to simplify the constraints to derive a type like the one above. Hosoya et al. [38] implement a hybrid solution in which intersections are meta-operations while full-fledged unions —which are necessary to encode XML types— are included in types. Other systems include both intersections and unions in the types, starting from the earliest work by Aiken and Wimmers [4] to more recent work by Dolan and Mycroft [22]. Union and intersection types are the most expressive solution but also the one that is technically most challenging; this is why the cited works impose some restrictions on the use of unions and intersections (e.g., no unions in covariant position and no intersections in contravariant ones). In the next chapter, we embrace unrestricted union and intersection types, adding them to both static and gradual types, following the approach of semantic subtyping. This will also require the addition of negation and recursive types.

Chapter 5.

Gradual typing with set-theoretic types

"Invention, it must be humbly admitted, does not consist in creating out of void, but out of chaos."

Mary Shelley, introduction to *Frankenstein*, 1831

In this chapter, we add set-theoretic types to the calculus we presented in the previous chapter. We continue with the same intuition that the dynamic type behaves as an existentially quantified type variable. This allows us to define both precision and subtyping on gradual set-theoretic types, yielding a very simple declarative type system. We then extend the algorithmic system and the cast language to reflect these additions.

Chapter Outline

Section 5.1 We introduce gradual set-theoretic types, and lift all the definitions presented in Chapter 4 to support them. In particular, we redefine type frames, discrimination, and precision.

Section 5.2 We define gradual subtyping for set-theoretic types, and study some of its properties. In particular, we prove its decidability by reducing the subtyping problem on gradual set-theoretic types to subtyping on static set-theoretic types. We also study the interaction between subtyping and precision, showing that the two are commutative.

Section 5.3 We present a cast language that supports set-theoretic types, whose semantics is inspired by the semantics of the language presented in the previous chapter. We prove this semantics to be a sound conservative extension of the semantics presented in Chapter 4. However, we highlight some problems with the syntactic definition of precision, which make the semantics particularly complex.

Section 5.4 We briefly show how to extend the inference algorithm of Chapter 4 to incorporate set-theoretic types. This algorithm is based on an existing inference algorithm for static set-theoretic types, which makes use of recursive types. While the presence of recursive gradual types breaks the completeness of our algorithm, we still prove its soundness.

5.1. Gradual set-theoretic types

5.1.1. Syntax

We start by defining the syntax of the types we are going to use throughout this chapter.

We follow the same approach we presented in the previous chapter: we introduce static types, gradual types, and type frames, which now all support set-theoretic connectives. Type frames

correspond to gradual types where occurrences of the dynamic type ? have been replaced with *frame variables*, which are ranged over by X. We distinguish theses frame variables from *type variables*, ranged over by α, which are used to express polymorphism.

As before, we suppose given a countable set $\mathscr{V}^\alpha$ of type variables, and a countable set $\mathscr{V}^X$ of frame variables. Additionally, to ease the formalism, we will use the metavariable A to range over both types of variables (that is, over $\mathscr{V}^X \cup \mathscr{V}^\alpha$).

As in Chapter 2, we suppose given a set $\mathscr{C}$ of *constants* (ranged over by c), and a set $\mathscr{B}$ of *base types* (ranged over by b). We also consider the two following functions relating these two sets together:

$$b_{(.)} : \mathscr{C} \to \mathscr{B} \qquad \mathbb{B}(.) : \mathscr{B} \to \mathscr{P}(\mathscr{C})$$

where b_c corresponds to the base type of the constant c, and $\mathbb{B}(b)$ is the set of all constants that can be given type b (that is, the set of c such that $b_(c) \leq b$). We assume that every constant has a corresponding singleton type, that is, for every constant c, $\mathbb{B}(b_c) = \{c\}$.

Static types, type frames and gradual types follow the same definition as in the previous chapter, except we add the set-theoretic connectives $\neg$, $\vee$, and the empty type $\mathbb{0}$:

Definition 5.1 (Static types, gradual types, type frames)**.** *The sets* STypes *of* static types*,* GTypes *of* gradual types*, and* TFrames *of* type frames *are the sets of terms t, τ, and T respectively generated coinductively by the following grammars:*

$$\begin{array}{rrll} \mathsf{STypes} \ni t & ::= & \alpha \mid b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0} & \textit{static types} \\ \mathsf{GTypes} \ni \tau & ::= & ? \mid \alpha \mid \tau \times \tau \mid \tau \to \tau \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0} & \textit{gradual types} \\ \mathsf{TFrames} \ni T & ::= & A \mid b \mid T \times T \mid T \to T \mid T \vee T \mid \neg T \mid \mathbb{0} & \textit{type frames} \end{array}$$

(where A ranges over $\mathscr{V}^X \cup \mathscr{V}^\alpha$, α ranges over $\mathscr{V}^\alpha$, and b ranges over $\mathscr{B}$) and that satisfy the following two conditions:

- (regularity) *the term has a finite number of different sub-terms;*
- (contractivity) *every infinite branch of a type contains an infinite number of occurrences of the $\times$ or $\to$ type constructors.*

We introduce the following abbreviations for gradual set-theoretic types, as presented in Chapter 2:

$$\tau_1 \wedge \tau_2 \stackrel{\text{def}}{=} \neg(\neg\tau_1 \vee \neg\tau_2) \qquad \tau_1 \setminus \tau_2 \stackrel{\text{def}}{=} \tau_1 \wedge \neg\tau_2 \qquad \mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}$$

and likewise for type frames and static types. As precised in Chapter 2, we refer to b, $\times$ and $\to$ as *type constructors*, and to $\vee$, $\wedge$, $\neg$ and $\setminus$ as *type connectives*.

Notice that types are defined *coinductively*, so as to support recursive types. Besides the interest of recursive types *per se*, we need them to solve subtyping constraints following a technique introduced by Courcelle [21]. As explained in Chapter 2, the contractivity condition on recursive types allows us to have an induction principle, while the regularity condition ensures the decidability of the subtyping relation for type frames and static types (which will, in turn, ensure that subtyping on gradual types is decidable).

As in the previous chapter, for a given type frame T, we write $\mathsf{vars}(T)$ for the set of variables A occurring in T, and we define $\mathsf{vars}^\alpha(T) = \mathsf{vars}(T) \cap \mathscr{V}^\alpha$ and $\mathsf{vars}^X(T) = \mathsf{Vars}(T) \cap \mathscr{V}^X$. We define the same operations on static and gradual types, for which the result of vars^X is, of course, always empty.

Type substitutions are defined as in Chapter 2, and we extend them to gradual types by defining $?\theta = ?$ for every substitution θ. We will often distinguish static type substitutions, that is, substitutions $\theta : \mathscr{V}^\alpha \cup \mathscr{V}^X \to \mathsf{STypes}$ that map variables into static types, and gradual type substitutions $\theta : \mathscr{V}^\alpha \cup \mathscr{V}^X \to \mathsf{GTypes}$ that map variables into gradual types.

5.1.2. Subtyping on type frames

The set-theoretic interpretation of types with type variables presented in Section 2.3 can be applied to both static types and type frames. The only difference being that, for the latter, the tags of an element $d \in \mathscr{D}^\alpha$ range over $\mathscr{V}^\alpha \cup \mathscr{V}^X$ instead of only $\mathscr{V}^\alpha$. This yields a definition of a subtyping relation $\leq$ and an equivalence relation $\simeq$ on static types and type frames, whose definition we do not repeat here. Simply notice that the subtyping relation thus induced on type frames is a conservative extension of the subtyping relation induced on static types since $\mathsf{STypes} \subseteq \mathsf{TFrames}$, which justifies the use of the same symbol.

We simply recall the following property of the subtyping relation presented in Section 2.3 as Proposition 2.13, which we restate for type frames, and that will be useful later on:

Proposition 5.2. *For every $T_1, T_2 \in \mathsf{TFrames}$, if $T_1 \leq T_2$ then for every type substitution $\theta : \mathscr{V}^X \cup \mathscr{V}^\alpha \to \mathsf{TFrames}$, $T_1\theta \leq T_2\theta$.*

5.1.3. Precision

The definition of precision given in the previous chapter can be extended to gradual set-theoretic types without changing it. As in the previous chapter, given a type frame T, we write $T^\dagger$ for the gradual type obtained by replacing all frame variables occurring in T with $?$. The set $\star(\tau)$ of the *discriminations* of a gradual set-theoretic type τ is still defined as:

$$\star(\tau) \stackrel{\text{def}}{=} \{T \in \mathsf{TFrames} \mid T^\dagger = \tau\}$$

The definition of precision, based on discrimination and type substitutions, does not need to change, even though we have changed the syntax of types. It simply uses the new definition of the discriminations of a gradual type:

$$\tau_1 \preccurlyeq \tau_2 \stackrel{\text{def}}{\iff} \exists T \in \star(\tau_1), \theta : \mathscr{V}^X \to \mathsf{GTypes}.\ T\theta = \tau_2$$

Note that, since types are now defined coinductively, an inductive definition would no longer work.

As for static subtyping, precision is preserved by type substitutions.

Proposition 5.3. *For every $\tau_1, \tau_2 \in \mathsf{GTypes}$, if $\tau_1 \preccurlyeq \tau_2$ then for every type substitution $\theta : \mathscr{V}^\alpha \to \mathsf{GTypes}$, $\tau_1\theta \preccurlyeq \tau_2\theta$.*

Proof. Let $\tau_1, \tau_2 \in \mathsf{GTypes}$ such that $\tau_1 \preccurlyeq \tau_2$, and $\theta : \mathscr{V}^\alpha \to \mathsf{GTypes}$.
By definition of $\preccurlyeq$, there exists $T \in \star(\tau_1), \theta_1 : \mathscr{V}^X \to \mathsf{GTypes}$ such that $T\theta_1 = \tau_2$.
Take any substitution $\theta' : \mathscr{V}^\alpha \to \mathsf{TFrames}$ such that for every $\alpha \in \mathscr{V}^\alpha$, $\alpha\theta' \in \star(\alpha\theta)$ and $\mathsf{vars}^X(\alpha\theta') \cap \mathsf{dom}(\theta_1) = \emptyset$. The first condition ensures that $T\theta' \in \star(\tau_1\theta)$.
Now consider the substitution $\theta'_1 = [X\theta_1\theta/X]_{X \in \mathsf{dom}(\theta_1)} \cup [X/?]_{X \in \mathsf{vars}^X(\theta')}$, and examine

the substitution $\theta'\theta'_1$. We distinguish two cases.

- for every $\alpha \in \mathrm{vars}^{\alpha}(T)$, we have $\alpha\theta'\theta'_1 = \alpha\theta$ since θ'_1 replaces all the frame variables introduced by θ' by ? and $\alpha\theta' \in \star(\alpha\theta)$. And since $\alpha\theta_1 = \alpha$, we have $\alpha\theta'\theta'_1 = \alpha\theta_1\theta$.
- for every $X \in \mathrm{vars}^{X}(T)$, since $T\theta_1 = \tau_2 \in \mathrm{GTypes}$, $X \in \mathrm{dom}(\theta_1)$. Moreover, $X\theta' = X$, and $X\theta'_1 = X\theta_1\theta$ therefore $X\theta'\theta'_1 = X\theta_1\theta$.

We deduce that $\theta'\theta'_1 = \theta_1\theta$. Since $T\theta_1\theta = \tau_2\theta$ and $T\theta' \in \star(\tau_1\theta)$, this proves that $\tau_1\theta \preccurlyeq \tau_2\theta$. □

5.2. Subtyping

Having defined the precision relation on gradual set-theoretic types, we now show how to define subtyping, which, for now, has only been defined on static types and type frames. In the previous chapter, we defined subtyping on gradual types by treating ? exactly like a type variable. The intuition being that subtyping should only act on base types and follow the usual variance rules for arrows and pairs, but should never be able to remove or replace occurrences of ?: this is the role of precision.

We might be tempted to use the same approach here: $\tau_1 \leq \tau_2$ would hold if and only if $T_1 \leq T_2$ holds, where T_1 and T_2 are both obtained respectively from τ_1 and τ_2 by replacing every occurrence of ? with frame variables. However, this relation is not satisfactory. In particular, it would validate $? \setminus ? \leq \mathbb{0}$, since by replacing both occurrences of ? with X, we obtain $X \setminus X \leq \mathbb{0}$. As a consequence, combined with precision, it would imply that the declarative type system would type *every* program, even fully static and nonsensical ones, since any type could be converted to any other by going through the empty type:

$$\tau_1 \leq \tau_1 \setminus (? \setminus ?) \preccurlyeq \tau_1 \setminus (\tau_1 \setminus \mathbb{0}) \leq \mathbb{0} \leq \tau_2$$

Note that such a derivation would insert a cast to $\tau_1 \setminus (\tau_1 \setminus \mathbb{0})$, which will always fail. This is, of course, undesirable, since a proper gradual type system should properly reject an ill-typed, fully statically-typed program.

Therefore, to define subtyping, the idea of replacing ? with type variables requires some care: we must distinguish occurrences that appear below negation connectives from those that do not, to ensure that $? \wedge \neg?$ (i.e., $? \setminus ?$) is not considered to be empty. This means that we cannot reuse our discrimination relation *as is*, since it replaces occurrences of ? in a type with arbitrary (possibly non-distinct) frame variables, independently of their position in the type.

The solution we present in this section consists in defining a new, more restrictive version of the discrimination function, which only generates *polarized* type frames. We say that a type frame is polarized if no frame variable occurs in it in both a positive position (under an even number of negation connectives) and a negative position (under an odd number of negation connectives). For example, $X \setminus X$ is not a polarized discrimination of $? \setminus ?$, but $X \setminus Y$ is (and $X \setminus Y \not\leq \mathbb{0}$). This will allow us to define subtyping on gradual types by saying that τ_1 is a subtype of τ_2 if there exist two *polarized* discriminations T_1, T_2 of τ_1 and τ_2 respectively such that $T_1 \leq T_2$.

We go further by studying some other possible definitions of the discriminations of a type, and the subtyping relations they induce (for example, by also forbidding a variable to occur in both a covariant and a contravariant position). We show that all these definitions are equivalent, all having their own advantages depending on the results we need to prove.

5.2.1. Polarization and variance

A type frame can be represented as an infinite tree where nodes correspond to connectives and constructors ($\vee$, $\to$, and $\times$ being binary, and $\neg$ being unary), and leaves are either a base type b or a variable A. Using such a representation, an occurrence of a variable X in a type frame T can be represented as a string on the alphabet $\{\times_{\mathsf{L}}, \times_{\mathsf{R}}, \to_{\mathsf{L}}, \to_{\mathsf{R}}, \vee_{\mathsf{L}}, \vee_{\mathsf{R}}, \neg\}$ describing the connectives and constructors present along the path from the root of the tree to the leave corresponding to X, as well as the direction of the path for binary nodes. For example, the occurrence of X in $(\mathsf{Int} \to X) \vee \mathsf{Bool}$ can be represented as $\vee_{\mathsf{L}}\to_{\mathsf{R}}$. Of course, the same variable can occur multiple times in the same type frame, even an infinite number of times since types are defined coinductively.

We distinguish three particular characteristics of an occurrence of a variable in a type frame, which we call *polarity*, *parity*, and *variance*.

Definition 5.4 (Polarity, parity, variance)**.** *For every type frame $T \in \mathsf{TFrames}$ and $A \in \mathsf{vars}(T)$, we define the* polarity*,* parity *and* variance *of an occurrence of A in T as follows.*

- Polarity*: an occurrence of A in T is said to be* positive *if the symbol $\neg$ occurs an even number of times in its path, and is* negative *otherwise.*
- Parity*: an occurrence of A in T is said to be* even *if the symbol $\to_L$ occurs an even number of times in its path, and is* odd *otherwise.*
- Variance*: an occurrence of A in T is said to be* covariant *if it is both positive and even or both negative and odd, and is* contravariant *otherwise.*

Notice how the notion of variance coincides with the standard one, where the variance of an occurrence flips every time its path descends below a negation or to the left of an arrow.

To ease the formalism, we define some notation to refer to the variables occurring in a type frame depending on their position. We write $\mathsf{vars}_{\mathsf{cov}}(T), \mathsf{vars}_{\mathsf{con}}(T), \mathsf{vars}_{+}(T)$, and $\mathsf{vars}_{-}(T)$ for, respectively, the set of variables that occur (at least) covariantly, contravariantly, positively, and negatively in a type frame T. We define similar notation for $\mathsf{vars}^X(.)$ and $\mathsf{vars}^\alpha(.)$ for frame variables and type variables respectively. As customary, we also extend all these notions to gradual types and static types, where variables are restricted to $\mathscr{V}^\alpha$.

Finally, we use these notions to introduce the sets of *polarized* and *variance-polarized* type frames.

Definition 5.5 (Polarized and variance-polarized type frames)**.** *For every type frame $T \in \mathsf{TFrames}$, we say that:*

- *T is* polarized *if no frame variable occurs both positively and negatively in it, that is, $\mathsf{vars}^X_{+}(T) \cap \mathsf{vars}^X_{-}(T) = \emptyset$;*
- *T is* variance-polarized *if no frame variable occurs in both a covariant and a contravariant position in it, that is, $\mathsf{vars}^X_{\mathsf{cov}}(T) \cap \mathsf{vars}^X_{\mathsf{con}}(T) = \emptyset$.*

We write $\mathsf{TFrames}^{\mathsf{var}}$ and $\mathsf{TFrames}^{\mathsf{pol}}$ for the set of variance-polarized and polarized type frames, respectively.

5.2.2. Defining subtyping

Having defined polarized type frames, we can now provide the formal definition of subtyping, following the intuition given in the beginning of this section.

First of all, we define the subset of the discriminations of a type that are also polarized:

$$\star^{\text{pol}}(\tau) \stackrel{\text{def}}{=} \star(\tau) \cap \mathsf{TFrames}^{\text{pol}}$$

Then, we use this definition to define subtyping on gradual types:

Definition 5.6 (Subtyping on gradual types)**.** *We define the* subtyping *relation* $\leq$ *between gradual types as follows:*

$$\tau_1 \leq \tau_2 \iff \exists T_1 \in \star^{\text{pol}}(\tau_1), T_2 \in \star^{\text{pol}}(\tau_2).\ T_1 \leq T_2$$

Moreover, we define the equivalence relation $\simeq$ *on gradual types as:*

$$\tau_1 \simeq \tau_2 \iff \tau_1 \leq \tau_2 \text{ and } \tau_2 \leq \tau_1$$

It is straightforward to verify that this relation is a conservative extension of subtyping on type frames as defined in the previous section, hence the use of the same symbol.

It is also easy to check that this is a conservative extension of the definition given in the previous chapter: if τ_1 and τ_2 are non-recursive and do not contain union, negation, or $\mathbb{0}$, then $\tau_1 \leq \tau_2$ holds if and only if it can be derived using the inductive rules of Chapter 4.

This definition of gradual subtyping makes it clear that occurrences of ? under negation types must be handled with care. Alternatively, we could have handled *all* contravariant occurrences of ? similarly, by defining gradual subtyping using variance-polarized type frames, following the same principle. If we define the subset of the discriminations of a type that are variance-polarized:

$$\star^{\text{var}}(\tau) \stackrel{\text{def}}{=} \star(\tau) \cap \mathsf{TFrames}^{\text{var}}$$

gradual subtyping could have been defined as:

$$\tau_1 \leq \tau_2 \iff \exists T_1 \in \star^{\text{var}}(\tau_1), T_2 \in \star^{\text{var}}(\tau_2).\ T_1 \leq T_2$$

It is obvious that this definition entails the first one, since any variance-polarized type frame is also polarized. However, it turns out that both definitions are actually equivalent (we dedicate Subsection 5.2.4 to proving this result), and both have their uses: the first one emphasizes the fact that the problem only arises because of negation types, while the latter is more convenient to use for some proofs.

5.2.3. Decidability of subtyping

The definition of gradual subtyping follows the intuition presented in the previous chapter, in which ? is only compatible with itself, and extends it to set-theoretic types: if τ_1 is a subtype of τ_2 then every occurrence of ? in τ_2 matches an occurrence of ? in τ_1. This notion of matching is formalized by the replacement of ? with frame variables, and the subtyping relation tries to find such a matching via an existential quantification. This, however, could be computationally problematic: the number of possible substitutions grows exponentially with the number of

occurrences of ? in the types at hand, and this number can even be infinite in the presence of recursive types.

However, when computing subtyping, it turns out that we do not need to consider every discrimination of the two types. It is sufficient to consider the single discrimination in which only two frame variables appear: one used to replace all the positive occurrences of ?, and the other used to replace all the negative occurrences of ?; thus eliminating the need for the existential quantification.

We dedicate this subsection and the following to proving this result, which requires some additional terminology and notation. We also prove that the discrimination where a variable is used to replace all covariant occurrences of ? and another for all the contravariant occurrences can be used to obtain the same result, thus yielding a first step towards proving that the two definition of subtyping given in the previous subsection are equivalent.

In the following, we distinguish two particular variables X^1 and X^0 in $\mathscr{V}^X$, and we define the four following sets of type frames:

$$\begin{aligned}
\mathsf{TFrames}^{\mathrm{pos}} &\stackrel{\text{def}}{=} \{T \in \mathsf{TFrames} \mid \mathrm{vars}^X_+(T) \subseteq \{X^1\} \text{ and } \mathrm{vars}^X_-(T) \subseteq \{X^0\}\} \\
\mathsf{TFrames}^{\mathrm{neg}} &\stackrel{\text{def}}{=} \{T \in \mathsf{TFrames} \mid \mathrm{vars}^X_+(T) \subseteq \{X^0\} \text{ and } \mathrm{vars}^X_-(T) \subseteq \{X^1\}\} \\
\mathsf{TFrames}^{\mathrm{cov}} &\stackrel{\text{def}}{=} \{T \in \mathsf{TFrames} \mid \mathrm{vars}^X_{\mathrm{cov}}(T) \subseteq \{X^1\} \text{ and } \mathrm{vars}^X_{\mathrm{con}}(T) \subseteq \{X^0\}\} \\
\mathsf{TFrames}^{\mathrm{con}} &\stackrel{\text{def}}{=} \{T \in \mathsf{TFrames} \mid \mathrm{vars}^X_{\mathrm{cov}}(T) \subseteq \{X^0\} \text{ and } \mathrm{vars}^X_{\mathrm{con}}(T) \subseteq \{X^1\}\}
\end{aligned}$$

It is straightforward to verify that $\mathsf{TFrames}^{\mathrm{pos}}$ and $\mathsf{TFrames}^{\mathrm{neg}}$ contain polarized type frames, and that $\mathsf{TFrames}^{\mathrm{cov}}$ and $\mathsf{TFrames}^{\mathrm{con}}$ contain variance-polarized type frames. We refer to type frames belonging to these four sets as being, respectively, *positively polarized*, *negatively polarized*, *covariantly polarized*, and *contravariantly polarized*.

Now, given a gradual type τ, and for each of these four types of polarization, there exists a unique discrimination of τ that respects this polarization. We use the following notation to refer to these four particular discriminations of τ:

Positive discrimination	$\tau^{\oplus} \in \star(\tau) \cap \mathsf{TFrames}^{\mathrm{pos}}$
Negative discrimination	$\tau^{\ominus} \in \star(\tau) \cap \mathsf{TFrames}^{\mathrm{neg}}$
Covariant discrimination	$\tau^{\textcircled{\scriptsize 0}} \in \star(\tau) \cap \mathsf{TFrames}^{\mathrm{cov}}$
Contravariant discrimination	$\tau^{\textcircled{\scriptsize 1}} \in \star(\tau) \cap \mathsf{TFrames}^{\mathrm{con}}$

While these discriminations cannot be directly defined inductively on types (due to the presence of recursive types), the following equalities hold nonetheless:

$$\begin{aligned}
?^{\oplus} &= X^1 & ?^{\ominus} &= X^0 \\
\alpha^{\oplus} &= \alpha & \alpha^{\ominus} &= \alpha \\
b^{\oplus} &= b & b^{\ominus} &= b \\
(\tau_1 \times \tau_2)^{\oplus} &= \tau_1^{\oplus} \times \tau_2^{\oplus} & (\tau_1 \times \tau_2)^{\ominus} &= \tau_1^{\ominus} \times \tau_2^{\ominus} \\
(\tau_1 \to \tau_2)^{\oplus} &= \tau_1^{\ominus} \to \tau_2^{\oplus} & (\tau_1 \to \tau_2)^{\ominus} &= \tau_1^{\oplus} \to \tau_2^{\ominus} \\
(\tau_1 \vee \tau_2)^{\oplus} &= \tau_1^{\oplus} \vee \tau_2^{\oplus} & (\tau_1 \vee \tau_2)^{\ominus} &= \tau_1^{\ominus} \vee \tau_2^{\ominus} \\
(\neg\tau)^{\oplus} &= \neg\tau^{\ominus} & (\neg\tau)^{\ominus} &= \neg\tau^{\oplus} \\
\mathbb{0}^{\oplus} &= \mathbb{0} & \mathbb{0}^{\ominus} &= \mathbb{0}
\end{aligned}$$

Similar equalities hold for $\tau^{\textcircled{\scriptsize 0}}$ and $\tau^{\textcircled{\scriptsize 1}}$, except the polarity also switches on the left of arrows in

addition to below negations:

$$(\tau_1 \to \tau_2)^{\textcircled{\scriptsize\emptyset}} = \tau_1^{\textcircled{\scriptsize\vee}} \to \tau_2^{\textcircled{\scriptsize\emptyset}} \qquad\qquad (\tau_1 \to \tau_2)^{\textcircled{\scriptsize\vee}} = \tau_1^{\textcircled{\scriptsize\emptyset}} \to \tau_2^{\textcircled{\scriptsize\vee}}$$

As we stated informally before, we will prove that, for every types τ_1 and τ_2, the following equivalences hold:

$$\tau_1 \leq \tau_2 \iff \tau_1^{\oplus} \leq \tau_2^{\oplus} \iff \tau_1^{\textcircled{\scriptsize\emptyset}} \leq \tau_2^{\textcircled{\scriptsize\emptyset}}$$

Note that Proposition 5.2 also ensures that $\tau_1^{\oplus} \leq \tau_2^{\oplus} \iff \tau_1^{\ominus} \leq \tau_2^{\ominus}$, since the positive and negative discriminations of a type are syntactically equal, except the variables X^0 and X^1 are switched. A similar result holds for the covariant and contravariant discriminations, so that the relations induced by subtyping on type frames and each of the four polarizations are all equivalent.

Proving the two equivalences stated above is not straightforward, however. The intuition behind proving the second one, $\tau_1^{\oplus} \leq \tau_2^{\oplus} \iff \tau_1^{\textcircled{\scriptsize\emptyset}} \leq \tau_2^{\textcircled{\scriptsize\emptyset}}$, is that it does not matter whether we use different variables to distinguish occurrences of ? to the left of arrows. Given any two types, subtyping only compares their subterms that occur under the same number of arrows: $\sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2$ can entirely be decided by comparing σ_1 and σ_2, as well as τ_1 and τ_2. The subterms σ_1 and τ_2 are, for example, completely independent.

Proving the first equivalence, namely $\tau_1 \leq \tau_2 \iff \tau_1^{\oplus} \leq \tau_2^{\oplus}$, is trickier. One direction is obvious: $\tau_1^{\oplus} \leq \tau_2^{\oplus}$ clearly implies that $\tau_1 \leq \tau_2$ since $\tau^{\oplus} \in \star(\tau)$ for every type τ. The idea behind proving the converse is to use Proposition 5.2 and apply well-chosen substitutions to the two type frames that prove $\tau_1 \leq \tau_2$, to obtain $\tau_1^{\oplus}$ and $\tau_2^{\oplus}$ from these type frames. It might occur that the types frames verifying $T_1 \leq T_2$ are such that a variable X occurs positively in T_1 and negatively in T_2. In this case, substituting X by any of the two distinguished variables X^0 and X^1 in T_1 and T_2 will never produce similarly polarized type frames. However, in this case, we will prove that the variable X is not actually relevant, and two different substitutions can be applied to T_1 and T_2 while still preserving subtyping.

5.2.4. Equivalence of the definitions of subtyping

In this rather technically difficult subsection, we prove the equivalence of the aforementioned definitions of subtyping. For this, we need some more notation to refer to variables that occur in a type frame in two specific positions: we write $\mathsf{vars}_{+\mathsf{cov}}(T)$, $\mathsf{vars}_{-\mathsf{cov}}(T)$, $\mathsf{vars}_{+\mathsf{con}}(T)$, and $\mathsf{vars}_{-\mathsf{con}}(T)$ for the variables that occur in a positive and covariant position, a negative and covariant position, a positive and contravariant position, and a negative and contravariant position, respectively. Note that, for example, $\mathsf{vars}_{+\mathsf{cov}}(T) \neq \mathsf{vars}_{+}(T) \cap \mathsf{vars}_{\mathsf{cov}}(T)$ because a variable can occur twice in T, once in a negative-covariant position, and once in a positive-contravariant position, but never in a positive-covariant position. Such a variable would thus be in both $\mathsf{vars}_{+}(T)$ and $\mathsf{vars}_{\mathsf{cov}}(T)$, but not in $\mathsf{vars}_{+\mathsf{cov}}(T)$ since it does not occur in a position that is both positive *and* covariant. However, it holds that $\mathsf{vars}_{\mathsf{cov}}(T) = \mathsf{vars}_{+\mathsf{cov}}(T) \cup \mathsf{vars}_{-\mathsf{cov}}(T)$ since every covariant position is also either negative or positive (and similarly for other variances and polarities).

As usual, we use similar notation for vars^{α} and vars^{X}, and extend it to both static types and gradual types.

The first lemma we prove states that, for every type frame T, we can obtain another type frame T' such that T can be obtained from T' by applying a substitution, and no variable occurs in T' in both a covariant and a contravariant position, or both in a positive and a negative position. While this result may seem trivial (taking a type frame T' where every variable occurs only once

may seem to be a trivial solution), it is not, due to recursive types. In a recursive type, a variable X can appear an infinite number of times both covariantly and contravariantly (as in $\mu X.X \to X$ for example), making renaming every occurrence of X difficult.

This result shows that it is indeed possible to substitute the covariant occurrences of X with a distinguished variable and the contravariant occurrences with some other variable while conserving the regularity and contractivity conditions of recursive types (and similarly for positive and negative occurrences). We generalize the result to all kinds of variables (both polymorphic variables and frame variables), which, we recall, are ranged over by the metavariable A.

More precisely, given a type frame T, for every variable A_i that occurs in T, we can introduce four new distinct variables $A_i^{+\wedge}$ (for positive-covariant occurrences), $A_i^{-\wedge}$ (for negative-covariant occurrences), $A_i^{+\vee}$ (for positive-contravariant occurrences), and $A_i^{-\vee}$ (for negative-contravariant occurrences), and replace every occurrence of A_i in T by the variable corresponding to its position. The lemma shows that the resulting term is indeed a type, that is, it satisfies the regularity and contractivity conditions.

Lemma 5.7. *For every type frame T such that $\mathsf{vars}(T) = \{A_i \mid i \in I\}$, there exists a type frame T' such that the four sets*

$$\mathsf{vars}_{+\mathsf{cov}}(T') \subseteq \{A_i^{+\wedge} \mid i \in I\} \qquad \mathsf{vars}_{+\mathsf{con}}(T') \subseteq \{A_i^{+\vee} \mid i \in I\}$$

$$\mathsf{vars}_{-\mathsf{cov}}(T') \subseteq \{A_i^{-\wedge} \mid i \in I\} \qquad \mathsf{vars}_{-\mathsf{con}}(T') \subseteq \{A_i^{-\vee} \mid i \in I\}$$

are pairwise disjoint and such that

$$T = T'\,[A_i/A_i^{+\wedge}]_{i\in I}\,[A_i/A_i^{+\vee}]_{i\in I}\,[A_i/A_i^{-\wedge}]_{i\in I}\,[A_i/A_i^{-\vee}]_{i\in I}$$

Proof. See appendix page 249. □

We then use this lemma to deduce several corollaries, which serve to distinguish variables that occur only in a given position.

Corollary 5.8. *For every type frame T such that $\mathsf{vars}^X(T) = \{X_1, \ldots, X_n\}$, there exists a type frame T' such that:*

- $\mathsf{vars}^X_{\mathsf{cov}}(T') \subseteq \mathsf{vars}^X(T)$
- $\mathsf{vars}^X_{\mathsf{con}}(T') \subseteq \{X'_1, \ldots, X'_n\}$
- $\mathsf{vars}^X_{\mathsf{cov}}(T') \cap \mathsf{vars}^X_{\mathsf{con}}(T') = \emptyset$
- $T = T'\,[X_i/X'_i]_{i\in\{1..n\}}$

Proof. Lemma 5.7 yields a type frame T' verifying the third condition, and stronger conditions than the other three. It is then sufficient to apply a substitution to unify variables that do not need to be distinguished, for example variables in $\mathsf{vars}^X_{\mathsf{cov}}(T')$, or variables in $\mathsf{vars}^X_{+\mathsf{cov}}(T') \cup \mathsf{vars}^X_{-\mathsf{cov}}(T')$. □

Corollary 5.9. *For every type frame T such that $\mathsf{vars}^X(T) = \{X_1, \ldots, X_n\}$, there exists a type frame T' such that:*

- $\mathsf{vars}^X_{\mathsf{even}}(T') \subseteq \mathsf{vars}^X(T)$

- $\mathsf{vars}^X_{\mathsf{odd}}(T') \subseteq \{X'_1, \dots, X'_n\}$
- $\mathsf{vars}^X_{\mathsf{even}}(T') \cap \mathsf{vars}^X_{\mathsf{odd}}(T') = \emptyset$
- $T = T' \, [X_i/X'_i]_{i \in \{1..n\}}$

Proof. Consequence of Lemma 5.7, following the same reasoning as for Corollary 5.8. □

Corollary 5.10. *For every gradual type τ such that* $\mathsf{vars}(\tau) = \{\alpha_1, \dots, \alpha_n\}$*, there exists a gradual type τ' such that:*

- $\mathsf{vars}_+(\tau') \subseteq \mathsf{vars}(\tau)$
- $\mathsf{vars}_-(\tau') \subseteq \{\alpha'_1, \dots, \alpha'_n\}$
- $\mathsf{vars}_+(\tau') \cap \mathsf{vars}_-(\tau') = \emptyset$
- $\tau = \tau' \, [\alpha_i/\alpha'_i]_{i \in \{1..n\}}$

Proof. Consequence of Lemma 5.7, by first choosing a discrimination $T \in \star(\tau)$, then following the same reasoning as for Corollary 5.8, and finally obtaining the gradual type $\tau' = T'^{\dagger}$. □

Recall that subtyping on static types (and, thus, on type frames) can be reduced to an emptiness problem: $T_1 \leq T_2$ if and only if $T_1 \wedge \neg T_2 \leq \mathbb{0}$. One of the main difficulties that arise when proving the final result of equivalence comes from the fact that the statement $T \not\leq \emptyset$ is not, in general, stable by type substitution (although the converse is, by Proposition 5.2). For example, $X \setminus Y \not\leq \emptyset$ holds, but applying the substitution $[X/Y]$ yields $X \setminus X \not\leq \emptyset$ which does not hold.

The following fundamental result, which highlights the importance of the various polarizations, states that non-emptiness is stable by substitution, provided the substitution does not act on variables that occur both positively and negatively.

Lemma 5.11. *For every type frame $T \not\leq \mathbb{0}$, if $\{X, Y\} \sharp \mathsf{vars}^X_-(T)$ or $\{X, Y\} \sharp \mathsf{vars}^X_+(T)$, then $T \, [X/Y] \not\leq \mathbb{0}$.*

Proof. See appendix page 250. □

Using the above lemma and the fact that $T_1 \leq T_2$ if and only if $T_1 \setminus T_2 \leq \mathbb{0}$, we deduce the following corollary which proves that some variables are not relevant when deciding subtyping, and can thus be arbitrarily renamed.

Corollary 5.12. *For all type frames T_1, T_2 such that $T_1 \leq T_2$, for every variable X such that $X \notin \mathsf{vars}^X_+(T_1) \cap \mathsf{vars}^X_+(T_2)$ and $X \notin \mathsf{vars}^X_-(T_1) \cap \mathsf{vars}^X_-(T_2)$, and for all Y such that $Y \sharp X, T_1, T_2$, it holds that $T_1 \, [Y/X] \leq T_2$.*

Proof. See appendix page 253. □

The above corollary provides the first step towards proving the equivalence of the various definitions of subtyping. Ths second step consists in proving that subtyping only compares subterms that occur under the same number of arrows. In other terms, if a variable appears in a type frame with both parities, then all of its negative (or positive) occurrences can be renamed without changing the properties of the type frame.

The first lemma is similar to Lemma 5.11 and states that if a type frame T is non-empty, and contains two variables X and Y that occur with different parities in T, then one can be substituted by the other while keeping the non-emptiness of T.

Lemma 5.13. *For every type frame $T \not\leq \mathbb{0}$ and every variables $X, Y \in \mathscr{V}^X$, if $X \notin \mathsf{vars}^X_{\mathsf{even}}(T)$ and $Y \notin \mathsf{vars}^X_{\mathsf{odd}}(T)$ then $T\,[X/Y] \not\leq \mathbb{0}$.*

Proof. See appendix page 253. □

We use this lemma to prove the next one, which states that if a type frame is empty, then we can find another identical type frame up to substitution that is also empty, and such that no variable occurs in it with both parities.

Lemma 5.14. *For every type frame T, if $T \leq \mathbb{0}$ then there exists a type frame T' and a substitution $\theta : \mathscr{V}^X \to \mathscr{V}^X$ such that:*

- $T' \leq \mathbb{0}$
- $T = T'\theta$
- $\mathsf{vars}^X_{\mathsf{even}}(T') \cap \mathsf{vars}^X_{\mathsf{odd}}(T') = \emptyset$

Proof. See appendix page 256. □

And we finally use this lemma to obtain a similar result for subtyping, following the same idea as for Corollary 5.8.

Corollary 5.15. *For every type frames T_1, T_2, if $T_1 \leq T_2$ then there exists two type frames T_1', T_2' and a substitution $\theta : \mathscr{V}^X \to \mathscr{V}^X$ such that:*

- $T_1' \leq T_2'$
- $T_1 = T_1'\theta$
- $T_2 = T_2'\theta$
- $\mathsf{vars}^X_{\mathsf{even}}(T_1', T_2') \cap \mathsf{vars}^X_{\mathsf{odd}}(T_1', T_2') = \emptyset$

Proof. Let $T = T_1 \setminus T_2$. We have $T \leq \mathbb{0}$ by definition of subtyping.

By Lemma A.5, we find T' and θ such that

$$T' \leq \mathbb{0} \qquad T = T'\theta \qquad \mathsf{vars}^X_{\mathsf{even}}(T') \sharp \mathsf{vars}^X_{\mathsf{odd}}(T') \,.$$

Since T' is empty, it cannot be a type variable or a frame variable. Then, we must have $T' = T_1' \setminus T_2'$ for two types such that $T_1 = T_1'\theta$ and $T_2 = T_2'\theta$.

We have $T_1' \leq T_2'$ by definition of subtyping.

We have $\mathrm{vars}^X_{\mathsf{even}}(T'_1, T'_2) = \mathrm{vars}^X_{\mathsf{even}}(T')$ and $\mathrm{vars}^X_{\mathsf{odd}}(T'_1, T'_2) = \mathrm{vars}^X_{\mathsf{odd}}(T')$, therefore the two sets are disjoint. □

Before continuing with the final proof, we need to distinguish one more particular discrimination of a gradual type. This discrimination is, in essence, the combination of all the others: it uses four distinguished variables, and uses one for each polarity-variance combination. Let $X^{-\wedge}$, $X^{+\wedge}$, $X^{-\vee}$, $X^{+\vee}$ be four distinguished variables in $\mathscr{V}^X$.

Given a gradual type τ, we define $\tau^\bullet$ as the unique type frame $T \in \star(\tau)$ such that $\mathrm{vars}_{+\mathsf{cov}}(T) \subseteq \{X^{+\wedge}\}$, $\mathrm{vars}_{-\mathsf{cov}}(T) \subseteq \{X^{-\wedge}\}$, $\mathrm{vars}_{+\mathsf{con}}(T) \subseteq \{X^{+\vee}\}$, and $\mathrm{vars}_{-\mathsf{con}}(T) \subseteq \{X^{-\vee}\}$.

This discrimination is fundamental because both $\tau^{\textcircled{\wedge}}$ and $\tau^{\oplus}$ can be obtained from $\tau^\bullet$ by substitution, and Proposition 5.2 guarantees that substitutions preserve subtyping between type frames. Therefore, all we need is to prove that subtyping between gradual types reduces to subtyping using this new discrimination. The following lemma states one direction of this result, the converse being immediate.

Lemma 5.16. *For every gradual types τ_1, τ_2, if $\tau_1 \leq \tau_2$ then $\tau_1^\bullet \leq \tau_2^\bullet$.*

Proof. See appendix page 257. □

We prove a similar result for the second definition of subtyping we gave at the end of Subsection 5.2.2, based on the operation $\star^{\mathsf{var}}(.)$.

Lemma 5.17. *For every gradual types τ_1, τ_2, if there exists $T_1 \in \star^{\mathsf{var}}(\tau_1)$ and $T_2 \in \star^{\mathsf{var}}(\tau_2)$ such that $T_1 \leq T_2$, then $\tau_1^\bullet \leq \tau_2^\bullet$.*

Proof. See appendix page 258. □

And finally, we can put all these results together to prove the final result of equivalence, stating that all of the previously defined discriminations induce equivalent definitions of subtyping.

Theorem 5.18 (Equivalence of the definitions of subtyping)**.** *Let τ_1 and τ_2 be two gradual types. The following statements are all equivalent:*

$$\textcircled{1}\ \tau_1 \leq \tau_2 \qquad \textcircled{2}\ \exists T_1 \in \star^{\mathsf{var}}(\tau_1), T_2 \in \star^{\mathsf{var}}(\tau_2).\ T_1 \leq T_2$$

$$\textcircled{3}\ \tau_1^{\oplus} \leq \tau_2^{\oplus} \qquad \textcircled{4}\ \tau_1^{\ominus} \leq \tau_2^{\ominus}$$

$$\textcircled{5}\ \tau_1^{\textcircled{\wedge}} \leq \tau_2^{\textcircled{\wedge}} \qquad \textcircled{6}\ \tau_1^{\textcircled{\vee}} \leq \tau_2^{\textcircled{\vee}}$$

$$\textcircled{7}\ \tau_1^\bullet \leq \tau_2^\bullet$$

Proof. We have already stated that ③ $\iff$ ④ and ⑤ $\iff$ ⑥ are immediate consequences of Proposition 5.2.
The implications ③ $\implies$ ① and ⑤ $\implies$ ② are immediate since $\tau^{\oplus} \in \star^{\mathsf{pol}}(\tau)$ and $\tau^{\textcircled{\wedge}} \in \star^{\mathsf{var}}(\tau)$.
Lemma 5.17 shows that ② $\implies$ ⑦ and Lemma 5.16 shows that ① $\implies$ ⑦.
If $\tau_1^\bullet \leq \tau_2^\bullet$ holds, then taking $\theta = [X^1/X^{+\wedge}]\,[X^1/X^{+\vee}]\,[X^0/X^{-\wedge}]\,[X^0/X^{-\vee}]$ shows that $\tau_1^\bullet\theta \leq \tau_2^\bullet\theta$ by Proposition 5.2. Since $\tau_1^\bullet\theta = \tau_1^{\oplus}$ and $\tau_2^\bullet\theta = \tau_2^{\oplus}$, we have ⑦ $\implies$ ③. This yields a cycle ⑦ $\implies$ ③ $\implies$ ① $\implies$ ⑦.
The same reasoning with $\theta = [X^1/X^{+\wedge}]\,[X^0/X^{+\vee}]\,[X^1/X^{-\wedge}]\,[X^0/X^{-\vee}]$ yields ⑦ $\implies$ ⑤,

and the cycle ⑦ $\implies$ ⑤ $\implies$ ② $\implies$ ⑦.
Combining the two cycles along with the equivalences ③ $\iff$ ④ and ⑤ $\iff$ ⑥ yields the result. □

This final result proves not only that subtyping is decidable by avoiding the existential quantification, but also that it reduces in linear time to subtyping on static types, since the polarized discriminations of a type τ can be computed in linear time from τ (by simply replacing every occurrence of ? by a type variable depending on its position). Moreover, this results gives us several equivalent definitions of gradual subtyping, each having its uses. In the next subsection, we will use two of them to study the properties of gradual subtyping and its interaction with precision.

Before proceeding with the properties of gradual subtyping, note that this decidability result using polarized discriminations only applies to subtyping, and not to precision. For precision, we must still be able to replace all occurrences of ? in a type with possibly distinct frame variables. For example, to decide that $? \vee ? \preccurlyeq \mathsf{Bool} \vee \mathsf{Int}$, we must consider the type frame $X^0 \vee X^1$, which is not polarized. However, this is not problematic since precision only considers syntactic equality up to a single type substitution, which is computationally much easier, as no existential quantification is required.

5.2.5. Properties of subtyping

Equipped with the various equivalent definitions of subtyping, we now study its properties and its interaction with precision.

One of the first questions that come to mind is whether gradual subtyping is preserved by type substitutions, as are subtyping on static types and subtyping on type frames (Proposition 5.2). It is, however, not the case in general due to negation types: it holds that $\alpha \setminus \alpha \leq \mathbb{0}$, but $? \setminus ? \not\leq \mathbb{0}$, even though $?\setminus? = (\alpha\setminus\alpha)\,[?/\alpha]$. However, we can still prove that gradual subtyping is preserved by *static* type substitutions, that is, substitutions that map variables to static types.

Proposition 5.19. *For every gradual types $\tau_1, \tau_2 \in \mathsf{GTypes}$, if $\tau_1 \leq \tau_2$ then for every static type substitution $\theta : \mathscr{V}^\alpha \to \mathsf{STypes}$, $\tau_1\theta \leq \tau_2\theta$.*

Proof. If $\tau_1 \leq \tau_2$, then by Theorem 5.18, we have $\tau_1^\oplus \leq \tau_2^\oplus$. By Proposition 5.2, it holds that $\tau_1^\oplus\theta \leq \tau_2^\oplus\theta$. Since θ cannot introduce frame variables, $\tau_1^\oplus\theta$ is positively polarized. And since θ only acts on type variables, $(\tau_1^\oplus\theta)^\dagger = \tau_1\theta$. Therefore, $\tau_1^\oplus\theta = (\tau_1\theta)^\oplus$. Likewise, $\tau_2^\oplus\theta = (\tau_2\theta)^\oplus$. Hence $(\tau_1\theta)^\oplus \leq (\tau_2\theta)^\oplus$ and $\tau_1\theta \leq \tau_2\theta$ by Theorem 5.18. □

While gradual subtyping is not preserved by gradual type substitutions, we can still deduce an interesting result about such substitutions. Namely, if two (arbitrary) substitutions θ_1 and θ_2 are such that $A\theta_1 \leq A\theta_2$ for every A that appears covariantly in T and $A\theta_2 \leq A\theta_1$ for every A that appears contravariantly in T, then $T\theta_1 \leq T\theta_2$. This result will be necessary to prove many of the results presented later in this section.

Proposition 5.20. *For every type frame $T \in \mathsf{TFrames}$, and every type substitutions $\theta_1, \theta_2 : \mathscr{V}^\alpha \cup \mathscr{V}^X \to \mathsf{GTypes}$ satisfying the following two conditions:*

$$\forall A \in \mathsf{vars}_{\mathsf{cov}}(T), A\theta_1 \leq A\theta_2 \qquad \forall A \in \mathsf{vars}_{\mathsf{con}}(T), A\theta_2 \leq A\theta_1$$

then it holds that $T\theta_1 \leq T\theta_2$.

Proof. See appendix page 259. □

We also prove a property of precision that is analogous to part of Theorem 5.18 for subtyping, namely, that precision can be defined equivalently using variance-polarized discriminations. This result will be useful later on to make the connection between subtyping and precision.

Lemma 5.21. *For every gradual types* $\tau_1, \tau_2 \in \mathsf{GTypes}$*, if* $\tau_1 \preccurlyeq \tau_2$*, then there exists* $T \in \star^{\mathsf{var}}(\tau_1)$ *and* $\theta : \mathscr{V}^X \to \mathsf{GTypes}$ *such that* $T\theta = \tau_2$.

Proof. See appendix page 260. □

The next result we prove is that we can always commute applications of subtyping and precision to apply precision first: if $\tau_1 \leq \tau_2 \preccurlyeq \tau_3$, then there exists a τ_2' such that $\tau_1 \preccurlyeq \tau_2' \leq \tau_3$. This is interesting in order to study the inversion of the typing relation, and has important consequences on the design of the algorithmic type system.

Lemma 5.22. *For every gradual types* $\tau_1, \tau_2, \tau_3 \in \mathsf{GTypes}$*, if* $\tau_1 \leq \tau_2 \preccurlyeq \tau_3$ *then there exists* $\tau_2' \in \mathsf{GTypes}$ *such that* $\tau_1 \preccurlyeq \tau_2' \leq \tau_3$.

Proof. See appendix page 260. □

The converse also holds: we can also commute applications of subtyping and precision to apply subtyping first, and the proof of the result is similar. However, we choose to favour the above direction for the following corollary.

By transitivity of subtyping and precision, this result entails that any sequence of applications of these two relations can be collapsed into an application of precision followed by an application of subtyping. Formally, we define $\lll$ as the preorder on gradual types that combines both relations via the following rules:

$$\frac{}{\tau \lll \tau} \qquad \frac{\tau_1 \leq \tau_2 \quad \tau_2 \lll \tau_3}{\tau_1 \lll \tau_3} \qquad \frac{\tau_1 \preccurlyeq \tau_2 \quad \tau_2 \lll \tau_3}{\tau_1 \lll \tau_3}$$

Then, we obtain the following corollary, formalizing the previous explanation.

Corollary 5.23. *For every gradual types* $\tau_1, \tau_2 \in \mathsf{GTypes}$*,* $\tau_1 \lll \tau_2$ *if and only if there exists* $\tau \in \mathsf{GTypes}$ *such that* $\tau_1 \preccurlyeq \tau \leq \tau_2$.

Proof. Suppose that $\tau_1 \lll \tau_2$. We reason by induction on the derivation of $\tau_1 \lll \tau_2$.
If $\tau_1 = \tau_2$ then $\tau_1 \preccurlyeq \tau_1 \leq \tau_2$.
If $\tau_1 \preccurlyeq \tau' \lll \tau_2$, then by IH we deduce that there exists τ'' such that $\tau' \preccurlyeq \tau'' \leq \tau_2$, and by transitivity of $\preccurlyeq$ we have $\tau_1 \preccurlyeq \tau'' \leq \tau_2$.
If $\tau_1 \leq \tau' \lll \tau_2$, then by IH we deduce that there exists τ'' such that $\tau' \preccurlyeq \tau'' \leq \tau_2$> By Lemma 5.22, we deduce that there exists τ such that $\tau_1 \preccurlyeq \tau \leq \tau''$ and by transitivity of $\leq$, we conclude that $\tau_1 \preccurlyeq \tau \leq \tau_2$.

⋮ The converse is immediate by applying each rule once. □

This last result has two important consequences. The first is that it greatly simplifies the design of the algorithmic type system and its constraint generation step. In particular, this result justifies the rule we presented in Subsection 4.3.3 that generates the constraint associated with a variable:

$$\langle\langle x : t \rangle\rangle = \exists \alpha.\, (x \dot{\preccurlyeq} \alpha) \wedge (\alpha \dot{\leq} t)$$

Using the declarative type system of Figure 4.1, if $\Gamma \vdash x : t$ then it must be because $\Gamma(x) = \tau$ and $\tau \lll t$. But Corollary 5.23 states that $\tau \lll t$ can be collapsed into one application of precision followed by one application of subtyping, which justifies that only one constraint is needed for each relation.

The second consequence of this result is that it proves, for any static type t and any gradual type τ, that $t \lll \tau$ necessarily entails $t \leq \tau$ (since a static type t can only materialize into itself). Note that this does not ensure that τ is static, since $t \leq t \vee ?$ holds, for example. However, this does ensure that our gradual type system still behaves like a static type system in the absence of explicit type annotations containing $?$, since any application of precision in this case can be replaced by an application of subtyping. This means that applying the materialization rule, while not prohibited, will produce casts that will always succeed.

We conclude this subsection with a crucial result about precision and subtyping in the presence of top and bottom types (respectively $\mathbb{1}$ and $\mathbb{0}$ in our framework). Since materialization amounts to replacing occurrences of $?$ with arbitrary gradual types, two particular materializations can be obtained by replacing the occurrences of $?$ with $\mathbb{1}$ or $\mathbb{0}$, depending on their variance. If every covariant occurrence of $?$ in a type τ is replaced with $\mathbb{1}$, and every contravariant occurrence with $\mathbb{0}$, this gives a static type that is a supertype of every materialization of τ. Similarly, by reversing the roles of $\mathbb{1}$ and $\mathbb{0}$, we can obtain a subtype of every materialization of τ. For example, given the type $\tau = ? \to ?$, every materialization of τ is of the form $\tau_1 \to \tau_2$, and is a subtype of $\mathbb{0} \to \mathbb{1}$ and a supertype of $\mathbb{1} \to \mathbb{0}$. We call such materializations the extremal materializations of a type, and introduce some notation to refer to them.

Definition 5.24 (Gradual extrema)**.** *For every gradual type $\tau \in \mathsf{GTypes}$, we define the minimal (resp. maximal) materialization of τ, noted $\tau^{\Downarrow}$ (resp. $\tau^{\Uparrow}$), as the static types obtained as follows:*

$$\mathsf{STypes} \ni \tau^{\Downarrow} \stackrel{\text{def}}{=} \tau^{\oslash}\,[\mathbb{0}/X^1]\,[\mathbb{1}/X^0]$$
$$\mathsf{STypes} \ni \tau^{\Uparrow} \stackrel{\text{def}}{=} \tau^{\oslash}\,[\mathbb{1}/X^1]\,[\mathbb{0}/X^0]$$

We then formalize the intuition behind these interpretations in the following theorem, which is actually a consequence of Proposition 5.20.

Theorem 5.25 (Fundamental property of gradual extrema)**.** *For every gradual type $\tau \in \mathsf{GTypes}$, the following holds*

- *$\tau \preccurlyeq \tau^{\Downarrow}$ and $\tau \preccurlyeq \tau^{\Uparrow}$;*
- *for every $\tau' \in \mathsf{GTypes}$ such that $\tau \preccurlyeq \tau'$, $\tau^{\Downarrow} \leq \tau' \leq \tau^{\Uparrow}$.*

Proof. The statements $\tau \preccurlyeq \tau^{\Downarrow}$ and $\tau \preccurlyeq \tau^{\Uparrow}$ are immediate consequences of the definitions of precision and gradual extrema, since $\tau^{\textcircled{0}} \in \star(\tau)$.

Let τ' such that $\tau \preccurlyeq \tau'$. By Lemma 5.21, there exists $T \in \star^{\mathsf{var}}(\tau)$ and $\theta : \mathscr{V}^X \to \mathrm{GTypes}$ such that $T\theta = \tau'$.

Let $\hat{\theta} = [X^1/X]_{X \in \mathsf{vars}^X_{\mathsf{cov}}(T)} \cup [X^0/X]_{X \in \mathsf{vars}^X_{\mathsf{con}}(T)}$. It holds that $T\hat{\theta} = \tau^{\textcircled{0}}$.

Let $\theta_{\Downarrow} = [\mathbb{0}/X^1]\,[\mathbb{1}/X^0]$ and $\theta_{\Uparrow} = [\mathbb{1}/X^1]\,[\mathbb{0}/X^0]$. We have $\tau^{\Downarrow} = \tau^{\textcircled{0}}\theta_{\Downarrow}$ and $\tau^{\Uparrow} = \tau^{\textcircled{0}}\theta_{\Uparrow}$.

Let $X \in \mathsf{vars}^X_{\mathsf{cov}}(T)$. We have $X\hat{\theta}\theta_{\Uparrow} = X^1\theta_{\Uparrow} = \mathbb{1}$, hence $X\hat{\theta}\theta_{\Uparrow} \geq X\theta$. Let $X \in \mathsf{vars}^X_{\mathsf{con}}(T)$. We have $X\hat{\theta}\theta_{\Uparrow} = X^0\theta_{\Uparrow} = \mathbb{0}$, hence $X\hat{\theta}\theta_{\Uparrow} \leq X\theta$.

Therefore, by Proposition 5.20 on T and the substitutions $\hat{\theta}\theta_{\Uparrow}$ and θ, we deduce $T\theta \leq T\hat{\theta}\theta_{\Uparrow}$, which gives $\tau' \leq \tau^{\Uparrow}$.

Using a similar reasoning on $\theta_{\Downarrow}$, we deduce that $\tau^{\Downarrow} \leq \tau'$. □

In particular, by reflexivity of precision, the extremal materializations of a gradual type τ are comparable with τ itself: $\tau^{\Downarrow} \leq \tau \leq \tau^{\Uparrow}$, hence their name.

The existence of extremal materialization for gradual types has crucial ramifications for gradual type systems, and the semantics of gradual types. Several existing relations can be reduced to static subtyping using these interpretations. For example, given consistent subtyping as defined in Definition 4.6, it can be easily proven that:[1]

$$\tau_1 \mathrel{\widetilde{\leq}} \tau_2 \iff {\tau_1}^{\Downarrow} \leq {\tau_2}^{\Uparrow}$$

In Chapter 12, we will present an interpretation of gradual types for which gradual types are fully determined by their extremal interpretations. In other words, two gradual types τ_1 and τ_2 such that ${\tau_1}^{\Downarrow} \simeq {\tau_2}^{\Downarrow}$ and ${\tau_1}^{\Uparrow} \simeq {\tau_2}^{\Uparrow}$ will be equivalent in this interpretation. We will go even further by providing a definition of subtyping and precision that is entirely based on static subtyping and the extremal interpretations of gradual types (see Theorem 12.14).

While these results do not hold in our system, most notably due to the syntactic nature of precision, we will show later in Chapter 6 how integrating these "more semantic" definitions of subtyping and precision to our system can make the operational semantics of the cast calculus much easier to define and study.

5.3. Source and cast languages

In this section, we briefly present our source language, and then propose a first approach to defining a target language and its operational semantics based on the semantics presented in Section 4.2, adapted to support the definitions of precision and subtyping we gave in the previous sections. However, the definition of this operational semantics proves to be pretty challenging, most notably due to the syntactic aspect of precision. Therefore, we will only highlight the main aspects of our target language, while referring the interested reader to Appendix B for the full proofs and definitions.

5.3.1. Syntax and declarative systems

To add set-theoretic types to the source and target languages, changing their syntax is not necessary, we simply need to allow set-theoretic types wherever types appear (in annotations, casts,

[1] This statement further emphasizes the non-transitivity of consistent subtyping, since generally ${\tau_2}^{\Uparrow} \not\leq {\tau_2}^{\Downarrow}$ and thus ${\tau_1}^{\Downarrow} \leq {\tau_2}^{\Uparrow}$ and ${\tau_2}^{\Downarrow} \leq {\tau_3}^{\Uparrow}$ does not imply ${\tau_1}^{\Downarrow} \leq {\tau_3}^{\Uparrow}$.

and type applications). Therefore, the grammars generating the terms of the source and cast languages as those presented in Chapter 4. We denote by $\mathsf{Terms}^{\mathsf{ST}}$ and $\mathsf{Terms}^{\langle \mathsf{ST} \rangle}$ the sets of terms of the source and cast languages, respectively, to distinguish these from the sets of terms that do not contain set-theoretic types ($\mathsf{Terms}^{\mathsf{HM}}$ and $\mathsf{Terms}^{\langle \mathsf{HM} \rangle}$). It is clear that $\mathsf{Terms}^{\mathsf{HM}} \subseteq \mathsf{Terms}^{\mathsf{ST}}$ and $\mathsf{Terms}^{\langle \mathsf{HM} \rangle} \subseteq \mathsf{Terms}^{\langle \mathsf{ST} \rangle}$.

The type system is also unchanged, following the extension with subtyping presented in Section 4.4. We simply add the following subsumption rule to the system presented in Figure 4.1 for the source language:

$$[\mathrm{T}^{\mathsf{ST}}_{\mathsf{Sub}}] \ \frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau} \ \tau' \leq \tau$$

and likewise to the system presented in Figure 4.5 for the cast language, where $\leq$ is the subtyping relation of Definition 5.6. We also rename the rules to reflect the fact they act on set-theoretic types, and write $[\mathrm{T}^{\mathsf{ST}}_{\cdot}]$ for the typing rules of the source language and $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\cdot}]$ for the typing rules of the target language.

The declarative compilation system is also unchanged from Figure 4.7, except we add the following compilation rule corresponding to an application of $[\mathrm{T}^{\mathsf{ST}}_{\mathsf{Sub}}]$:

$$[\mathrm{C}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}] \ \frac{\Gamma \vdash e \rightsquigarrow E : \tau'}{\Gamma \vdash e \rightsquigarrow E : \tau} \ \tau' \leq \tau$$

As for the type system, we rename the compilation rules as $[\mathrm{C}^{\langle \mathsf{ST} \rangle}_{\cdot}]$ to reflect the fact they manipulate terms with set-theoretic types.

5.3.2. Parallel normal forms and operators

Our goal when defining this semantics was to define a conservative extension of the calculus defined in Section 4.2, and, in particular, of the semantics presented in Figure 4.6. This proved to be extremely challenging due to several reasons, the first of which we outline here.

Recall the reduction rule for applications presented in Figure 4.6:

$$[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{CApp}}] \qquad V\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle \, V' \rightsquigarrow (V \, (V'\langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle))\langle \tau_2 \Rightarrow_p \tau_2' \rangle$$

The idea behind this rule is that, if we have a function V of type $\tau_1 \to \tau_2$, then casting it to $\tau_1' \to \tau_2'$ gives a function that can be applied to arguments of type τ_1', casts them to τ_1, then applies V and casts the result back to τ_2'.

In the presence of set-theoretic types, this is not so simple, as the type of a function is not necessarily a single arrow type (for example, it can be $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$ for an overloaded function). A first idea might be to directly use type operators inspired by those presented in Section 2.4 to compute the domain and codomain of the types at hand, giving a rule of the form:

$$V\langle \tau \Rightarrow_p \tau' \rangle \, V' \rightsquigarrow (V \, (V'\langle \mathsf{dom}(\tau') \Rightarrow_{\overline{p}} \mathsf{dom}(\tau) \rangle))\langle \mathsf{cod}(\tau) \Rightarrow_p \mathsf{cod}(\tau') \rangle$$

There are, however, several problems with such a rule. The first problem is a typing problem: consider, for example, $\tau' = (\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$ and suppose that V' is of type Int. The codomain of τ' is $\mathsf{Int} \vee \mathsf{Bool}$ (since a function of type τ' can produce both integers and booleans), however, the expression $V\langle \tau \Rightarrow_p \tau' \rangle \, V'$ can be given type Int, as τ' can be given type $\mathsf{Int} \to \mathsf{Int}$

by subsumption. Since $\mathsf{Int} \vee \mathsf{Bool} \not\leq \mathsf{Int}$, this shows that subject reduction would not hold with such a rule: the left hand side would have type Int, while the right hand side would only have type $\mathsf{Int} \vee \mathsf{Bool}$.

A solution to this particular problem is to use an operator inspired by the result type operator defined in Section 2.4 rather than a codomain operator, but this requires to know the type of the argument. Therefore, we define an operator $\mathsf{type}(.) : \mathsf{Values}^{\langle \mathsf{ST} \rangle} \to \mathsf{GTypes}$ which returns the type of a value[2]. Note that since abstractions are fully annotated in the cast language, this operator does not use the type system: it simply checks the type annotations.

$$\begin{array}{rclrcl} \mathsf{type}(c) & = & b_c & \mathsf{type}(\lambda^{\tau_1 \to \tau_2} x.\, E) & = & \tau_1 \to \tau_2 \\ \mathsf{type}((V_1, V_2)) & = & (\mathsf{type}(V_1), \mathsf{type}(V_2)) & \mathsf{type}(V\langle \tau_1 \Rightarrow_p \tau_2 \rangle) & = & \tau_2 \end{array}$$

Using this definition, we can then use $\tau \circ \mathsf{type}(V')$ instead of $\mathsf{cod}(\tau)$ in the above rule, and similarly for τ'.

However, there is still a second problem with this rule, which comes from the rather syntactic definition of the precision relation. Recall the typing rules for casts in the cast language:

$$[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cast}+}]\ \frac{\Gamma \vdash E : \tau'}{\Gamma \vdash E\langle \tau' \Rightarrow_l \tau \rangle : \tau}\ \tau' \preccurlyeq \tau \qquad [\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cast}-}]\ \frac{\Gamma \vdash E : \tau'}{\Gamma \vdash E\langle \tau' \Rightarrow_{\bar{l}} \tau \rangle : \tau}\ \tau \preccurlyeq \tau'$$

The domain, result, and projection type operators as presented in Section 2.4 are independent of the syntax of types: they act on types modulo equivalence, by first transforming them into equivalent types in disjunctive normal form. Therefore, given two types τ_1 and τ_2 such that $\tau_1 \preccurlyeq \tau_2$, there is no guarantee that $\mathsf{dom}(\tau_1) \preccurlyeq \mathsf{dom}(\tau_2)$ since the disjunctive normal forms associated with τ_1 and τ_2 might be completely different syntactically. This means that, using a rule such as the one presented above, a well-typed term could reduce to an ill-typed term using the above typing rules for casts.

Our solution consists in computing the two domain types and two result types in *parallel*. For a function cast using $\langle \tau \Rightarrow_p \tau' \rangle$ and applied to a value of type σ, we rewrite τ and τ' in two types in disjunctive normal form τ_N and τ'_N such that $\tau \simeq \tau_N$, $\tau' \simeq \tau'_N$ and $\tau_N \preccurlyeq \tau'_N$. We then extract the part of the domain of τ'_N that can be applied to σ by a purely syntactic operation, and we perform the exact same operation on τ_N, ensuring that the types we obtain are still syntactically similar. We do the same for the result of τ_N and τ'_N applied to σ.

This yields, in the end, an operation $\circ$ that takes a cast $\langle \tau \Rightarrow_p \tau' \rangle$ and a type σ, and returns a new cast $\langle \tau_1 \to \tau'_1 \Rightarrow_p \tau_2 \to \tau'_2 \rangle$ where $\tau_1 \to \tau'_1 \preccurlyeq \tau_2 \to \tau'_2$ (or vice versa, depending on the polarity of p), $\sigma \leq \tau_2 \leq \mathsf{dom}(\tau_2)$, and $\tau'_2 \leq \tau_2 \circ \tau$ and similarly for τ_1. We then use this cast as we would in the absence of set-theoretic types, by separating it into a cast of the argument and a cast of the result:

$$V\langle \tau \Rightarrow_p \tau' \rangle\, V' \rightsquigarrow (V\,(V'\langle \tau_1 \Rightarrow_{\bar{p}} \tau'_1 \rangle))\langle \tau_2 \Rightarrow_p \tau'_2 \rangle \qquad \text{where } \langle \tau \Rightarrow_p \tau' \rangle \circ \mathsf{type}(V') = \langle \tau_1 \to \tau_2 \Rightarrow_p \tau'_1 \to \tau'_2 \rangle$$

The definition of this operator and the underlying disjunctive normal forms for gradual types being quite complex, we refer the reader to Appendix B for the complete definitions. The same idea is also used to define a similar operator for projections, although this one is much simpler since it does not depend on the type of an argument.

[2] Of course, we haven't properly defined the values of the cast language yet. However, we can define the type(.) operator on the subset of $\mathsf{Terms}^{\langle \mathsf{ST} \rangle}$ defined by the grammar $V ::= c \mid \lambda^{\tau \to \tau} x.\, E \mid V\langle \tau \Rightarrow_p \tau \rangle \mid (V, V)$ which will encompass all values

5.3.3. Computing ground types

If we truly want to mimic the operational semantics presented in Section 4.2, a second question arises: what is the set-theoretic equivalent of a ground type? Given a gradual type τ, the ground type associated with τ is meant to be an intermediate type between $?$ and τ that only contains information about the top-level constructor of τ (which is sufficient to determine if a cast should fail or not).

However, in the presence of set-theoretic types, types do not necessarily have a unique top-level constructor. For example, $(\mathsf{Int} \to \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool})$ can be either an arrow or a pair. Intuitively, the ground type associated to this type must retain information about both constructors, and one could expect it to be $(? \to ?) \vee (? \times ?)$. Thus, we could think of defining ground types using the following grammar:

$$\rho ::= b \mid ? \to ? \mid ? \times ? \mid \rho \vee \rho \mid \neg\rho$$

However, to make matters worse, casts do not always lose information about *all* the top-level constructors at once. In Section 4.2, a type less precise than a type $\tau_1 \to \tau_2$ that is not an arrow type is necessarily $?$. However, with set-theoretic types, we can have a cast such as $\langle(\mathsf{Int} \to \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool}) \Rightarrow_p (\mathsf{Int} \to \mathsf{Int}) \vee ?\rangle$. This cast keeps all the information about the $\mathsf{Int} \to \mathsf{Int}$ part of the origin type, but loses all information about the $\mathsf{Bool} \times \mathsf{Bool}$ part. An intermediate type between the two cannot be $(? \to ?) \vee (? \times ?)$: this type is not truly "intermediate" since it is not comparable with $(\mathsf{Int} \to \mathsf{Int}) \vee ?$ using precision. A truly intermediate type would be $(\mathsf{Int} \to \mathsf{Int}) \vee (? \times ?)$: it keeps the information about constructors that has been lost, while also keeping all the information that has been left unchanged by the cast.

This shows that there is no proper definition of the ground type of a set-theoretic type τ. At best, we can define an intermediate type between two type τ_1 and τ_2, provided they are syntactically compatible (i.e., comparable via precision). This leads us to defining the notion of *relative ground types* and a *grounding* operation.

Definition 5.26 (Grounding and relative ground types)**.** *For all types $\tau, \tau' \in \mathsf{GTypes}$ such that $\tau' \leqslant \tau$, we define the* grounding *of τ with respect to τ', noted τ/τ', as follows:*

$$\begin{array}{rcl@{\qquad}rcl}
(\tau_1 \vee \tau_2)/(\tau_1' \vee \tau_2') & = & (\tau_1/\tau_1') \vee (\tau_2/\tau_2') & \neg\tau/\neg\tau' & = & \neg(\tau/\tau') \\
(\tau_1 \vee \tau_2)/? & = & (\tau_1/?) \vee (\tau_2/?) & \neg\tau/? & = & \neg(\tau/?) \\
(\tau_1 \to \tau_2)/? & = & ? \to ? & (\tau_1 \times \tau_2)/? & = & ? \times ? \\
b/? & = & b & \mathbb{0}/? & = & \mathbb{0} \\
\alpha/? & = & \alpha & \tau/\tau' & = & \tau' \qquad \textit{otherwise}
\end{array}$$

A type τ is ground with respect to τ' *if and only if $\tau/\tau' = \tau$.*

The idea behind this definition is to traverse the connectives of both types inductively to get to the top-level constructors, and then keep the least amount of information we can keep about this top-level constructor: if $\tau' = ?$ then we define τ/τ' as $\mathsf{gnd}(\tau)$. Otherwise, we define τ/τ' as τ', since τ' is less precise than τ.

Remark 5.27.
Note that, when computing $\tau_1 \vee \tau_2/?$, we consider $?$ to be equivalent to $? \vee ?$, which allows us to proceed by induction under the $\vee$ connective, and we compute $(\tau_1/?) \vee (\tau_2/?)$. The fact that

$? and ? \vee ?$ are considered equivalent should not be surprising given the properties of the union of types.
However, what is more interesting is that we apply the same reasoning to negation types: when computing $\neg\tau / ?$, we consider $?$ to be equivalent to $\neg ?$, which allows us, once again, to traverse the $\neg$ connective. This hints towards an interpretation that would make $?$ and $\neg ?$ equivalent, which is justified by the fact that both can materialize to the same types. We explore this further in Chapter 6 and Chapter 12. ⌟

As a final remark, the notion of *relative ground types* is also fundamental to compute the application of casts as presented in the previous subsection. Rewriting two types into two equivalent disjunctive normal forms that are syntactically similar is only possible if the two types are syntactically identical above their top-level constructors. As an example, consider $\tau = (\mathsf{Int} \to \mathsf{Int}) \wedge ((\mathsf{Bool} \to \mathsf{Bool}) \vee (\mathsf{Nat} \to \mathsf{Nat}))$ and $\tau' = (\mathsf{Int} \to \mathsf{Int}) \wedge ?$. While it holds that $\tau' \leqslant \tau$, τ' is already in disjunctive normal form while τ is not. And writing τ as an equivalent DNF produces $((\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})) \vee ((\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Nat} \to \mathsf{Nat}))$ which is not syntactically compatible with τ' anymore. The solution is to first go through the intermediate type τ / τ' and compute the DNF of τ / τ' and τ, which are guaranteed to contain the same top-level constructors. The whole difficulty of the operational semantics will therefore be to introduce intermediate ground types whenever needed, and eliminate casts on which the type operators cannot be applied.

5.3.4. Operational semantics

As anticipated, we only outline the main aspects of our semantics, referring the interested reader to Appendix B for the details.

As we stated before, the idea behind this operational semantics is to be as close as possible to the semantics presented in Section 4.2. To achieve this, we use the grounding operation defined in the previous subsection to compute intermediate types, and we compare these intermediate types to determine whether a succession of two casts should fail or can be eliminated. While this may seem simple, formalizing this proves to be a difficult task, due to all the corner cases that can arise from the presence of set-theoretic types.

Whenever we encounter an expression of the form $V\langle \tau_1 \Rightarrow_p \tau_2 \rangle$ where $\tau_2 \leqslant \tau_1$, we distinguish three cases. The first case is when τ_1 is already a ground type relatively to τ_2, that is, $\tau_1 / \tau_2 = \tau_1$. This is analogous to having an expression of the form $V\langle \rho \Rightarrow_p ? \rangle$ in the calculus of Section 4.2. Such an expression cannot be reduced as is, and is considered to be a *boxed* value. It can only be reduced by *unboxing* it, via a cast going to a more precise type (similarly to $[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{Collapse}}]$).

The second case is when the cast from τ_1 to τ_2 does not lose any information from the top-level constructors of τ_1. If this is the case, then we have $\tau_1 / \tau_2 = \tau_2$, since τ_2 is the least precise type between τ_1 and τ_2 that keeps all the information about the top-level constructors of τ_1. This is analogous to having an expression of the form $V\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle$ or $V\langle \tau_1 \times \tau_2 \Rightarrow_p \tau_1' \times \tau_2' \rangle$ in the calculus of Section 4.2. Such an expression also cannot be reduced as is, and is considered to be a value. It will be reduced later on if it is applied to a value or projected, in which case, the reduction uses the operators defined in Subsection 5.3.2.

The third and last case occurs when none of the above conditions are verified. That is, the cast from τ_1 to τ_2 loses information about top-level constructors, and τ_1 is not already a ground type relative to τ_2. In which case, and similarly to the rule $[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{ExpandL}}]$ or Section 4.2, we add an

intermediate type to the cast, giving the following reduction rule:

$$[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{ExpandL}}] \quad V\langle \tau_1 \Rightarrow_p \tau_2 \rangle \rightsquigarrow V\langle \tau_1 \Rightarrow_p \tau_1/\tau_2 \rangle \langle \tau_1/\tau_2 \Rightarrow_p \tau_2 \rangle \quad \text{if } \tau_1/\tau_2 \neq \tau_1 \text{ and } \tau_1/\tau_2 \neq \tau_2$$

Note that all this reasoning applies only if $\tau_2 \preccurlyeq \tau_1$, that is, if the cast goes from a type to a less precise type. If $\tau_1 \preccurlyeq \tau_2$, then all the grounding operations must be reversed, and we obtain, in particular, the following rule:

$$[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{ExpandR}}] \quad V\langle \tau_1 \Rightarrow_p \tau_2 \rangle \rightsquigarrow V\langle \tau_1 \Rightarrow_p \tau_2/\tau_1 \rangle \langle \tau_2/\tau_1 \Rightarrow_p \tau_2 \rangle \quad \text{if } \tau_2/\tau_1 \neq \tau_1 \text{ and } \tau_2/\tau_1 \neq \tau_2$$

Following the same intuition that a "boxing" cast is a cast of the form $\langle \tau_1 \Rightarrow_p \tau_2 \rangle$ where τ_1 is a ground type relatively to τ_2, and an "unboxing" cast is, symmetrically, a cast of the form $\langle \tau_1' \Rightarrow_p \tau_2' \rangle$ where τ_2' is a ground type relatively to τ_1', we deduce an equivalent of the rules $[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{Blame}}]$ and $[\mathrm{R}^{\langle \mathsf{HM} \rangle}_{\mathsf{Collapse}}]$ presented in the previous chapter:

$$\begin{array}{rlll}
[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Collapse}}] & V\langle \tau_1 \Rightarrow_p \tau_2 \rangle \langle \tau_1' \Rightarrow_q \tau_2' \rangle & \rightsquigarrow & V \qquad \text{if } \tau_1 \leq \tau_2' \text{ and } \tau_1/\tau_2 = \tau_1 \text{ and } \tau_2'/\tau_1' = \tau_2' \\
[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Blame}}] & V\langle \tau_1 \Rightarrow_p \tau_2 \rangle \langle \tau_1' \Rightarrow_q \tau_2' \rangle & \rightsquigarrow & \mathsf{blame}\ q \quad \text{if } \tau_1 \not\leq \tau_2' \text{ and } \tau_1/\tau_2 = \tau_1 \text{ and } \tau_2'/\tau_1' = \tau_2'
\end{array}$$

All the other rules of the semantics are there to deal with corner cases that do not occur in a language that does not contain set-theoretic types. For example, a cast may try to immediately "unbox" a value that has not been boxed. In the absence of set-theoretic types, this would amount to having an expression of the form $V\langle ? \Rightarrow_p \rho \rangle$ where V is not boxed. This is impossible: for such an expression to be well-typed, V must have type $?$. However, this is only possible if it is first boxed by a cast to $?$. With set-theoretic types, $?$ can be introduced by subtyping, yielding, for example, the cast $(\lambda^{\mathsf{Int}\to\mathsf{Int}} x.\, x)\langle ? \vee (\mathsf{Int} \to \mathsf{Int}) \Rightarrow_p (? \to ?) \vee (\mathsf{Int} \to \mathsf{Int}) \rangle$. According to our informal definition, such a cast is an "unboxing" cast (the target type is ground with respect to the source type), but the above elimination rules cannot apply since there is no "boxing" cast around the value. Therefore, we need separate rules to handle such casts.

Our semantics satisfies the same soundness property we stated in Chapter 4, which we prove using the standard lemmas of progress and subject reduction.

Lemma 5.28 (Progress)**.** *For every term $E \in \mathsf{Terms}^{\langle \mathsf{ST} \rangle}$, if $\emptyset \vdash E : \forall \vec{\alpha}.\tau$ then one of the following holds:*

- *there exists $E' \in \mathsf{Terms}^{\langle \mathsf{ST} \rangle}$ such that $E \rightsquigarrow E'$;*
- *there exists $\ell \in \mathscr{L}$ such that $E \rightsquigarrow \mathsf{blame}\ \ell$;*
- *$E \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$.*

Lemma 5.29 (Subject reduction)**.** *For every term $E, E' \in \mathsf{Terms}^{\langle \mathsf{ST} \rangle}$, if $\Gamma \vdash E : \forall \vec{\alpha}.\tau$ and $E \rightsquigarrow E'$ then $\Gamma \vdash E' : \forall \vec{\alpha}.\tau$.*

Theorem 5.30 (Soundness)**.** *For every term $E \in \mathsf{Terms}^{\langle \mathsf{ST} \rangle}$, if $\emptyset \vdash E : \forall \vec{\alpha}.\tau$ then one of the following holds:*

- *there exists $V \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ such that $E \rightsquigarrow^* V$;*
- *there exists $\ell \in \mathscr{L}$ such that $E \rightsquigarrow^* \mathsf{blame}\ \ell$;*

- *E diverges.*

As for the calculus presented in the previous chapter, the following blame safety property holds as an immediate corollary of the above soundness theorem.

Corollary 5.31 (Blame safety)**.** *For every term $E \in \mathsf{Terms}^{\langle \mathsf{HM} \rangle}$ and every blame label $\ell \in \mathscr{L}$, if $\emptyset \vdash E : \forall \vec{\alpha}.\tau$ then $E \not\rightsquigarrow^* \mathsf{blame}\ \overline{\ell}$.*

Lastly, an important aspect of the cast language defined in this section is that it is a conservative extension of the cast calculus defined in Chapter 4; which justifies the choice of the reduction rules. Denoting by HM the system with subtyping defined in Section 4.4 and by ST the system defined in this section, there is a strong bisimulation relation between ST and HM, as stated by the following result.

Theorem 5.32 (Conservativity)**.** *For every term $E \in \mathsf{Terms}^{\langle \mathsf{HM} \rangle}$ such that $\emptyset \vdash_{\mathsf{HM}} E : \tau$, $E \rightsquigarrow_{\mathsf{HM}} E' \iff E \rightsquigarrow_{\mathsf{ST}} E'$ and $E \rightsquigarrow_{\mathsf{HM}} \mathsf{blame}\ p \iff E \rightsquigarrow_{\mathsf{ST}} \mathsf{blame}\ p$.*

This result ensures that all the properties of the cast language presented in Section 4.2 follow by conservativity from the properties of this extension.

5.4. Inference

In Chapter 4, we described a type inference algorithm for a system without subtyping, and briefly described the problems that arise when adding subtyping. That description was intended to be extended here; this motivated some design choices, such as the use of subtyping constraints. Now we describe what must be changed to adapt the system to set-theoretic types. As stated in the previous chapter, we only outline the general idea of the algorithm, referring the reader to the appendix for the formal definitions and to Petrucciani [56] for more details and the complete proofs.

5.4.1. Type constraints and solutions

We keep the same definition for type constraints except, of course, for the different definition of types. However, the conditions for a type substitution θ to be a solution of a constraint D in Δ must be changed: subtyping constraints now require subtyping instead of equality. So we now write $\theta \Vdash_\Delta D$ when:

- for every $(t_1 \mathrel{\dot{\leq}} t_2) \in D$, we have $t_1\theta \leq t_2\theta$;
- for every $(\tau \mathrel{\dot{\preccurlyeq}} \alpha) \in D$, we have $\tau\theta \preccurlyeq \alpha\theta$ and, for all $\beta \in \mathsf{vars}(\tau)$, $\beta\theta$ is a static type;
- $\mathsf{dom}(\theta) \cap \Delta = \emptyset$.

5.4.2. Type constraint solving

To solve type constraint sets, we replace unification with an algorithm designed for set-theoretic types and semantic subtyping: the *tallying* algorithm of Castagna et al. [15]. Given a set $\overline{T^1 \mathrel{\dot{\leq}} T^2}$ of subtyping constraints between type frames, tallying computes a finite set Θ of type substitutions such that, for all $\theta \in \Theta$ and $(T^1 \mathrel{\dot{\leq}} T^2) \in \overline{T^1 \mathrel{\dot{\leq}} T^2}$, we have $T^1\theta \leq T^2\theta$. The set computed by tallying can contain multiple incomparable substitutions. Thus, unlike unification where the

principal solution to the problem is a unique substitution, the principal solution for tallying is *a set* of substitutions.

For example, the constraint $(\alpha \times \beta) \dot{\leq} (\mathsf{Int} \times \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool})$ has four solutions, two of them being $\{\alpha \coloneqq \mathsf{Int}, \beta \coloneqq \mathsf{Int}\}$ and $\{\alpha \coloneqq \mathsf{Bool}, \beta \coloneqq \mathsf{Bool}\}$, which are not comparable (the other two are the trivial solutions $\{\alpha \coloneqq \mathbb{0}\}$ and $\{\beta \coloneqq \mathbb{0}\}$). Nevertheless, the tallying algorithm of Castagna et al. [15] is sound and complete with respect to the tallying problem (i.e., checking whether there exists a substitution solving a set $\overline{T^1 \dot{\leq} T^2}$ of subtyping constraints) insofar as *the set* of substitutions computed by the algorithm is principal: any other solution is an instance of one in the set.

We want to use tallying to define an algorithm to solve type constraints. Previously, we converted precision constraints $(\tau \dot{\preccurlyeq} \alpha)$ to equality constraints $(T \doteq \alpha)$ and used unification. To do the same here, we first need to extend tallying to handle such equality constraints. This is easy to do in our case by adding simple pre- and post-processing steps. The pre-processing step ensures the algorithm fails immediately should constraints be ill-formed (for example, if there are two constraints $(T_1 \doteq \alpha)$ and $(T_2 \doteq \alpha)$ where $T_1 \neq T_2$). In practice, the constraints generated by our constraint generation system are always well-formed and the pre-processing step of the algorithm never fails.

The pre-processing step ends by taking every equality constraint $(T \doteq \alpha)$ and performing the substitution $[T/\alpha]$ in all subtyping constraints. The algorithm then applies the tallying algorithm of Castagna et al. [15] to the remaining subtyping constraints. And finally, for every substitution θ obtained as a solution, we simply extend it with the substitutions $[T\theta/\alpha]$ that correspond to the equality constraints.

The resulting algorithm $\mathsf{tally}^{\doteq}_{(\cdot)}(\cdot)$ satisfies the following property:

$$\forall \theta \in \mathsf{tally}^{\doteq}_{\Delta}\left(\overline{t^1 \dot{\leq} t^2} \cup \overline{T \doteq \alpha}\right). \quad \begin{cases} \forall (t^1 \dot{\leq} t^2) \in \overline{t^1 \dot{\leq} t^2}.\ t^1\theta \leq t^2\theta \\ \forall (T \doteq \alpha) \in \overline{T \doteq \alpha}.\ T\theta = \alpha\theta \\ \mathsf{dom}(\theta) \subseteq \mathsf{vars}\left(\overline{t^1 \dot{\leq} t^2} \cup \overline{T \doteq \alpha}\right) \setminus \Delta \end{cases}$$

Using $\mathsf{tally}^{\doteq}$, we can define the version of solve for set-theoretic types following the same approach as before. However, there are two difficulties.

The main difficulty is the presence of recursive types and their behaviour with respect to precision. Consider the recursive type defined by the equation $\tau = (? \times \tau) \vee b$, where b is some basic type. It corresponds to the type of lists of elements of type $?$, terminated by a constant in b. Since recursive types in our definition are infinite regular trees, $\tau = (? \times \tau) \vee b$ and $\tau' = (? \times ((? \times \tau') \vee b)) \vee b$ denote exactly the same type. What types can τ materialize to? Clearly, both $\tau_1 = (\mathsf{Int} \times \tau_2) \vee b$ and $\tau_2 = (\mathsf{Int} \times ((\mathsf{Bool} \times \tau_2) \vee b)) \vee b$ are possible. Indeed, $?$ occurs infinitely many times in τ. Precision could in principle allow us to change each occurrence to a different type. However, since types must be regular trees, only a finite number of occurrences can be replaced with different types (otherwise, the resulting tree would not be a gradual type). While finite, this number is unbounded.

Recall that step 1 of the algorithm solve picked a discrimination T_j of each τ_j such that no frame variable appeared more than once in T_j. If we consider the recursive type τ above, there is no T such that $T^\dagger = \tau$ and that T has no repeated frame variables: it would need to have infinitely many frame variables and thus be non-regular. While we will never need infinitely many variables, we do not know in advance (in this pre-processing step) how many we will need.

A solution to this would be to change the tallying algorithm so that discrimination is performed during tallying. Then, it could be done lazily, introducing as many frame variables as needed. However, this sacrifices some of the modularity of our current approach.

We chose to give a definition where no constraint is placed on how many frame variables are used to replace ?. Of course, a sensible choice is to use different variables as much as possible except for the infinitely many occurrences of ? in a recursive loop.

There is a second difficulty. For a subtyping constraint $(t_1 \dot{\leq} t_2)$, a substitution θ computed by tallying ensures $t_1\theta \leq t_2\theta$. However, what we want is rather $(t_1\theta)^\dagger \leq (t_2\theta)^\dagger$. This does not necessarily hold unless the type frames $t_1\theta$ and $t_2\theta$ are polarized. For example, if the constraint is $(\alpha \dot{\leq} \mathbb{0})$ and the substitution is $[X \setminus X/\alpha]$, we have $X \setminus X \leq \mathbb{0}$ but $? \setminus ? \not\leq \mathbb{0}$. Therefore, we need to define solve so that it ensures polarization in these cases. This can be done by tweaking the variable renaming step we already had.

Apart from these differences, the algorithm follows the same idea as the algorithm of the previous chapter, using $\text{tally}^{\dot{=}}$ instead of unification. The formal definition of the algorithm solve can be found in the appendix.

The algorithm solve is sound, but due to the presence of recursive gradual types (as explained above), it is not complete:

Proposition 5.33 (Soundness of solve)**.** *If $\theta \in \text{solve}_\Delta(D)$, then $\theta \Vdash_\Delta D$.*

5.4.3. Structured constraints, generation, and simplification

The syntax of structured constraints can be kept unchanged except for the change in the syntax of types. Constraint generation is also unchanged. Constraint simplification still uses the same rules, but it relies on the new solve algorithm. Soundness still holds, with the same statement as Theorem 4.23:

Theorem 5.34 (Soundness of type inference)**.** *Let $\mathscr{D}$ be a derivation of $\Gamma; \text{vars}(e) \vdash \langle\!\langle e : t \rangle\!\rangle \rightsquigarrow D$. Let θ be a type substitution such that $\theta \Vdash_{\text{vars}(e)} D$. Then, we have $\Gamma\theta \vdash e : t\theta \rightsquigarrow \{\!|e|\!\}^{\mathscr{D}}_\theta$.*

However, completeness no longer holds, mainly as a consequence of the possible materializations of ? in recursive types. Therefore, the first step to attempt to recover completeness for inference would be to study how to change the solve algorithm to make it complete.

Note also that type constraint solving can now produce more than one incomparable solution. So constraint simplification is non-deterministic: in the rule for let constraints, there can be multiple solutions to try. Soundness ensures that any solution will give a type and a compiled expression that are sound with respect to the declarative system.

Chapter 6.

A set-theoretic foundation for casts

"La perfection est atteinte, non pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à retirer."

Antoine de Saint-Exupéry

The semantics we presented in the previous chapter in Section 5.3 was obtained by directly extending the semantics presented in Section 4.2 with set-theoretic types. The core ideas were the same, which allowed us to prove that the semantics of Section 5.3 is a sound conservative extension of the semantics of Section 4.2.

However, this extension requires the introduction of many complex definitions (relative ground types, parallel normal forms, type operators) to accommodate for the presence of set-theoretic types. Due to their semantic nature (type connectives are not meant to be manipulated syntactically), which contrasts with the syntactic nature of the semantics presented in Section 4.2, set-theoretic types proved to be more of a hindrance than a help.

The crux of the problem comes from the syntactic definition of precision: precision must preserve the syntactic structure of types, which means that, for example, $? \vee \mathsf{Bool}$ is less precise than $\mathsf{Int} \vee \mathsf{Bool}$ but is not comparable with $\mathsf{Bool} \vee \mathsf{Int}$, even though both types are equivalent. The advantage of this definition is twofold: first, it provides a very simply way to embed gradual typing in any type system, only requiring a notion of type variables and type substitutions. Second, being based on syntactic equality between types and type substitutions, it can easily be added to any type inference algorithm by adding a simple unification step.

However, when it comes to the cast language, this makes any semantic manipulation of types (such as the application of one of the type operators presented in Section 2.4) cumbersome. The major manifestation of this problem being, arguably, the definition of *relative ground types* and the extremely complex operational semantics that follows.

This motivated us to study gradual typing from a denotational perspective, with the aim of finding a semantic definition of precision, similarly to the definition of *semantic subtyping* presented in Section 2.3. The details of this study are covered in Part II of this manuscript. In this section, we introduce some of the findings from that study (namely, new interpretations of precision and subtyping on gradual types) and use them to deduce some powerful results about the representation of gradual types. We then use these results to greatly simplify the semantics presented in Section 5.3.

Chapter Outline

Section 6.1 In this first section, we introduce the *semantic precision* and *semantic gradual subtyping* relations, based on the findings of Chapter 12. Although we do not present the formal set-theoretic interpretation of types that led to these relations, we show how they can be intuitively deduced from the relations presented in the previous chapter. We then

prove a crucial result about the representation of gradual set-theoretic types, which states that every type can be represented equivalently by a type containing a single occurrence of the dynamic type.

Section 6.2 We use the aforementioned result about the representation of gradual types to deduce gradual extensions of the type operators presented in Chapter 2. The resulting definitions are simple and intuitive, and we show that they immediately inherit the soundness properties of the static type operators, which greatly simplifies the proofs.

Section 6.3 We present a new operational semantics for the language of Chapter 5, based on the new relations and operators. We emphasize the simplicity of this semantics compared to the previous one. We prove the soundness of this semantics, and show that most of it follows from the properties of the operators we defined before.

Section 6.4 Finally, we conclude the chapter with a short summary of the results and improvements presented in this chapter.

6.1. Semantic precision and semantic gradual subtyping

6.1.1. A conservative first attempt

The notion of equivalence between types is central in semantic subtyping: two types are equivalent if and only if they represent the same set of values, and therefore behave identically in every context. Following this intuition, two equivalent gradual types should materialize into the same types. However, this is currently not the case. Consider for example the types $\mathsf{Int} \vee ?$ and $? \vee \mathsf{Int}$: although they are equivalent (for obvious reasons), the former materializes into $\mathsf{Int} \vee \mathsf{Bool}$ while the latter does not. The latter does, however, materialize to $\mathsf{Bool} \vee \mathsf{Int}$ which is *equivalent* to $\mathsf{Int} \vee \mathsf{Bool}$.

This leads us to considering precision *modulo equivalence*, in which a type τ_1 is more precise than τ_2 if τ_1 and τ_2 are respectively equivalent to two types τ_1' and τ_2' such that τ_1' is more precise (syntactically) than τ_2'. Formally, the resulting relation, which we call *semantic precision*, is defined as follows:

Definition 6.1 (Semantic precision)**.** *We define the* semantic precision *relation* $\tilde{\preccurlyeq}$ *between gradual types as follows:*

$$\tau_1 \tilde{\preccurlyeq} \tau_2 \stackrel{\text{def}}{\iff} \exists \tau_1', \tau_2' \in \mathsf{GTypes}.\ \tau_1 \simeq \tau_1' \preccurlyeq \tau_2' \simeq \tau_2$$

It is straightforward to show that this yields a relation that is more general than precision: if $\tau_1 \preccurlyeq \tau_2$ then $\tau_1 \tilde{\preccurlyeq} \tau_2$.

While the relation $\tilde{\preccurlyeq}$ does not suffer from the same syntactic limitations as $\preccurlyeq$ and ensures that two equivalent types behave identically, its definition still relies on the latter, which makes reasoning about it impractical. However, we can find an equivalent definition of $\tilde{\preccurlyeq}$ than only relies on subtyping on static types, and the extremal materializations of gradual types.

To establish this equivalent definition, we first need the following lemma, which states that if two gradual types are in a subtyping relation, then so are their extremal materializations.

Lemma 6.2. *For every gradual types* τ_1, τ_2*, if* $\tau_1 \leq \tau_2$ *then* ${\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow}$ *and* ${\tau_1}^{\Uparrow} \leq {\tau_2}^{\Uparrow}$*.*

Proof. By Theorem 5.18, we have $\tau_1^{\textcircled{\scriptsize\&}} \leq \tau_2^{\textcircled{\scriptsize\&}}$. By Proposition 5.2, it holds that $\tau_1^{\textcircled{\scriptsize\&}}[\mathbb{0}/X^1][\mathbb{1}/X^0] \leq \tau_2^{\textcircled{\scriptsize\&}}[\mathbb{0}/X^1][\mathbb{1}/X^0]$. And by Definition 5.24, this yields ${\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow}$. Using a similar reasoning, we deduce ${\tau_1}^{\Uparrow} \leq {\tau_2}^{\Uparrow}$. □

Using this lemma, we can prove the following result, which states that semantic precision reduces to subtyping on static types using the extremal materializations of gradual types.

Proposition 6.3. *For every gradual types $\tau_1, \tau_2 \in \mathsf{GTypes}$, the following holds:*

$$\tau_1 \precapprox \tau_2 \iff \begin{cases} {\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow} \\ {\tau_2}^{\Uparrow} \leq {\tau_1}^{\Uparrow} \end{cases}$$

Proof. We prove the two implications separately.

- Suppose that $\tau_1 \precapprox \tau_2$. By definition, there exists τ_1', τ_2' such that $\tau_1 \simeq \tau_1' \preccurlyeq \tau_2' \simeq \tau_2$. By Lemma 6.2, we deduce that ${\tau_1}^{\Downarrow} \simeq {\tau_1'}^{\Downarrow}$ and ${\tau_1}^{\Uparrow} \simeq {\tau_1'}^{\Uparrow}$ ①, and similarly for τ_2 and τ_2' ②.
  Now, by Theorem 5.25, it holds that ${\tau_1'}^{\Downarrow} \leq \tau_2'$ and $\tau_2' \leq {\tau_1'}^{\Uparrow}$. By Lemma 6.2, we deduce that ${\tau_1'}^{\Downarrow} \leq {\tau_2'}^{\Downarrow}$ and ${\tau_2'}^{\Uparrow} \leq {\tau_1'}^{\Uparrow}$ ③.
  Finally, using results ①, ② and ③ and by transitivity of subtyping, we deduce that ${\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow}$ and ${\tau_2}^{\Uparrow} \leq {\tau_1}^{\Uparrow}$.

- Now suppose that ${\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow}$ and ${\tau_2}^{\Uparrow} \leq {\tau_1}^{\Uparrow}$. Consider $\tau_1' = \tau_1 \vee (\tau_1 \wedge \tau_2)$, and $\tau_2' = {\tau_1}^{\Downarrow} \vee ({\tau_1}^{\Uparrow} \wedge \tau_2)$. By Theorem 5.25, it holds that $\tau_1' \preccurlyeq \tau_2'$. Moreover, since $\tau_1 \wedge \tau_2 \leq \tau_1$, we have immediately $\tau_1 \simeq \tau_1'$.
  Now, by hypothesis, ${\tau_2}^{\Uparrow} \leq {\tau_1}^{\Uparrow}$. By Theorem 5.25, this entails $\tau_2 \leq {\tau_1}^{\Uparrow}$. Thus, $\tau_2 \wedge {\tau_1}^{\Uparrow} \simeq \tau_2$. We also have by hypothesis ${\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow}$. Theorem 5.25 yields ${\tau_1}^{\Downarrow} \leq \tau_2$. Therefore, ${\tau_1}^{\Downarrow} \vee \tau_2 \simeq \tau_2$. Finally, this yields $\tau_2' \simeq \tau_2$, hence the result.

□

This result conveys a very strong message: any gradual type can be seen as an *interval*[1] of *possible types*, where ? denotes the interval of all types, and a type τ denotes the interval ranging from $\tau^{\Downarrow}$ to $\tau^{\Uparrow}$ (or, more precisely, the sub-lattice of the types included between the two). Semantic precision then allows us to reduce this interval, by going to any type τ' such that $\tau^{\Downarrow} \leq {\tau'}^{\Downarrow}$ and ${\tau'}^{\Uparrow} \leq \tau^{\Uparrow}$, possibly until reaching a static type (that is, a type τ such that $\tau^{\Downarrow} = \tau^{\Uparrow}$).

Continuing on this interpretation of gradual types as intervals, it would be tempting to introduce a semantic definition of gradual subtyping, in which a type τ_1 is a subtype of τ_2 if the interval denoted by τ_1 only contains subtypes of elements of the interval denoted by τ_2. Formally, this would amount to defining gradual subtyping so that Lemma 6.2 is an equivalence:

$$\tau_1 \lesssim \tau_2 \overset{\text{def}}{\iff} \begin{cases} {\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow} \\ {\tau_1}^{\Uparrow} \leq {\tau_2}^{\Uparrow} \end{cases}$$

However, such a definition is problematic, as we will discuss in more details in Chapter 12. For now, remark that using such a definition, it holds that $? \to ? \leq ? \to \mathbb{0}$, as we have $(? \to ?)^{\Downarrow} =$

[1] It is, actually, a lattice, thanks to the presence of the intersection and union connectives. However, as we will see later, this lattice is entirely determined by its maximal and minimal elements (i.e., two types with the same extremal materializations represent the same lattice), hence the use of the term "interval".

$\mathbb{1} \to \mathbb{0} \leq \mathbb{1} \to \mathbb{0} = (? \to \mathbb{0})^{\Downarrow}$ and $(? \to ?)^{\Uparrow} = \mathbb{0} \to \mathbb{1} \leq \mathbb{0} \to \mathbb{0} = (? \to \mathbb{0})^{\Uparrow}$. Thus, by subsumption, it is possible to give type $\mathbb{0}$ to the application of a function of type $? \to ?$ to a value of type $?$, which is unsound as such an application *can* produce a result.

This problem occurs because, under the definition of semantic subtyping presented in Chapter 2, all types $\mathbb{0} \to t$ are equivalent. Hence, comparing two arrow types whose domain is $?$ amounts to comparing only their minimal materializations, as their maximal materializations will always be equivalent. This allows subtyping to bypass the standard variance properties of arrow types even when their domains are non-empty, thus allowing unsound subsumptions.

6.1.2. Modifying semantic subtyping

There are two solutions to this problem. The first is to simply use semantic precision in conjunction with gradual subtyping as defined in the previous chapter. However, this is not satisfactory, since this would mean subtyping would still rely on the various discrimination operations, while precision would not. Additionally, precision and subtyping would induce different equivalence relations on gradual types, although the intuition dictates that two types that behave identically for precision should also behave identically for subtyping, and vice versa. This is not the case for $?$ and $\neg?$ for example: both materialize to the same types (since $\neg\neg\tau$ is equivalent to τ) but they are not related by subtyping.

The second solution, guided by our research in Chapter 12, is to slightly modify the interpretation of types of semantic subtyping and the induced subtyping relation so that the types $\mathbb{0} \to t$ are not equivalent anymore. This makes gradual subtyping as defined above using the extremal materializations sound, at the cost of the conservativity of semantic subtyping as defined in Chapter 2.

The change is straightforward: we introduce a new element $\mho$ to the input of relations in the interpretation of semantic subtyping, and, for a relation to be in the interpretation of a type $t_1 \to t_2$, we force an output to be in the interpretation of t_2 whenever the corresponding input is $\mho$. Formally, the new interpretation domain is defined as follows.

Definition 6.4 (New interpretation domain)**.** *The interpretation domain $\mathscr{D}'$ is the set of finite terms d produced inductively by the following grammar:*

$$\begin{aligned} d &::= c^L \mid (d, d)^L \mid \{(\iota, \partial), \dots, (\iota, \partial)\}^L \\ \iota &::= d \mid \mho \\ \partial &::= d \mid \Omega \end{aligned}$$

where c ranges over the set $\mathscr{C}$ of constants, L ranges over $\mathscr{P}_f(\mathscr{V}^\alpha)$, and where Ω and $\mho$ are such that $\Omega, \mho \notin \mathscr{D}'$.
We also write $\mathscr{D}'_\Omega = \mathscr{D}' \cup \{\Omega\}$ and $\mathscr{I}' = \mathscr{D}' \cup \{\mho\}$.

And, following our above explanation, the new interpretation of types is defined by only changing the interpretation of arrow types presented in Definition 2.5 to account for the new input $\mho$:

$$(\{(\iota_i, \partial_i) \mid i \in I\}^L : t_1 \to t_2) = \forall i \in I.\ (\iota_i = \mho \vee (\iota_i : t_1)) \implies (\partial_i : t_2)$$

We write $[\![.]\!]' : \text{Types} \to \mathscr{P}(\mathscr{D}')$ the new interpretation of types this predicate induces, defined by $[\![t]\!]' = \{d \in \mathscr{D}' \mid (d : t)\}$.

Using this new interpretation, the interpretation of the type $\mathbb{0} \to \mathbb{1}$ still contains all relations, whereas the interpretation of, for example, $\mathbb{0} \to \mathsf{Int}$ does not contain the relation $\{\mathfrak{O}, \mathtt{true}\}$, as for a relation to belong to $\mathbb{0} \to \mathsf{Int}$, it must map $\mathfrak{O}$ to an integer. Thus, the two types are now distinct. This ensures that all arrow types verify the usual variance properties for subtyping.

In the following, we consider the relation $\dot{\leq}$ to be the subtyping relation induced on Types by the above interpretation, following Definition 2.6. Note that it is straightforward to see that, for every type t, $[\![t]\!]' = \emptyset \implies [\![t]\!] = \emptyset$, which in turn implies that if $t_1 \dot{\leq} t_2$ then $t_1 \leq t_2$.

6.1.3. Semantic gradual subtyping

We can now lift this new subtyping relation from static types to gradual types using the aforementioned method. This allows us to obtain a subtyping relation on gradual types, which we call *semantic gradual subtyping*, which is a conservative extension of the subtyping relation on static types.

Definition 6.5 (Semantic gradual subtyping)**.** *We define the* semantic gradual subtyping *relation* $\gtrsim$ *between gradual types as follows:*

$$\tau_1 \gtrsim \tau_2 \overset{\text{def}}{\iff} \begin{cases} {\tau_1}^{\Downarrow} \dot{\leq} {\tau_2}^{\Downarrow} \\ {\tau_1}^{\Uparrow} \dot{\leq} {\tau_2}^{\Uparrow} \end{cases}$$

And we define the semantic gradual equivalence *relation* $\approx$ *as* $\tau_1 \approx \tau_2 \overset{\text{def}}{\iff} \tau_1 \gtrsim \tau_2$ *and* $\tau_2 \gtrsim \tau_1$.

This relation will be derived in Chapter 12 using set-containment on a new set-theoretic interpretation of gradual types, similarly to semantic subtyping as presented in Section 2.3.

Note that, as anticipated, using this relation $\neg?$ is equivalent to $?$, which was not the case before. While surprising at first, this result is actually intuitive: $?$ and $\neg?$ have the same materializations modulo equivalence, since every type τ is equivalent to $\neg\neg\tau$. Therefore, following the intuition we gave in the beginning of this section, $?$ and $\neg?$ behave identically and should be considered equivalent.

Naturally, by modifying the subtyping relation on static types, we also indirectly modified the semantic precision relation whose definition was based on the former. In the following, we consider semantic precision to be defined following Proposition 6.3, that is:

$$\tau_1 \gtrsim\!\!\!\!\!\!\sim \tau_2 \overset{\text{def}}{\iff} \begin{cases} {\tau_1}^{\Downarrow} \dot{\leq} {\tau_2}^{\Downarrow} \\ {\tau_2}^{\Uparrow} \dot{\leq} {\tau_1}^{\Uparrow} \end{cases}$$

As for subtyping, we will show in Chapter 12 how this definition can be introduced semantically, using a set-theoretic interpretation of gradual types.

Note that since our new subtyping relation on Types implies the semantic subtyping relation defined in Chapter 2, it also holds that this definition of precision implies the previous one presented in Definition 6.1 as a first attempt. The converse is obviously not true, for the same reasons as subtyping: since $\mathbb{0} \to \mathbb{1}$ and $\mathbb{0} \to \mathsf{Int}$ are equivalent for the old definition of subtyping, they are also materializations of each other for the old definition of semantic precision. However, these two types are incomparable under the new definition of semantic precision.

Having introduced these new precision and subtyping relations, we now add them to the type

system of the cast language, by replacing the rules involving the old relations. Only three rules need to change, the others being those that can be found in Figure 4.5.

$$[\mathrm{T}^{\langle \mathrm{ST} \rangle}_{\mathrm{Sub}}]\ \frac{\Gamma \vdash E : \tau'}{\Gamma \vdash E : \tau}\ \tau' \gtrsim \tau \qquad [\mathrm{T}^{\langle \mathrm{ST} \rangle}_{\mathrm{Cast+}}]\ \frac{\Gamma \vdash E : \tau'}{\Gamma \vdash E\langle \tau' \Rightarrow_l \tau \rangle : \tau}\ \tau' \gtrapprox \tau \qquad [\mathrm{T}^{\langle \mathrm{ST} \rangle}_{\mathrm{Cast-}}]\ \frac{\Gamma \vdash E : \tau'}{\Gamma \vdash E\langle \tau' \Rightarrow_{\bar{l}} \tau \rangle : \tau}\ \tau \gtrapprox \tau'$$

6.1.4. Some properties

We now establish several properties of semantic gradual subtyping and semantic precision that will be useful to define our new semantics. The first property restates Proposition 2.13 for our new subtyping relation on static types (generalized to type frames). Its proof follows the same strategy as the proof of Proposition 2.13: we adapt the interpretation of Definition 2.8 to account for the new input $\mho$, and prove the same properties.

Proposition 6.6. *For every types $T_1, T_2 \in \mathsf{TFrames}$, if $T_1 \dot{\leq} T_2$ then $T_1\theta \dot{\leq} T_2\theta$ for every type substitution θ.*

Proof. See appendix page 262. □

This result allows us to restate Proposition 5.2 and Proposition 5.3 for our new relations, proving that they are both stable by static type substitutions. This result is considerably easier to prove than Proposition 5.3, thanks to the fact that semantic precision is defined using static subtyping, and thus inherits its properties.

Proposition 6.7. *For every gradual types τ_1, τ_2, and every static type substitution $\theta : \mathscr{V}^\alpha \to \mathsf{STypes}$, the following holds:*

$$\tau_1 \gtrsim \tau_2 \implies \tau_1\theta \gtrsim \tau_2\theta$$
$$\tau_1 \gtrapprox \tau_2 \implies \tau_1\theta \gtrapprox \tau_2\theta$$

Proof. Immediate consequence of Proposition 6.6, and the fact that for every type τ, $(\tau\theta)^{\Uparrow} = \tau^{\Uparrow}\theta$ and $(\tau\theta)^{\Downarrow} = \tau^{\Downarrow}\theta$. □

By also restating Proposition 5.20 for our new subtyping relation (see appendix page 262), we can also deduce the following crucial property, which ensures that we still have $\tau^{\Downarrow} \dot{\leq} \tau^{\Uparrow}$ for every gradual type τ.

Lemma 6.8. *For every gradual type $\tau \in \mathsf{GTypes}$, we have $\tau^{\Downarrow} \dot{\leq} \tau^{\Uparrow}$.*

Proof. See appendix page 263. □

As a corollary, this lemma entails that Theorem 5.25 still holds for our new relations.

Corollary 6.9. *For every gradual type $\tau \in \mathsf{GTypes}$, the following holds*

- *$\tau \gtrapprox \tau^{\Downarrow}$ and $\tau \gtrapprox \tau^{\Uparrow}$;*
- *for every $\tau' \in \mathsf{GTypes}$ such that $\tau \gtrapprox \tau'$, $\tau^{\Downarrow} \gtrsim \tau' \gtrsim \tau^{\Uparrow}$.*

Proof. By Definition 6.5, it is clear that for every static types $t_1, t_2 \in \mathsf{Types}$, $t_1 \dot{\leq} t_2 \iff t_1 \precsim t_2$, since ${t_1}^{\Downarrow} = {t_1}^{\Uparrow} = t_1$ and similarly for t_2.

Since $\tau^{\Downarrow\Downarrow} = \tau^{\Downarrow}$, we have $\tau^{\Downarrow} \dot{\leq} \tau^{\Downarrow\Downarrow}$ and by Lemma 6.8 we have $\tau^{\Downarrow\Uparrow} = \tau^{\Downarrow} \dot{\leq} \tau^{\Uparrow}$, hence $\tau \precapprox \tau^{\Downarrow}$ by definition of $\precapprox$. The same reasoning yields $\tau \precapprox \tau^{\Uparrow}$.

Now let $\tau' \in \mathsf{GTypes}$ such that $\tau \precapprox \tau'$. We have $\tau^{\Downarrow} \dot{\leq} \tau'^{\Downarrow}$ by definition of $\precapprox$. Additionally, by Lemma 6.8, we have $\tau'^{\Downarrow} \dot{\leq} \tau'^{\Uparrow}$ hence $\tau^{\Downarrow} \dot{\leq} \tau'^{\Uparrow}$ by transitivity of subtyping. This proves that $\tau^{\Downarrow} \precsim \tau'$. A similar reasoning yields $\tau' \precsim \tau^{\Uparrow}$. □

Finally, we prove a fundamental and surprising result about the representation of gradual types, which states that every gradual type τ can be represented using a single occurrence of ?, bounded by $\tau^{\Downarrow}$ and $\tau^{\Uparrow}$ using type connectives. In essence, this result formalizes the intuition that a gradual type ranges over an interval of types bounded by two static types.

Theorem 6.10. *For every gradual type τ,*

$$\tau \cong \tau^{\Downarrow} \vee (? \wedge \tau^{\Uparrow})$$

Proof. See Theorem 12.15 (page 221). □

This result has very strong consequences: we do not need the full syntax of gradual types, we simply need static types and a single top-level occurrence of ? (or a way to denote the types ranging from a static type to another).

The importance of this result will become even more apparent in the next section: given a type operator defined on static types, it can be extended to any gradual type τ by simply applying it to the extremal materializations of τ, and reconstructing the result using the above theorem.

6.2. Type operators

The syntactic definition of precision was the main reason behind the introduction of complex type operators in Chapter 5. Their goal was to ensure we always preserved the precision relation between two types in a cast, particularly when applying a cast function or projecting a cast pair. This was necessary to ensure type preservation: a cast is allowed only between two types where one is a materialization of the other.

Now that we introduced a semantic precision, which is less dependent on the syntax of types, we can redefine the type operators in a much simpler way. In particular, these new operators are not defined on casts anymore, they are simply defined on types as in Section 2.4. We then prove later on that they preserve semantic precision in Proposition 6.20, 6.21, and 6.22.

6.2.1. Properties of the static operators

Unfortunately, since we changed the interpretation of types on which the operators defined in Section 2.4 are based, we need to make some changes to the definition of the static type operators, and restate and prove their soundness properties.

Thankfully, the domain (Definition 2.19) and projection (Definition 2.21) operators do not need to change. The only change our new interpretation brings has to do with the result type of an application when the considered domain is empty: since $\mathbb{0} \to \mathbb{1}$ and $\mathbb{0} \to \mathbb{0}$ were equivalent

under the previous interpretation of types, the most precise type that could be inferred for an application of a function of type $\mathbb{0} \to \mathbb{1}$ to an argument of type $\mathbb{0}$ was $\mathbb{0}$. Under our new interpretation, the most precise type that can be inferred is now $\mathbb{1}$, since the type $\mathbb{0} \to \mathbb{1}$ cannot be subsumed to a type with a smaller codomain. We slightly modify Definition 2.20 to account for this change.

Definition 6.11 (Result type operator)**.** *For every type $t \dot{\leq} \mathbb{0} \to \mathbb{1}$ and every type s such that $s \dot{\leq} \mathsf{dom}(t)$, we define the* result type *of t applied to s, noted $t \circ s$ as:*

$$t \circ s \overset{\text{def}}{=} \bigvee_{i \in I} \bigvee_{Q \subsetneq P_i} \bigwedge_{s_i \to t_i \in P_i \setminus Q} t_i \qquad \text{if } s \dot{\not\leq} \mathbb{0}, \; s \dot{\not\leq} \bigvee_{s_i \to t_i \in Q} s_i$$

$$\overset{\text{def}}{=} \bigvee_{i \in I} \bigwedge_{s_i \to t_i \in P_i} t_i \qquad \text{otherwise}$$

where

$$\mathscr{N}_U(t) = \bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)$$

This new definition of the result type operator on static types follows the same idea as Definition 2.20, except we add an additional clause for when the type of the argument is empty. In this case, we simply take the type of the result to be the union of the intersections of the codomains of the arrows that compose $\mathscr{N}_U(t)$.

Using the new subtyping relation on static types, this new definition of the result type operator, as well as the domain and projection operators as defined in Section 2.4, we restate the same soundness properties we presented in Section 2.4. We do not detail the proofs of these properties as they can be obtained from the proofs presented by Frisch et al. [27] by simply replacing their subtyping relation with ours. The only difference comes from the decomposition lemma for subtyping on arrow types, which, as anticipated, features a new condition that handles the case of an empty domain. As it is fairly complex and out of the scope of this section, we relegate its statement and proof to the appendix (Lemma A.22, page 264).

Proposition 6.12. *For every types $t, t' \in \mathsf{Types}$, if $t \dot{\leq} \mathbb{0} \to \mathbb{1}$ then $t \dot{\leq} \mathsf{dom}(t) \to \mathbb{1}$ and if $t \dot{\leq} t' \to \mathbb{1}$ then $t' \dot{\leq} \mathsf{dom}(t)$.*

Proposition 6.13. *For every types $t, t', s \in \mathsf{Types}$, if $t \dot{\leq} \mathbb{0} \to \mathbb{1}$ and $s \dot{\leq} \mathsf{dom}(t)$ then $t \dot{\leq} s \to t \circ s$. Moreover, if $t \dot{\leq} s \to t'$ then $t \circ s \dot{\leq} t'$.*

Proposition 6.14. *For every types $t, t_1, t_2 \in \mathsf{Types}$, if $t \dot{\leq} \mathbb{1} \times \mathbb{1}$ then $t \dot{\leq} \pi_1(t) \times \pi_2(t)$ and if $t \dot{\leq} t_1 \times t_2$ then $\pi_1(t) \dot{\leq} t_1$ and $\pi_2(t) \dot{\leq} t_2$.*

6.2.2. Definition of the gradual operators

To define these new operators, we once again follow the idea that a gradual type denotes an interval. If $\tau \cong \mathbb{1} \times \mathbb{1}$ and represents the interval ranging from $\tau^{\Downarrow}$ to $\tau^{\Uparrow}$, then its first projection must denote the interval ranging from $\pi_1\, \tau^{\Downarrow}$ to $\pi_1\, \tau^{\Uparrow}$, where π_1 is defined on static types in Section 2.4. We then use Theorem 6.10 to reconstruct the projection $\pi_1\, \tau$ of τ as $(\pi_1\, \tau^{\Downarrow}) \vee ((\pi_1\, \tau^{\Uparrow}) \wedge ?)$.

The definitions of the other two operators follow a similar reasoning, extending the definition of their static counterparts given in Section 2.4, except care must be taken when dealing with contravariance: if τ ranges from $\tau^{\Downarrow}$ to $\tau^{\Uparrow}$, then its domain must range from $\mathsf{dom}(\tau^{\Uparrow})$ to $\mathsf{dom}(\tau^{\Downarrow})$ (notice the inversion of $\tau^{\Uparrow}$ and $\tau^{\Downarrow}$), since $\mathsf{dom}(\tau^{\Uparrow}) \dot{\leq} \mathsf{dom}(\tau^{\Downarrow})$. Similarly, the type of the result of an application is covariant in the type of the function but contravariant in the type of the argument: for a function of type τ and an argument of type τ', the smallest possible result is obtained when τ is the smallest and τ' is the largest, and conversely.

Definition 6.15 (Gradual type operators)**.** *For every gradual type τ, τ', we define the following gradual type operators:*

$$\begin{array}{rrcl} \textit{Domain operator} & \widetilde{\mathsf{dom}}(\tau) & \stackrel{\text{def}}{=} & \mathsf{dom}(\tau^{\Uparrow}) \vee (? \wedge \mathsf{dom}(\tau^{\Downarrow})) \\ \textit{Result operator} & \tau \mathbin{\tilde{\circ}} \tau' & \stackrel{\text{def}}{=} & (\tau^{\Downarrow} \circ \tau'^{\Uparrow}) \vee (? \wedge (\tau^{\Uparrow} \circ \tau'^{\Downarrow})) \\ \textit{Projection operator} & \widetilde{\pi_i}(\tau) & \stackrel{\text{def}}{=} & (\pi_i(\tau^{\Downarrow})) \vee (? \wedge (\pi_i(\tau^{\Uparrow}))) \end{array}$$

The value of these operators is undefined whenever the corresponding static type operators are undefined.

Note that since for every static type t, $t^{\Downarrow} = t^{\Uparrow} = t$, and since $t \vee (t \wedge ?) \cong t$, it is straightforward to verify that these operators are conservative extensions of their static counterparts. They also verify intuitive results, such as $\widetilde{\mathsf{dom}}(? \to ?) = \mathbb{0} \vee (? \wedge \mathbb{1}) \cong ?$ and similarly $(? \to ?) \mathbin{\tilde{\circ}} ? \cong ?$.

Remark 6.16.
To summarize, the general principle we use to extend type operators from static types to gradual types is the following. Given a type operator $\mathrm{F} : \mathsf{Types} \to \mathsf{Types}$*, we define its gradual extension* $\widetilde{\mathrm{F}} : \mathsf{GTypes} \to \mathsf{GTypes}$ *as*

$$\widetilde{\mathrm{F}}(\tau) = \mathrm{F}(\tau^{\Downarrow}) \vee (\mathrm{F}(\tau^{\Uparrow}) \wedge ?)$$

provided F *is increasing for* $\dot{\leq}$*, and reverse the extremal concretizations if* F *is decreasing. Of course, care must be taken to ensure that* F *is monotonic for* $\dot{\leq}$*. However, if it is not known whether* F *is increasing or decreasing, the gradual operator can be defined as*

$$\widetilde{\mathrm{F}}(\tau) = (\mathrm{F}(\tau^{\Downarrow}) \wedge \mathrm{F}(\tau^{\Uparrow})) \vee (((\mathrm{F}(\tau^{\Downarrow}) \vee \mathrm{F}(\tau^{\Uparrow})) \wedge ?)$$

In this case, the minimal materialization of the result is computed using $\mathrm{F}(\tau^{\Downarrow}) \wedge \mathrm{F}(\tau^{\Uparrow})$ *(which is equivalent to the smallest of the two types* $\mathrm{F}(\tau^{\Downarrow})$ *and* $\mathrm{F}(\tau^{\Uparrow})$*), and the maximal materialization is computed similarly using the union connective.*

6.2.3. Soundness of the gradual operators

The next step is to prove that these operators are sound, that is, they satisfy the minimality conditions presented in Section 2.4. We do not prove these properties from scratch. Instead, we prove that by definition, the gradual type operators inherit all the properties of the static operators we stated previously. For the domain operator, the minimality property states that for every function type τ, $\widetilde{\mathsf{dom}}(\tau)$ is the largest type σ such that $\tau \mathrel{\tilde{\leq}} \sigma \to \mathbb{1}$. Formally, we have the following proposition:

Proposition 6.17. *For every gradual types $\tau, \tau' \in$ GTypes, if $\tau \mathrel{\tilde{\leq}} \mathbb{0} \to \mathbb{1}$ then $\tau \mathrel{\tilde{\leq}} \widetilde{\mathrm{dom}}(\tau) \to \mathbb{1}$ and if $\tau \mathrel{\tilde{\leq}} \tau' \to \mathbb{1}$ then $\tau' \mathrel{\tilde{\leq}} \widetilde{\mathrm{dom}}(\tau)$.*

Proof. By hypothesis, $\tau \mathrel{\tilde{\leq}} \mathbb{0} \to \mathbb{1}$. By definition of $\mathrel{\tilde{\leq}}$, this entails $\tau^{\Downarrow} \mathrel{\dot{\leq}} \mathbb{0} \to \mathbb{1}$ and $\tau^{\Uparrow} \mathrel{\dot{\leq}} \mathbb{0} \to \mathbb{1}$. Moreover, by Corollary 6.9, we have $\tau^{\Downarrow} \mathrel{\dot{\leq}} \tau^{\Uparrow}$. By Proposition 6.12, this entails $\mathrm{dom}(\tau^{\Uparrow}) \mathrel{\dot{\leq}} \mathrm{dom}(\tau^{\Downarrow})$. This ensures that $(\widetilde{\mathrm{dom}}(\tau))^{\Uparrow} \simeq \mathrm{dom}(\tau^{\Downarrow})$ ① and $(\widetilde{\mathrm{dom}}(\tau))^{\Downarrow} \simeq \mathrm{dom}(\tau^{\Uparrow})$. Now, by Proposition 6.12, we have $\tau^{\Downarrow} \mathrel{\dot{\leq}} \mathrm{dom}(\tau^{\Downarrow}) \to \mathbb{1}$. By ①, we deduce $\tau^{\Downarrow} \mathrel{\dot{\leq}} (\widetilde{\mathrm{dom}}(\tau))^{\Uparrow} \to \mathbb{1}$ which gives $\tau^{\Downarrow} \mathrel{\dot{\leq}} (\widetilde{\mathrm{dom}}(\tau) \to \mathbb{1})^{\Downarrow}$. A similar reasoning proves $\tau^{\Uparrow} \mathrel{\dot{\leq}} (\widetilde{\mathrm{dom}}(\tau) \to \mathbb{1})^{\Uparrow}$, which yields $\tau \mathrel{\tilde{\leq}} \widetilde{\mathrm{dom}}(\tau) \to \mathbb{1}$.
Now suppose that $\tau \mathrel{\tilde{\leq}} \tau' \to \mathbb{1}$, by definition of $\mathrel{\tilde{\leq}}$ this yields $\tau^{\Downarrow} \mathrel{\dot{\leq}} \tau'^{\Uparrow} \to \mathbb{1}$. By Proposition 6.12, this implies $\tau'^{\Uparrow} \mathrel{\dot{\leq}} \mathrm{dom}(\tau^{\Downarrow})$. Using ① we deduce that $\tau'^{\Uparrow} \mathrel{\dot{\leq}} (\widetilde{\mathrm{dom}}(\tau))^{\Uparrow}$. A similar reasoning proves $\tau'^{\Downarrow} \mathrel{\dot{\leq}} (\widetilde{\mathrm{dom}}(\tau))^{\Downarrow}$, which ensures $\tau' \mathrel{\tilde{\leq}} \widetilde{\mathrm{dom}}(\tau)$. □

For the result type operator, we prove that $\tau \mathbin{\tilde{\circ}} \sigma$ is the *smallest* type τ' such that $\tau \mathrel{\tilde{\leq}} \sigma \to \tau'$, provided τ is a function type that can be applied to σ. Since the proof of this result follows the same idea as the proof of Proposition 6.17, we refer the interested reader to the appendix.

Proposition 6.18. *For every gradual types $\tau, \tau', \sigma \in$ GTypes, if $\tau \mathrel{\tilde{\leq}} \mathbb{0} \to \mathbb{1}$ and $\sigma \mathrel{\tilde{\leq}} \widetilde{\mathrm{dom}}(\tau)$ then $\tau \mathrel{\tilde{\leq}} \sigma \to (\tau \mathbin{\tilde{\circ}} \sigma)$. Moreover, if $\tau \mathrel{\tilde{\leq}} \sigma \to \tau'$ then $\tau \mathbin{\tilde{\circ}} \sigma \mathrel{\tilde{\leq}} \tau'$.*

Proof. See appendix page 264. □

Finally, for the projection type operator, we prove that $\widetilde{\pi_1}(\tau)$ and $\widetilde{\pi_2}(\tau)$ are respectively the smallest types τ_1 and τ_2 such that $\tau \mathrel{\tilde{\leq}} \tau_1 \times \tau_2$, provided τ is a pair type.

Proposition 6.19. *For every gradual types $\tau, \tau_1, \tau_2 \in$ GTypes, if $\tau \mathrel{\tilde{\leq}} \mathbb{1} \times \mathbb{1}$ then $\tau \mathrel{\tilde{\leq}} \widetilde{\pi_1}(\tau) \times \widetilde{\pi_2}(\tau)$ and if $\tau \mathrel{\tilde{\leq}} \tau_1 \times \tau_2$ then $\widetilde{\pi_1}(\tau) \mathrel{\tilde{\leq}} \tau_1$ and $\widetilde{\pi_2}(\tau) \mathrel{\tilde{\leq}} \tau_2$.*

Proof. See appendix page 265. □

6.2.4. Preservation of semantic precision

Finally, soundness is not the only important property of these operators. We also prove that they preserve precision, thus ensuring that applying an operator to the two types of a cast preserves typability, as in, for example, a reduction rule of the form:

$$\pi_i\,(V\langle \tau \Rightarrow_p \tau' \rangle) \rightsquigarrow (\pi_i\,V)\langle \widetilde{\pi_i}(\tau) \Rightarrow_p \widetilde{\pi_i}(\tau') \rangle$$

The proof of these results is straightforward, thanks to the definition of semantic precision and Proposition 6.3.

Proposition 6.20. *For every gradual types $\tau, \tau' \in$ GTypes such that $\tau \mathrel{\tilde{\leq}} \mathbb{0} \to \mathbb{1}$ and $\tau' \mathrel{\tilde{\leq}} \mathbb{0} \to \mathbb{1}$, if $\tau \mathrel{\tilde{\leqslant}} \tau'$ then $\widetilde{\mathrm{dom}}(\tau) \mathrel{\tilde{\leqslant}} \widetilde{\mathrm{dom}}(\tau')$.*

Proof. By Proposition 6.3, we have $\tau^{\Downarrow} \dot{\leq} \tau'^{\Downarrow}$. And by definition of $\widetilde{\leq}$, we have $\tau^{\Downarrow} \dot{\leq} \mathbb{0} \to \mathbb{1}$ and $\tau'^{\Downarrow} \dot{\leq} \mathbb{0} \to \mathbb{1}$. By Proposition 6.12, this entails $\mathsf{dom}(\tau'^{\Downarrow}) \dot{\leq} \mathsf{dom}(\tau^{\Downarrow})$. Which yields $(\widetilde{\mathsf{dom}}(\tau'))^{\Uparrow} \dot{\leq} (\widetilde{\mathsf{dom}}(\tau))^{\Uparrow}$. A similar reasoning on $\tau^{\Uparrow}$ and $\tau'^{\Uparrow}$ yields $(\widetilde{\mathsf{dom}}(\tau))^{\Downarrow} \dot{\leq} (\widetilde{\mathsf{dom}}(\tau'))^{\Downarrow}$, hence $\widetilde{\mathsf{dom}}(\tau) \mathrel{\widetilde{\geq}} \widetilde{\mathsf{dom}}(\tau')$. □

Proposition 6.21. *For every gradual types $\tau, \tau', \sigma \in \mathsf{GTypes}$ such that $\tau \mathrel{\widetilde{\leq}} \mathbb{0} \to \mathbb{1}$, $\tau' \mathrel{\widetilde{\leq}} \mathbb{0} \to \mathbb{1}$, $\sigma \mathrel{\widetilde{\leq}} \widetilde{\mathsf{dom}}(\tau)$, and $\sigma \mathrel{\widetilde{\leq}} \widetilde{\mathsf{dom}}(\tau')$, if $\tau \mathrel{\widetilde{\geq}} \tau'$ then $\tau \mathbin{\tilde{\circ}} \sigma \mathrel{\widetilde{\geq}} \tau' \mathbin{\tilde{\circ}} \sigma$.*

Proof. See appendix page 265. □

Proposition 6.22. *For every gradual types $\tau, \tau' \in \mathsf{GTypes}$ such that $\tau \mathrel{\widetilde{\leq}} \mathbb{1} \times \mathbb{1}$ and $\tau' \mathrel{\widetilde{\leq}} \mathbb{1} \times \mathbb{1}$, if $\tau \mathrel{\widetilde{\geq}} \tau'$ then for every $i \in \{1, 2\}$, $\widetilde{\pi_i}(\tau) \mathrel{\widetilde{\geq}} \widetilde{\pi_i}(\tau')$.*

Proof. See appendix page 265. □

6.3. Operational semantics

We can now present our new, enhanced operational semantics for the calculus presented in Section 5.3. The terms and evaluation contexts are unchanged. The type system is also unchanged, except for the three rules involving precision and subtyping, which now use their semantic versions, and a slight change to the typing rule for constants, which we introduce and explain below.

6.3.1. Semantic ground types

The central idea behind this semantics is simple: rather than using intermediate types (or ground types) to store information about type constructors, we instead use the intersection connective. Consider a cast function such as $V\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow_p ? \rangle$. The strategy we presented in Chapter 4 and we later adapted in Section 5.3 consisted in splitting the cast by inserting $? \to ?$ as an intermediate ground type: $V\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow_p ? \to ? \rangle\langle ? \to ? \Rightarrow_p ? \rangle$. Then, depending on whether this expression is later cast to a compatible or to an incompatible ground type, we either eliminated the casts we introduced or failed. In essence, ground types were used to check the compatibility between two type constructors.

The intersection connective can actually perform similar operations in a much simpler way. To start with, note that every uncast value can either be given type $\mathbb{1} \times \mathbb{1}$, or $\mathbb{0} \to \mathbb{1}$, or a base type b (a value cannot be both a pair and a function, for example), which we call its *constructor type*. Now, if τ_1 and τ_2 are two constructor types, then their compatibility can be checked easily by verifying that $\tau_1 \wedge \tau_2 \not\mathrel{\widetilde{\geq}} \mathbb{0}$.

This leaves us with a problem: ensuring that the information about the constructor of a value is propagated correctly in casts, especially since casts can introduce unions. Otherwise, a sequence such as $V\langle (\mathsf{Int} \to \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool}) \Rightarrow_p ? \rangle\langle ? \Rightarrow_q (\mathsf{Bool} \times \mathsf{Bool}) \rangle$ would always be considered fine since $((\mathsf{Int} \to \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool})) \wedge (\mathsf{Bool} \times \mathsf{Bool})$ is not empty, and thus the two casts

are compatible. However, if V is a function, it is clear that this expression should fail, since it is trying to cast a function to a pair.

The solution, once again, is to use the intersection connective. When reducing the above expression, we can perform a syntactic lookup on V to deduce its constructor type: if V is a λ-abstraction then its constructor type is $\mathbb{0} \to \mathbb{1}$, if V is a pair then its constructor type is $\mathbb{1} \times \mathbb{1}$, and if V is a constant then its constructor type is simply its (basic) type. Then, we can intersect all the types in the casts following the value with this top type. Since intersecting a type with another always produces a smaller type, this operation preserves the type of the expression. In the case of the expression above, we obtain:

$$V\langle((\mathsf{Int} \to \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool})) \wedge (\mathbb{0} \to \mathbb{1}) \Rightarrow_p \mathsf{?} \wedge (\mathbb{0} \to \mathbb{1})\rangle\langle \mathsf{?} \wedge (\mathbb{0} \to \mathbb{1}) \Rightarrow_q (\mathsf{Bool} \times \mathsf{Bool}) \wedge (\mathbb{0} \to \mathbb{1})\rangle$$

or equivalently:

$$V\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow_p \mathsf{?} \wedge (\mathbb{0} \to \mathbb{1})\rangle\langle \mathsf{?} \wedge (\mathbb{0} \to \mathbb{1}) \Rightarrow_q \mathbb{0}\rangle$$

We then notice that we are casting a value to $\mathbb{0}$, which should obviously fail as no value has type $\mathbb{0}$, and we can blame the second cast.

With this in mind, the first major change comes from the definition of values, and more precisely, cast values.

Definition 6.23 (Values of the cast language)**.** *We define the set* $\mathsf{Values}^{\langle \mathsf{ST} \rangle}$ *of values, ranged over by* V*, using the following grammar:*

$$\mathsf{Values}^{\langle \mathsf{ST} \rangle} \ni V ::= c \mid \lambda^{\tau \to \tau} x.\, E \mid (V, V) \mid \Lambda \vec{\alpha}.\, E \mid V\langle \tau \Rightarrow_p \tau \rangle$$

where additionally, values of the form $V\langle \tau \Rightarrow_p \tau' \rangle$ *verify* $\tau' \not\cong \mathbb{0}$ *and either* $\tau \vee \tau' \cong \mathbb{0} \to \mathbb{1}$ *or* $\tau \vee \tau' \cong \mathbb{1} \times \mathbb{1}$*.*

Since we now propagate the information about the constructor type of a value in the casts that follow it, every value of the form $V\langle \tau \Rightarrow_p \tau' \rangle$ is such that both τ and τ' are subtypes of the same constructor type (which is equivalent to saying that $\tau \vee \tau'$ is a subtype of a constructor type). Additionally, we forbid τ' to be empty since no values should be of type $\mathbb{0}$, and in this case the cast should simply fail, blaming label p.

Note that we only consider $\mathbb{0} \to \mathbb{1}$ and $\mathbb{1} \times \mathbb{1}$ as constructor types in this definition: we do not include base types. This is because a cast on a constant can always be reduced, there is no need to wait until a later cast, an application, or a projection to determine whether it should succeed or not. When reducing a cast constant $c\langle \tau_1 \Rightarrow_p \tau_2 \rangle$ we can simply eliminate the cast if $b_c \wedge \tau_2$ is not empty, and blame p otherwise.

As a side effect, this means we eliminate casts such as $3\langle \mathsf{Int} \Rightarrow_p \mathsf{?} \rangle$: here, the expression reduces to 3, since $b_3 \wedge \mathsf{?} \not\cong \mathbb{0}$. While not customary in the gradual typing literature, this operation is sound, as such a cast will never fail. It is usually only kept for the purpose of type preservation, since $3\langle \mathsf{Int} \Rightarrow_p \mathsf{?} \rangle$ has type $\mathsf{?}$ but 3 has type Int (or, in our system, the singleton type b_3), which is not a subtype of $\mathsf{?}$. However, since casting a constant (or any value) to $\mathsf{?}$ is always a safe operation, we can simply consider that a constant c is always of type $b_c \wedge \mathsf{?}$ by replacing the rule $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cst}}]$ presented in Figure 4.5 with the following typing rule:

$$[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cst}}] \ \frac{}{\Gamma \vdash c : b_c \wedge \mathsf{?}}$$

Using this typing rule, 3 now has type $b_3 \wedge ?$, which is a subtype of $?$. As such, the above reduction correctly preserves the type of the expression.

6.3.2. Value operators

As anticipated, the semantics relies on several additional operators, which collect information about the type and structure of a value.

The first operator we define is the *constructor operator* which associates to every value its constructor type, following the intuition we gave previously. This operator will serve to propagate the constructor type of a value along the casts that follow it.

Definition 6.24 (Constructor operator)**.** *We define the operator* $\mathsf{cons}(.) : \mathsf{Values}^{\langle \mathsf{ST} \rangle} \to \mathsf{GTypes}$ *as follows:*

$$\begin{aligned} \mathsf{cons}(c) &= b_c \\ \mathsf{cons}(\lambda^{\tau_1 \to \tau_2} x.\, E) &= \mathbb{0} \to \mathbb{1} \\ \mathsf{cons}((V_1, V_2)) &= \mathbb{1} \times \mathbb{1} \\ \mathsf{cons}(V\langle \tau_1 \Rightarrow_p \tau_2 \rangle) &= \mathsf{cons}(V) \\ \mathsf{cons}(\Lambda \vec{\alpha}.\, E) &= \textit{undefined} \end{aligned}$$

Note that the constructor operator is not defined for type abstractions. The reason for this is simple: our semantics requires all type abstractions to be explicitly instantiated before being cast. In particular, the subtyping relation only acts on types (not type schemes), and as such, an expression of the form $(\Lambda \vec{\alpha}.\, E)\langle \tau_1 \Rightarrow_p \tau_2 \rangle$ is never well typed since $\Lambda \vec{\alpha}.\, E$ cannot have type τ_1.

The second operator we need is the *value type operator*, which performs a syntactic lookup on a value to determine its most precise type. That is, the type of a λ-abstraction is simply its annotation, the type of a pair of values is the product of the types of its components, the type of a constant c is $b_c \wedge ?$ as explained above, and the type of a cast value is the target type of the cast.

Definition 6.25 (Value type operator)**.** *We define the operator* $\mathsf{type}(.) : \mathsf{Values}^{\langle \mathsf{ST} \rangle} \to \mathsf{GTypes}$ *as follows:*

$$\begin{aligned} \mathsf{type}(c) &= b_c \wedge ? \\ \mathsf{type}(\lambda^{\tau_1 \to \tau_2} x.\, E) &= \tau_1 \to \tau_2 \\ \mathsf{type}((V_1, V_2)) &= \mathsf{type}(V_1) \times \mathsf{type}(V_2) \\ \mathsf{type}(V\langle \tau_1 \Rightarrow_p \tau_2 \rangle) &= \tau_2 \\ \mathsf{type}(\Lambda \vec{\alpha}.\, E) &= \textit{undefined} \end{aligned}$$

The value type operator is needed for the same reasons as in Section 5.3 when reducing a cast application. If a function is cast to $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$ and is applied to a value of type Int, then the result must be cast to Int (which is the expected type of the application), instead of $\mathsf{Int} \vee \mathsf{Bool}$ (which is the codomain of the type of the function), as the latter would be unsound.

As an example, consider the function $\lambda^{? \to ?} x.\, (\mathsf{true}\langle \mathsf{Bool} \Rightarrow_p ? \rangle)$ followed by the cast $\langle ? \to ? \Rightarrow_q (\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool}) \rangle$ and then applied to 3. If the result were cast to $\mathsf{Int} \vee \mathsf{Bool}$, we would obtain $\mathsf{true}\langle \mathsf{Bool} \Rightarrow_p ? \rangle \langle ? \Rightarrow_q \mathsf{Int} \vee \mathsf{Bool} \rangle$, which reduces to true. However, since we can statically deduce that the application is of type Int, this would be unsound. The solution,

$$
\begin{array}{rrcll}
[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cons}}] & V\langle \tau \Rightarrow_p \tau' \rangle & \rightsquigarrow & V\langle \tau \wedge \mathsf{cons}(V) \Rightarrow_p \tau' \wedge \mathsf{cons}(V) \rangle & \text{if } \tau \vee \tau' \not\cong \mathsf{cons}(V) \\
[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Blame}}] & V\langle \tau \Rightarrow_p \tau' \rangle & \rightsquigarrow & \mathsf{blame}\ p & \text{if } \tau' \cong \mathbb{0} \\
[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Simpl}}] & c\langle \tau \Rightarrow_p \tau' \rangle & \rightsquigarrow & c & \text{if } b_c \wedge ? \cong \tau' \\
[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{App}}] & (\lambda^{\tau \to \tau'} x.\, E)\, V & \rightsquigarrow & E\,[V/x] & \\
[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Proj}}] & \pi_i\,(V_1, V_2) & \rightsquigarrow & V_i & \text{for } i \in \{1,2\} \\
[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CApp}}] & (V\langle \tau \Rightarrow_p \tau' \rangle)\, V' & \rightsquigarrow & (V\,(V'\langle \widetilde{\mathsf{dom}}(\tau') \wedge \sigma \Rightarrow_{\overline{p}} \widetilde{\mathsf{dom}}(\tau) \wedge \sigma \rangle))\langle \tau \,\tilde{\circ}\, (\sigma \wedge \widetilde{\mathsf{dom}}(\tau)) \Rightarrow_p \tau' \,\tilde{\circ}\, (\sigma \wedge \widetilde{\mathsf{dom}}(\tau)) \rangle & \\
 & & & & \text{where } \sigma = \mathsf{type}(V') \\
[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CProj}}] & \pi_i\,(V\langle \tau \Rightarrow_p \tau' \rangle) & \rightsquigarrow & (\pi_i\, V)\langle \widetilde{\pi}_i(\tau) \Rightarrow_p \widetilde{\pi}_i(\tau') \rangle & \text{for } i \in \{1,2\} \\
[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{TApp}}] & (\Lambda \vec{\alpha}.\, E)[\vec{t}] & \rightsquigarrow & E\,[\vec{t}/\vec{\alpha}] & \\
[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Let}}] & \mathsf{let}\ x = V\ \mathsf{in}\ E & \rightsquigarrow & E\,[V/x] & \\
[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}] & \mathscr{E}\,[E] & \rightsquigarrow & \mathscr{E}\,[E'] & \text{if } E \rightsquigarrow E' \\
[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CtxBlame}}] & \mathscr{E}\,[E] & \rightsquigarrow & \mathsf{blame}\ p & \text{if } E \rightsquigarrow \mathsf{blame}\ p
\end{array}
$$

FIGURE 6.1. Operational semantics of the cast calculus

when reducing a cast application, is to lookup the type of the argument using type(.), and use this information along with the result type operator $\tilde{\circ}$ to deduce the expected type of the result. On the above example, we would compute $((\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})) \,\tilde{\circ}\, \mathsf{Int}$, which evaluates to Int, and the application would then reduce to $\mathsf{true}\langle \mathsf{Bool} \Rightarrow_p ? \rangle\langle ? \Rightarrow_q \mathsf{Int} \rangle$, which would properly fail (blaming label q).

6.3.3. Operational semantics

The full operational semantics of the cast calculus are presented in Figure 6.1, and most of the rules have already been explained. Rules $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{App}}]$, $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Proj}}]$, $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{TApp}}]$, $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Let}}]$, $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}]$, and $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CtxBlame}}]$ are unchanged from Chapter 5 as they do not involve casts. Rule $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cons}}]$ propagates the information about the constructor type of a value if needed. Rule $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Blame}}]$ fails if a value is cast to an empty type. Together with $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cons}}]$, this ensures that if a value is cast to an incompatible constructor type, then the cast fails. Rule $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Simpl}}]$ performs the simplification of cast constants we discussed earlier. Rule $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CProj}}]$ reduces the projection of a cast value by simply applying the projection type operator to both components of the cast.

Rule $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CApp}}]$ is, however, more complicated than anticipated. Similarly to $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CProj}}]$, and following our previous explanation, one could have expected the rule to look like:

$$(V\langle \tau \Rightarrow_p \tau' \rangle)\, V' \rightsquigarrow (V\,(V'\langle \widetilde{\mathsf{dom}}(\tau') \Rightarrow_{\overline{p}} \widetilde{\mathsf{dom}}(\tau) \rangle))\langle \tau \,\tilde{\circ}\, \sigma \Rightarrow_p \tau' \,\tilde{\circ}\, \sigma \rangle$$

where $\sigma = \mathsf{type}(V')$. That is, we could have simply applied the domain and result type operators to both components of the cast. However, this does not preserve typability. Consider for example the expression $V\langle (? \to ?) \to ? \Rightarrow_p (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int} \rangle\, V'$ where $\mathsf{type}(V') = \mathsf{Int} \to \mathsf{Int}$. In this expression, we cannot compute $((? \to ?) \to ?) \,\tilde{\circ}\, \mathsf{type}(V')$ because $\mathsf{Int} \to \mathsf{Int}$ is not a subtype of $? \to ?$. Therefore, we intersect the type of the argument with the domain of the function to ensure we can compute the type of the result, which gives a cast of the form $\langle \tau \,\tilde{\circ}\, (\sigma \wedge \widetilde{\mathsf{dom}}(\tau)) \Rightarrow_p \tau' \,\tilde{\circ}\, (\sigma \wedge \widetilde{\mathsf{dom}}(\tau)) \rangle$. However, the cast can only be well-typed if it is applied to an expression of its source type, that is, $\tau \,\tilde{\circ}\, (\sigma \wedge \widetilde{\mathsf{dom}}(\tau))$. Thus, we also add an intersection with σ on the first cast, which ensures that the argument can be statically given type $\widetilde{\mathsf{dom}}(\tau) \wedge \sigma$

and thus the application $V\ (V'\langle\widetilde{\mathsf{dom}}(\tau') \wedge \sigma \Rightarrow_{\overline{p}} \widetilde{\mathsf{dom}}(\tau) \wedge \sigma\rangle)$ can be given type $\tau \mathbin{\tilde{\circ}} (\sigma \wedge \widetilde{\mathsf{dom}}(\tau))$.

While the rule for cast applications is more complicated than anticipated, it must be emphasized how the reduction rules are *much* simpler than the rules presented in Section 5.3. In particular, operators have straightforward definitions, and the use of the intersection connective in $[\mathrm{R}^{\langle\mathsf{ST}\rangle}_{\mathsf{Cons}}]$ removes the need for relative ground types, which is arguably the most complex notion introduced in Chapter 5.

6.3.4. Properties

The semantics verifies the same soundness properties stated in Section 5.3, namely progress, subject reduction, and blame safety. As before, the latter holds as an immediate corollary of progress. In this subsection, we provide some insight into the proof of these properties and detail the required lemmas.

The first lemma concerns the soundness of the operator $\mathsf{type}(.)$: we prove that a value V can indeed be statically given type $\mathsf{type}(V)$ and, moreover, that $\mathsf{type}(V)$ is the minimal type that can be derived by the type system for V.

Lemma 6.26. *For every value* $V \in \mathsf{Values}^{\langle\mathsf{ST}\rangle}$ *and every type environment* Γ*, if* $\Gamma \vdash V : \tau$ *then* $\Gamma \vdash V : \mathsf{type}(V)$ *and* $\mathsf{type}(V) \mathrel{\tilde{\leqslant}} \tau$.

Proof. See appendix page 265.
The proof is done by a straightforward induction over the derivation $\Gamma \vdash V : \tau$ and case analysis over V. □

We then obtain two corollaries of this lemma, which are crucial to the soundness of our semantics. The first result concerns the soundness of $\mathsf{cons}(.)$, which is similar to the soundness of $\mathsf{type}(.)$ except that the type $\mathsf{cons}(V)$ is not minimal.

Lemma 6.27. *For every value* $V \in \mathsf{Values}^{\langle\mathsf{ST}\rangle}$ *and every type environment* Γ*, if* $\Gamma \vdash V : \tau$ *then* $\Gamma \vdash V : \mathsf{cons}(V)$.

Proof. Immediate consequence of Lemma 6.26 since for every value V, $\mathsf{type}(V) \mathrel{\tilde{\leqslant}} \mathsf{cons}(V)$. □

The second result proves that no value has type $\mathbb{0}$. This is a critical aspect of semantic subtyping, and care must be taken to ensure that this holds even in the presence of cast values.

Lemma 6.28. *For every value* $V \in \mathsf{Values}^{\langle\mathsf{ST}\rangle}$ *and every type environment* Γ*, if* $\Gamma \vdash V : \tau$ *then* $\tau \not\mathrel{\tilde{\leqslant}} \mathbb{0}$.

Proof. By case analysis, we immediately have that $\mathsf{type}(V) \not\mathrel{\tilde{\leqslant}} \mathbb{0}$, and we conclude by Lemma 6.26. □

The next result is fundamental to prove the blame safety of our semantics, and is a major property of our precision relation. It states that if a type is non-empty, then making it less precise (by replacing some of its subterms with ? for example) cannot make it empty. In essence, this proves that upcasting a value (i.e., casting it to a less precise type) cannot fail. Together with the correlation between the polarity of blame labels and the direction of casts, this entails the blame safety of our semantics.

Lemma 6.29. *For every types* $\tau_1, \tau_2 \in \mathsf{GTypes}$*, if* $\tau_1 \mathrel{\tilde{\leqslant}} \tau_2$ *and* $\tau_2 \not\mathrel{\tilde{\leqslant}} \mathbb{0}$ *then* $\tau_1 \not\mathrel{\tilde{\leqslant}} \mathbb{0}$.

Proof. By Corollary 6.9, we have $\tau_1^{\Downarrow} \dot{\leq} \tau_1^{\Uparrow}$. By definition of $\tilde{\leq}$, this entails $\tau_1 \not\tilde{\leq} \mathbb{0} \iff \tau_1^{\Uparrow} \not\dot{\leq} \mathbb{0}$. By Proposition 6.3, we have $\tau_2^{\Uparrow} \dot{\leq} \tau_1^{\Uparrow}$. And since $\tau_2 \not\tilde{\leq} \mathbb{0}$, then $\tau_2^{\Uparrow} \not\dot{\leq} \mathbb{0}$. Therefore, necessarily $\tau_1^{\Uparrow} \not\dot{\leq} \mathbb{0}$ and $\tau_1 \not\tilde{\leq} \mathbb{0}$. □

The next result proves the soundness of our strategy of eliminating casts on constants. It states that for every constant c and every type τ, then either c is of type τ or c is of type $\neg\tau$. This is not the case for functions, for example, since a function of type Bool → Bool is neither of type Int → Int not ¬(Int → Int), as both intersection (Bool → Bool) ∧ (Int → Int) and (Bool → Bool) ∧ ¬(Int → Int) are non-empty.

Lemma 6.30. *For every type $\tau \in \mathsf{GTypes}$, and every constant $c \in \mathscr{C}$, if $b_c \wedge \tau \not\tilde{\leq} \mathbb{0}$ then $b_c \wedge ? \tilde{\leq} \tau$.*

Proof. Since $(b_c \wedge ?)^{\Downarrow} = \mathbb{0}$, we immediately have $(b_c \wedge ?)^{\Downarrow} \dot{\leq} \tau^{\Downarrow}$.
Moreover, $(b_c \wedge ?)^{\Uparrow} = b_c$. By definition of $\tilde{\leq}$, we have $(b_c \wedge \tau)^{\Uparrow} \not\dot{\leq} \mathbb{0}$, which entails $b_c \wedge \tau^{\Uparrow} \not\dot{\leq} \mathbb{0}$. Since $[\![b_c]\!]$ is a singleton, we have either $b_c \dot{\leq} \tau^{\Uparrow}$ or $b_c \wedge \tau^{\Uparrow} = \mathbb{0}$. Since the latter does not hold, we have $b_c \dot{\leq} \tau^{\Uparrow}$, hence the result. □

Following this lemma, we prove a straightforward result which states that no value is a cast constant. This will be important when performing a case disjunction on an expression such as $V\langle \tau_1 \Rightarrow_p \tau_2 \rangle$: if τ_1 is a base type, then necessarily V is a constant.

Lemma 6.31. *For every value $V \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ such that $\emptyset \vdash V : \tau$ and $\mathsf{cons}(V) \in \mathscr{B}$, we have $V \in \mathscr{C}$.*

Proof. By induction on V.

- $\underline{V = c.}$ Immediate.
- $\overline{V = V'\langle \tau' \Rightarrow_p \tau'' \rangle.}$, where $\tau' \vee \tau'' \tilde{\leq} \mathbb{0} \to \mathbb{1}$ or $\tau' \vee \tau'' \tilde{\leq} \mathbb{1} \times \mathbb{1}$. By definition of cons(.), we have $\mathsf{cons}(V) = \mathsf{cons}(V') \in \mathscr{B}$. By IH, there exists $c \in \mathscr{C}$ such that $V' = c$. By inversion of the typing rules, we deduce that $b_c \wedge ? \tilde{\leq} \tau'$. But since $b_c \wedge ? \not\tilde{\leq} (\mathbb{1} \times \mathbb{1}) \vee (\mathbb{0} \to \mathbb{1})$, we have a contradiction.
- The other cases cannot satisfy $\mathsf{cons}(V) \in \mathscr{B}$.

□

We can now state and prove the progress property for our semantics. As in Section 5.3, it states that a well-typed expression is either a value or can be reduced, and if it reduces to a blame, then this blame is necessarily positive.

Lemma 6.32 (Progress)**.** *For every term $E \in \mathsf{Terms}^{\langle \mathsf{ST} \rangle}$, if $\emptyset \vdash E : \forall \vec{\alpha}.\tau$ then one of the following holds:*

- *there exists $E' \in \mathsf{Terms}^{\langle \mathsf{ST} \rangle}$ such that $E \rightsquigarrow E'$;*
- *there exists $\ell \in \mathscr{L}$ such that $E \rightsquigarrow \mathsf{blame}\ \ell$;*
- *$E \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$.*

Proof. See appendix page 266.
The proof is done by induction on the derivation $\emptyset \vdash E : \forall\vec{\alpha}.\tau$ and case analysis over the last rule used. Most cases are straightforward, the most interesting being the cases of $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cast}+}]$ and $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cast}-}]$ when $E = V\langle \tau_1 \Rightarrow_p \tau_2 \rangle$. If p is negative, that is, $\tau_2 \precsim \tau_1$ by rule $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cast}-}]$, then Lemma 6.28 ensures that $\tau_1 \not\precsim \mathbb{0}$ since V cannot have type $\mathbb{0}$, and Lemma 6.29 proves that necessarily $\tau_2 \not\precsim \mathbb{0}$. Thus, p cannot be blamed. □

We continue with the lemmas required to prove the subject reduction for our semantics. As customary, we need three substitution lemmas: one for variables, one for contexts, and one for type substitutions. They are proven by straightforward inductions on expressions, contexts, and type derivations respectively.

Lemma 6.33. *If* $\Gamma, x : \forall\vec{\alpha}'.\tau' \vdash E : \forall\vec{\alpha}.\tau$*, then for every expression* E' *such that* $\Gamma \vdash E' : \forall\vec{\alpha}'.\tau'$*, we have* $\Gamma \vdash E\,[E'/x] : \forall\vec{\alpha}.\tau$*.*

Proof. See appendix page 267. □

Lemma 6.34. *If* $\Gamma \vdash \mathscr{E}\,[E] : \forall\vec{\alpha}.\tau$*, then* $\Gamma \vdash E : \forall\vec{\alpha}'.\tau'$ *and for every expression* E' *such that* $\Gamma \vdash E' : \forall\vec{\alpha}'.\tau'$*, we have* $\Gamma \vdash \mathscr{E}\,[E'] : \forall\vec{\alpha}.\tau$*.*

Proof. See appendix page 268. □

Lemma 6.35. *If* $\Gamma \vdash E : \forall\vec{\alpha}.\tau$*, then for every static type substitution* $\theta : \mathscr{V}^{\alpha} \to \mathsf{GTypes}$ *such that* $\mathrm{dom}(\theta) \cap \vec{\alpha} = \emptyset$*,* $\Gamma\theta \vdash E\theta : \forall\vec{\alpha}.\tau\theta$*.*

Proof. See appendix page 269. □

Finally, we prove the subject reduction of our semantics. As customary, this states that the type of an expression is preserved by reduction.

Lemma 6.36 (Subject reduction)**.** *For every term* $E, E' \in \mathsf{Terms}^{\langle \mathsf{ST} \rangle}$*, if* $\Gamma \vdash E : \forall\vec{\alpha}.\tau$ *and* $E \rightsquigarrow E'$ *then* $\Gamma \vdash E' : \forall\vec{\alpha}.\tau$*.*

Proof. See appendix page 270.
The proof is done by induction on E and case analysis on the reduction rule used for $E \rightsquigarrow E'$. Lemma 6.33 proves the result for $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{App}}]$, Lemma 6.35 proves $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{TApp}}]$, and the case of $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}]$ follows from Lemma 6.34.
The most interesting and difficult case, which we detail here, is the case of rule $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CApp}}]$, which invokes the soundness lemmas for the various type operators. The case of $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CProj}}]$ is similar (and simpler), and the remaining cases are straightforward.
For rule $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Capp}}]$, we have $(V\langle \tau_1 \Rightarrow_p \tau_2 \rangle)\,V' \rightsquigarrow (V\,(V'\langle \widetilde{\mathrm{dom}}(\tau_2) \wedge \sigma \Rightarrow_{\overline{p}} \widetilde{\mathrm{dom}}(\tau_1) \wedge \sigma \rangle))\langle \tau_1 \mathbin{\tilde{\circ}} (\sigma \wedge \widetilde{\mathrm{dom}}(\tau_1)) \Rightarrow_p \tau_2 \mathbin{\tilde{\circ}} (\sigma \wedge \widetilde{\mathrm{dom}}(\tau_1)) \rangle$ where $\sigma = \mathrm{type}(V')$. Suppose that p is positive. The negative case is proven similarly. By inversion of the typing rules, we have that there exists σ' and τ' such that:

$$\textcircled{1}\quad \emptyset \vdash V : \tau_1 \qquad \textcircled{2}\quad \tau_1 \precsim \tau_2 \qquad \textcircled{3}\quad \emptyset \vdash V' : \sigma'$$

$$\textcircled{4}\quad \tau_2 \precsim \sigma' \to \tau' \qquad \textcircled{5}\quad \tau' \precsim \forall\vec{\alpha}.\tau$$

By Lemma 6.26 and ③, we deduce that $\sigma \cong \sigma'$. By Proposition 6.17 and ④, we deduce $\sigma' \cong \widetilde{\mathsf{dom}}(\tau_2)$, hence $\sigma \cong \widetilde{\mathsf{dom}}(\tau_2)$. Thus, by Lemma 6.26, we have $\emptyset \vdash V' : \widetilde{\mathsf{dom}}(\tau_2) \wedge \sigma$ Ⓐ.
Proposition 6.20 and ② ensure $\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma \lessapprox \widetilde{\mathsf{dom}}(\tau_2) \wedge \sigma$. Together with Ⓐ, this proves $\emptyset \vdash V'\langle \widetilde{\mathsf{dom}}(\tau_2) \wedge \sigma \Rightarrow_{\overline{p}} \widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma\rangle : \widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma$ Ⓑ.
By Proposition 6.18, we have $\tau_1 \cong (\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma) \to \tau_1 \mathbin{\tilde{\circ}} (\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma)$. By $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]$ and ①, along with $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{App}}]$ and Ⓑ, we deduce that $\emptyset \vdash V\,(V'\langle \widetilde{\mathsf{dom}}(\tau_2) \wedge \sigma \Rightarrow_{\overline{p}} \widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma\rangle) : \tau_1 \mathbin{\tilde{\circ}} (\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma)$ Ⓒ.
By Proposition 6.21, we deduce that $\tau_1 \mathbin{\tilde{\circ}} (\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma) \lessapprox \tau_2 \mathbin{\tilde{\circ}} (\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma)$. Along with Ⓒ, this entails $\emptyset : E' : \tau_2 \mathbin{\tilde{\circ}} (\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma)$. Finally, ④ and Proposition 6.18 prove $\tau_2 \mathbin{\tilde{\circ}} (\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma) \cong \tau'$ and the result follows from ⑤ and $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]$. □

To conclude this section, our semantics enjoys both the soundness and blame safety properties stated in Chapter 4, as corollaries of the progress and subject reduction properties.

Theorem 6.37 (Soundness)**.** *For every term $E \in \mathsf{Terms}^{\langle \mathsf{ST} \rangle}$, if $\emptyset \vdash E : \forall \vec{\alpha}.\tau$ then one of the following holds:*

- *there exists $V \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ such that $E \rightsquigarrow^* V$ and $\vdash V : \forall \vec{\alpha}.\tau$;*
- *there exists $\ell \in \mathscr{L}$ such that $E \rightsquigarrow^* \mathsf{blame}\ \ell$;*
- *E diverges.*

Proof. Immediate consequence of Lemma 6.36 and Lemma 6.32. □

Corollary 6.38 (Blame safety)**.** *For every term $E \in \mathsf{Terms}^{\langle \mathsf{HM} \rangle}$ and every blame label $\ell \in \mathscr{L}$, if $\emptyset \vdash E : \forall \vec{\alpha}.\tau$ then $E \not\rightsquigarrow^* \mathsf{blame}\ \overline{\ell}$.*

Proof. Immediate consequence of Theorem 6.37. □

6.4. Summary

In this chapter, we considerably improved and simplified the semantics presented in Section 5.3, and introduced many powerful and interesting results. We summarize these contributions here, before concluding the first part of this manuscript.

Semantic relations. First of all, we introduced two new relations, *semantic gradual subtyping* and *semantic precision*, which are, without doubt, the most important definitions of this chapter. What differentiates them from the relations presented in the previous chapters is their *semantic* nature: they are defined so that they both induce the same equivalence relation. In particular, this ensures that two equivalent types (for subtyping) have the same materializations, which is a crucial property that was missing from our definition of *precision* in Chapter 5.

Efficient representation of gradual types. Moreover, we have shown in Chapter 5 (see Theorem 5.25) that, in the presence of top and bottom types (set-theoretic connectives are not necessary for this result), gradual types have extremal materializations. It turns out that, for the new

semantic relations, every gradual type can be equated to its extremal materializations, without any loss of information. This proves that these relations can be easily decided provided static subtyping also is, since they reduce in linear time to static subtyping (see Definition 6.5 and Proposition 6.3). Even more importantly, and perhaps surprisingly, given the extremal materializations of a gradual type, we can reconstruct it (or an equivalent type) using set-theoretic connectives and *a single* occurrence of the dynamic type, as shown in Theorem 6.10.

Simple definition of gradual type operators. All these properties were brought together in Section 6.2. Using the equivalent representation of gradual types, we could devise a very simply way to "lift" static types operators to gradual types (provided these operators are monotonic), while preserving all their semantic properties. The proofs of the soundness properties of the gradual operators are remarkably simple, especially when compared to the proofs of the same properties for the operators introduced in Chapter 5.

Simple operational semantics. In Chapter 5, set-theoretic types were more of a hindrance than a help. Accommodating the semantics for their presence required the definition of many complex notions, especially the *grounding* operation. In this chapter, we instead we were able to *efficiently use* set-theoretic types to make the operational semantics of our cast language much simpler. The notion of ground types is entirely encoded using the intersection connective, and the operational semantics solely depends on standard notions of semantic subtyping.

Shorter proofs. Last but not least, it must be noted that the proofs of the several properties presented throughout this chapter are *much* simpler and shorter than the proofs presented in Chapter 5. Not only is this a good metric to compare the complexity of the two approaches, it also is much less error-prone.

Chapter 7.

Discussion

The goal at the origin of this thesis was to combine polymorphic gradual typing and set-theoretic types. We soon realized that the task was hard, because the systems were intrinsically different: gradual typing is of syntactic nature ("?" is a syntactic placeholder), while set-theoretic types rely on a semantic-based definition of subtyping. To overcome this discrepancy, the only feasible option seemed to be to give a semantic-oriented interpretation of gradual types: dealing syntactically with set-theoretic types is unfeasible.

The solution we found to this impasse was to give a semantic interpretation of gradual types indirectly, by mapping them into sets of types that already had a semantic interpretation, namely those of Castagna and Xu [14]. Switching to a more semantic-oriented formalization makes all the chickens come home to roost. We realized that gradual typing, which was hitherto blurred in the typing rules, could be neatly perceived and captured by a subsumption-like rule, which we refer to as materialization, using the precision preorder on types. We also realized that the precision preorder was orthogonal to the much more common preorder on types that is subtyping and that, therefore, the two preorders could be coupled without much interference (but a lot of interplay).

More than that: when, for pedagogical purposes, we studied a restricted version of our system in Chapter 4, we realized that the restriction of precision to non set-theoretic types yielded a well-known relation with many names (precision, less-or-equally-informative, naive subtyping). While the relation was well known, it had never been singled out in a dedicated, structural rule of the type system. We did so, and thereby we demonstrated how adding the rule $[\mathrm{T}_{\mathsf{Mater}}]$ alone is enough to endow a declarative type system with graduality. We believe that this declarative formulation is a valuable contribution to the understanding of gradual typing and complements the algorithmic systems on which previous work has focused. As an example, materialization gives a new meaning to the cast calculus: its expressions encode the proofs of the declarative systems, and casts, in particular, spot the places where $[\mathrm{T}_{\mathsf{Mater}}]$ was used. Casts thus satisfy much stronger invariants than by using consistency, allowing for a simpler statement of blame safety.

That said, it is not all a bed of roses. While the precision relation and materialization may enlighten the cast calculus by a previously unseen logical meaning, to define its reduction rules in Section 5.3 we had to go back to the down-and-dirty syntax of types, which is not so easy, especially with set-theoretic types. This motivated us to pursue some work about the denotational aspects of gradual typing, which led to a new, more refined, and less syntax-dependent version of precision, which proved to be much easier to manipulate. This work will be presented in the second part of this manuscript. However, since the operational semantics of cast languages is outside the scope of this second part, we chose to introduce this new definition of the precision relation in Chapter 6. The flexibility of this new relation allowed us to simplify and improve considerably the semantics presented in Chapter 5.

Nevertheless, even with a syntactic definition of precision, we believe that our declarative formalization makes graduality more intelligible and that our work raises new questions and opens fresh, unforeseen perspectives, which we discuss towards the end of this chapter.

7.1. Related work

The contributions of this part of the thesis include the replacement of consistency with the materialization rule and the integration of gradual typing with set-theoretic types (intersection, union, negation, recursive) and Hindley-Milner polymorphism (with inference). The integration of all of these features is novel, but prior work has studied the combination of subsets of these features.

Toro and Tanter [74] introduce a new kind of union type inspired by gradual typing, that provides implicit downcasts from a union to any of its constituent types. There is some overlap in the intended use-cases of these gradual union types and our design, though there are considerable differences as well, given that our work handles polymorphism and the full range of set-theoretic types. A similar overlap exists with the work by Jafery and Dunfield [41] who introduce gradual sum types, yet, with the same kind of limitations as Toro and Tanter [74]. Ângelo and Florido [6] study the combination of gradual typing and intersection types, but in a somewhat limited form, as the design does not support subtyping or the other set-theoretic types. Ortin and García [54] also investigate the combination of intersection and union types with gradual typing, but without higher-order functions and polymorphism.

As discussed in Chapter 3, Siek and Vachharajani [67] showed how to do unification-based inference in a gradually typed language. Garcia and Cimini [30] took this a step further and provide inference for Hindley-Milner polymorphism and prove that their algorithm yields principal types. The work in this part of the thesis builds on this prior work and contributes the additional insight that a special-purpose constraint solver is not needed to handle gradual typing, but an off-the-shelf unification algorithm can be used in combination of some pre and post-processing of the solution. In another line of work, Rastogi et al. [59] develop a flow-based type inference algorithm for ActionScript to facilitate type specialization and the removal of runtime checks as part of their optimizing compiler. Campora et al. [10] improve the support for migrating from dynamic to static typing by integrating gradual typing with variational types. They define a constraint-based type inference algorithm that accounts for the combination of these two features.

The combination of gradual typing with subtyping has been studied by many authors in the context of object-oriented languages. Siek and Taha [66] showed how to augment an object calculus with gradual typing. Their declarative type system uses consistency in the elimination rules and has a subsumption rule to support subtyping. Their algorithmic type system combines consistency and subtyping into a single relation, consistent-subtyping. Many subsequent works adapted consistent-subtyping to different settings [40, 9, 71, 47, 31, 46, 77].

There is a long history of type inference with intersection types [63, 44]. The style of type inference known as soft typing employed union types [11, 5]. The set-constraints of Aiken and Wimmers [4] employed both intersection and union types. Our work builds on recent results by Castagna et al. [16] regarding type inference for languages with set-theoretic types and Hindley-Milner inference. Our work extends their approach to handle gradual typing. The addition of subtyping to a language presents a significant challenge for type inference, and there is a long line of work on this problem [4, 57, 22, 28, 50]. This challenge is intertwined with that of inference with intersection and union types, as we discussed in Section 5.4.

Ours is not the first line of work that tries to attack the syntactic hegemony currently ruling the gradual types community. The first and, alas hitherto unique, other example of this is the already cited work of Garcia et al. [31] on "Abstracting Gradual Typing" (AGT) (and its several follow-ups) which was a source of inspiration for our interpretation of gradual types. AGT uses abstract interpretation to relate gradual types to sets of static types. This is done via two functions: a *concretization* function that maps a gradual type τ into the set of static types obtained by replacing static types for all occurrences of ? in τ; an *abstraction* function that maps a set of static types to the gradual type whose concretization best approximates the set. Like AGT, we map gradual types into sets of static types, although they are different from those obtained by concretization, since we use type variables rather than generic static types. As long as only concretization is involved, we can follow and reproduce the AGT approach in ours: (1) AGT concretizations of a type τ can be defined in our system as the set of static types that are more precise than τ; (2) this definition can then be used to give a different characterization of the AGT's consistency relation; and (3) by using that characterization we can show consistency to be decidable, define consistent subtyping, and show that the problem of deciding consistent subtyping in AGT reduces in linear time to deciding semantic subtyping. But then it is not possible to follow the approach further since the AGT definition of the abstraction function is inherently syntactic and, thus, is unfit to handle type connectives whose definition is fundamentally of semantic nature. In other terms, we have no idea about whether —let alone how— AGT could handle set-theoretic types and this is why we had to find a new semantic characterizations of constructions that in AGT are smoothly obtained by a simple application of the abstraction function.

On the topic of gradual typing and polymorphism, there has been considerable work on explicit parametric polymorphism, in the context of System F [2, 3, 39] and Java Generics [40]. The presence of first-class polymorphism, as in System F, requires considerable care in the operational semantics of a cast calculus. In contrast, the second-class polymorphism (in the sense of Harper [33]) in this paper does not significantly impact the operational semantics because casts do not need to handle the universal type.

The operational semantics for cast calculi are informed by research on runtime contract enforcement, especially regarding blame tracking [24]. There is a large body of research on contracts; the most closely related to this paper are the intersection and union contracts of Keil and Thiemann [43] and the polymorphic contracts of Sekiyama et al. [64].

7.2. Future work

This work lays a foundation for integrating gradual typing and full set-theoretic types and, as such, it opens many new questions and issues. We detail the two main issues we think would be interesting to study in the near future.

7.2.1. Intersection types for functions

The first is to address a restriction we imposed to our system: namely, that it is not possible to assign intersection types to a function. Forbidding that (other than by subsumption) was an early design choice of this work, motivated by several reasons: its presence would complicate the dynamic semantics of the cast calculus; it would make type reconstruction and constraint solving much more difficult, and it would have probably hindered completeness even for simple systems; a system without this restriction would have been interesting only if the language had a type-case construct, which we wanted to avoid for simplicity and for sticking as close as possible

to ML.

As we discussed in the introduction, some of our previous work [13] featured a system supporting type-cases, gradual typing, and unrestricted intersection types for functions. This led to very complicated and ad-hoc semantics, which suffered from an exponential blow-up during compilation (one type-case had to be inserted for every sub-intersection of the interface of the function, which, for an intersection of n arrow types meant that 2^n type-cases had to be inserted). Apart from this restriction, the work presented in this part of the manuscript completely subsumes our previous work and features much simpler and intuitive semantics, hence its omission from the thesis.

However, the drawback is that we have function types that are less expressive than they could be. For instance, the type deduced for `mymap` in Chapter 3

$$\text{Bool} \to (\alpha \to \beta) \to \texttt{((}\alpha\,\texttt{array} \vee \alpha\,\texttt{list)} \wedge \texttt{?)} \to \texttt{(}\beta\,\texttt{array} \vee \beta\,\texttt{list)}$$

is not completely satisfactory insofar as it does not capture the precise correlation between input and output. As a matter of fact, the following program (which transforms lists into arrays and viceversa) would get the same type:

```
let mymap2 (condition) (f) (x : (α array | α list) & ?) =
  if condition then Array.to_list(Array.map f x) else Array.of_list(List.map f x)
```

It would be interesting to remove this restriction in future, so as to allow the system to check that (the unannotated version of) `mymap` has the type

$$\text{Bool} \to (\alpha \to \beta) \to \texttt{(((}\alpha\,\texttt{array \& ?)} \to \beta\,\texttt{array) \& ((}\alpha\,\texttt{list \& ?)} \to \beta\,\texttt{list))}$$

and that the new `mymap2` function has instead type

$$\text{Bool} \to (\alpha \to \beta) \to \texttt{(((}\alpha\,\texttt{array \& ?)} \to \beta\,\texttt{list) \& ((}\alpha\,\texttt{list \& ?)} \to \beta\,\texttt{array))}$$

two types where the correlation between the input and the output is more precisely described. In the long term not only we would like to check the types above, but also we plan to develop flow analyses that are able to infer such types for code without any type annotation.

7.2.2. Unifying our approach

When we started this work, we were guided by our intuition that `?` behaves as a type variable. While we quickly encountered some problems with negation types, we found a solution we deemed satisfactory (namely, the introduction of the concept of polarization), and this approach proved to be well-suited to type inference. By reducing precision to a unification problem, we could rely on the existing algorithm for tallying to design our type inference algorithm presented in Section 5.4.

However, for the sake of simplicity, we chose to only add pre- and post-processing steps to the tallying algorithm instead of truly extending it to support precision, as this would be quite a difficult undertaking. This led to a sound but incomplete type inference algorithm, since we could not guarantee that constraints involving recursive gradual types could be solved using a finite number of type variables.

Moreover, as we discussed in Section 5.3, this approach proved to also be quite problematic when defining the semantics of the associated cast language, since the syntactic aspect of precision required the introduction of many complex notions. This motivated our subsequent work

on the denotational aspects of gradual typing, which is presented in the second part of this manuscript.

This work proved to be fruitful: in Chapter 6, we introduced new versions of both precision and subtyping. By reducing precision entirely to static subtyping, we made it completely independent of the syntax of types, which in turn made it possible to derive much clearer semantics than the semantics presented in Section 5.3, at the cost of a slight change in the subtyping relation on static types.

This highlights an obvious direction for future work: introducing the new definitions of precision and subtyping right from the start (in the source language) instead of designing the source language and the cast language around two different relations. This would of course pose new problems for type inference, since precision constraints could not be solved using unification anymore. However, precision constraints could then be rewritten as subtyping constraints, which would have two important consequences. First, this would remove the need for the pre- and post-processing steps. Second, constraints would be solved without introducing new type variables to replace `?`, which may in turn restore the completeness of the inference algorithm.

Part II.

Denotational semantics

Chapter 8.

Introduction

The second part of this manuscript is devoted to denotational semantics. Our work on an operational semantics for a gradually-typed language with set-theoretic types (presented in Chapter 5) highlighted a fundamental gap in our understanding of the semantics of gradual types. This motivated us to study them from a denotational perspective, with the hope of finding a semantic interpretation of the precision and subtyping relations, which, as we argued in Part I, are the essence of gradual typing.

However, this was not our only goal. By taking a few steps back, and starting with a simple λ-calculus with set-theoretic types, we hoped to reconcile the interpretation of types in semantic subtyping with the interpretation of terms and values, thus filling in a missing piece in our understanding of semantic subtyping.

8.1. The denotational semantics of semantic subtyping

As we presented in Chapter 2, semantic subtyping is a technique to define a subtyping relation as set-theoretic containment. It consists in interpreting types as sets of values and then defining one type to be subtype of another if and only if the interpretation of the former is contained in the interpretation of the latter.

Since a subtyping relation is a pre-order, then it immediately induces the notions of least upper bound and greatest lower bound of a set of types. It is then natural to use such notions—thus, the subtyping relation—to characterize, respectively, union and intersection types. This property was used in the context of XML processing languages by Hosoya, Pierce, and Vouillon [38, 36, 35, 37]: by combining union types with product and recursive types it is easy to encode XML typing systems such as DTDs or XML Schemas. The work of Hosoya et al., however, had an important limitation, since it could not define the subtyping relation for functions types and, therefore, it could not be used to type languages with higher order functions. This impossibility resided in a circularity of the definition: to define subtyping one needs to define the type of each value; for non functional values this can be done by induction, but with functional values—i.e., λ-abstractions—this requires to type the bodies of the functions which, in turn, needs the very subtyping relation one is defining.

The solution to this circularity problem was found by Frisch et al. [26, 27] and consisted of three steps: (I) interpret types as sets of elements of some domain $\mathscr{D}$ and use this interpretation to define a subtyping relation; (II) use the subtyping relation just defined to type a functional language and in particular its values; (III) show that if we interpret a type as the set of values of this language that have that type, then this new interpretation induces the same subtyping relation as the starting one (which interprets types into subsets of the domain $\mathscr{D}$). To implement this solution there was a final important hurdle to clear, that is, to define a domain $\mathscr{D}$ in which it were possible to give a set-theoretic interpretation of function spaces. We presented a

possible definition of such a domain in Section 2.3 (Definition 2.19). In this implementation, the set-theoretic intuition we have of function spaces is that a function is of type $t\to s$ if whenever it is applied to a value of type t and it returns a result (i.e., the application does not diverge), then this result is of type s. Intuitively, if we interpret functions as binary relations on $\mathscr{D}$, then the interpretation of $t\to s$, noted $[\![t\to s]\!]$, should be the set of binary relations in which if the first projection is in (the interpretation of) t, then the second projection is in (the interpretation of) s, namely $\{f \subseteq \mathscr{D}^2 \mid \forall(d_1,d_2) \in f.\ d_1 \in [\![t]\!] \Rightarrow d_2 \in [\![s]\!]\}$. Note that this set is equivalent to $\mathscr{P}(\overline{[\![t]\!]\times\overline{[\![s]\!]}})$, where the over-line denotes set complement.[1] Thus, one would like to define $[\![t\to s]\!]$ as $\mathscr{P}(\overline{[\![t]\!]\times\overline{[\![s]\!]}})$, but this would imply that $\mathscr{P}(\mathscr{D}^2) \subseteq \mathscr{D}$, which is impossible for cardinality reasons. Frisch et al. [26, 27] realized that the three-step solution described above worked also if one considered only functions with *finite* graphs, that is, also if we define $[\![t\to s]\!]$ as $\mathscr{P}_f(\overline{[\![t]\!]\times\overline{[\![s]\!]}})$, where $\mathscr{P}_f(S)$ denotes the set of *finite* parts of a set S, and that was it: since it is easy to find a domain that satisfies $\mathscr{P}_f(\mathscr{D}^2) \subseteq \mathscr{D}$, then we had defined semantic subtyping also for function types (see [12] for a more extensive explanation and [27] for all the details).

That said, the solution made some readers uneasy. The fact of using finite graph functions to define a relation for general function spaces looked more as a technical trick than as a theoretical breakthrough. Pierre-Louis Curien suggested that the construction was a *pied de nez* to (it cocked a snook at) denotational semantics, insofar as it used a semantic construction to define a language for which a denotational semantics was not known to exist. The common belief was that the solution worked because considering *all* finite functions in the interpretation of a function space was equivalent to give the finite approximations of the non-finite functions in that space, in the same way as, say, Scott domains are built by giving finite approximations of the functions therein.

8.2. Our approach

In this part of the thesis we formalize the above intuition and define a denotational semantics for a language with semantic subtyping, in which functions are interpreted as the infinite set of their finite approximations. This yields a model with a simple *inductive* definition, which does not need isomorphisms or the solution of domain equation. One of the nice characteristics of the model we propose is that it does not distinguish between expressions and types. Both are interpreted as sets of (domain elements representing) values: the set of (all approximations of) all values typed by a type and the set of (all approximations of) all values produced by an expression (our interpretation accounts also for non-determinism). This is interesting insofar as semantic subtyping is particularly suitable to type languages with a type-case construction,[2] thus a denotational semantics that does not distinguish between types and expressions can be more easily extended to define the semantics of type-cases (even though there are other problems, as we will show in Chapter 11).

Defining a formal denotational semantics for ℂDuce seemed like a daunting task. Thus, we started by working on a very simple functional core calculus, with pairs and projections but no type-case construct. Following our goal of interpreting terms and types using the same domain, it seemed logical to try and define the semantics of this language using the interpretation domain of semantic subtyping presented in Section 2.3.

[1] Strictly speaking, the outermost is the complement w.r.t. $\mathscr{D}\times\mathscr{D}$, that is, $\mathscr{P}((\overline{[\![t]\!]\times\overline{[\![s]\!]}} \cap (\mathscr{D}\times\mathscr{D}))$; however, we will use this approximation for the sake of clarity.

[2] Actually, as we explain in Chapter 11 the presence of a type-case is a key ingredient to make the step (III) of Frisch et al. [26, 27]'s solution hold.

This first approach is presented in Chapter 9. It is based on a very simple idea: the denotational semantics of a function f is simply the set of all the finite relations mapping inputs x to outputs $f(x)$. For example, the denotational semantics of the successor function $\lambda x{:}\mathsf{Int}.\, x + 1$ is the set of all finite relations of the form $\{(n, n+1) \mid n \in N\}$ where $N \in \mathscr{P}_f(\mathbb{N})$. From there, the semantics of an application is computed by mapping all the elements denoting the argument by all the relations denoting the left hand side of the application.

This reasoning is only valid for well-typed terms. For example, consider the ill-typed application of the successor function to `true`. Using a standard β-reduction, this reduces to `true + 1`, which is a stuck term. However, following the above reasoning, the denotational semantics of this application is always empty, since no relation denoting the successor function can map `true` to a result. The *computational adequacy* property (as stated in Theorem 10.20) states that if the semantics of an expression is empty then it must diverge. Although this property only applies to well-typed terms, it would be interesting for the semantics to distinguish between diverging terms and terms that reduce to a stuck expression. Therefore, we need to introduce an element to denote stuck terms in our semantics. Thankfully, the semantic subtyping approach already introduces an element symbolizing a type error, which is denoted Ω (see Subsection 2.1.3). Thus, we formalize the meaning of Ω from a denotational point of view by allowing functions to map incompatible arguments to Ω.[3]

While this first approach seems fairly intuitive, it is not perfect. In particular, since relations can only map single elements to their results, this approach falls apart as soon as a function is applied to an argument whose denotation contains multiple elements. As a example, consider the function $\lambda x{:}\mathbb{1}.\, (x, x)$ which makes a pair from its argument. Intuitively, the semantics of a pair is obtained by computing the cartesian product of the semantics of its components. This means that, if we bind x to an element d in the above function, the semantics of the pair (x, x) is exactly the singleton $\{(d, d)\}$. This means that the relations denoting this function can only contain pairs of the form $(d, (d, d))$. However, if it is applied to an argument v whose denotation contain at least two elements d_1, d_2, then it should be able to return (d_1, d_2), as the application reduces to (v, v) which contains this denotation. This is reflected in the formulation of the *weak computational soundness* property (Theorem 9.10) which states that if a term $\mathbf{e}$ reduces to a term $\mathbf{e}'$, then the denotation of $\mathbf{e}$ is *a subset* of but not equal to the denotation of $\mathbf{e}'$.

To solve this problem, we modify the interpretation domain to allow relations to map a finite number of inputs to a single output. This second approach is presented in Chapter 10. Of course, modifying the interpretation domain for our denotational semantics goes against our initial goal of interpreting types and terms in the same domain. Therefore, we also define a new interpretation of types in this domain, which we show induces the same subtyping relation as the interpretation presented in Section 2.3. The semantics we present in Chapter 10 follows the same intuition as the semantics presented in Chapter 9, except two occurrences of the same variable can now be denoted by two different approximations of the same value. This allows us to get back the computational soundness property as expected: if a term $\mathbf{e}$ reduces to a term $\mathbf{e}'$, then their denotations are equal.

We go further by proving the *adequacy* of our semantics: if the denotation of a term is empty, then this term diverges. The proof of this property follows a technique inspired by the concept of *logical relations*. A logical relation relates expressions with their result, provided both are of a given type t (usually a parameter of the relation). However, we show that since terms and types

[3] In Chapter 13, we will discuss how we can further formalize the meaning of Ω as an error by introducing what we call a *typed* β-reduction, using which an application is only reduced if the type of the argument is compatible with the input type of the function.

are interpreted in the same domain, and since we have an induction principle on the interpretation domain, we can simplify this technique by directly relating a term with the denotation of its result.

Having defined sound and adequate semantics for our functional core calculus, we then proceed in Chapter 11 with our goal of defining a denotational semantics for ℂDuce . This requires three additions to our functional core calculus: we must allow functions to be typed with non-trivial intersection of arrow types; we must add a type-case construct; and we must add some form of non-determinism.

Adding support for non-deterministic expression is fairly straightforward: we simply extend the domain with *marks*, which are strings denoting the non-deterministic path that led to a result. In our setting, we consider a binary choice operator $\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2)$ that randomly reduces to $\mathbf{e}_1$ or to $\mathbf{e}_2$. We then consider marks to be strings on the alphabet $\{l, r\}$, where every character represents a choice. For example, if a expression is denoted by 3^{lr}, then it means that it reduces to 3 provided that the first choice we encounter reduces to its left hand side, and the second reduces to its right hand side.

Adding support for type-cases is, also, fairly intuitive. As anticipated, since types and terms are interpreted as sets of elements of the same domain, comparing the type of the result of an expression $\mathbf{e}$ to a type t simply amounts to comparing the denotation of $\mathbf{e}$ to the semantic interpretation of t using set-containment. Therefore, to compute the semantics of a type-case $(x = \mathbf{e} \in t)? \mathbf{e}_1 : \mathbf{e}_2$, we simply compute the semantics of $\mathbf{e}$, check whether it is included in the semantic interpretation of t, and compute the semantics of $\mathbf{e}_1$ or $\mathbf{e}_2$ accordingly.

Most of the difficulty lies in the derivation of non-trivial intersection types for functions. To achieve this, we equip functions with what we call *interfaces*: instead of only specifying the input type of a function, we annotate it using an intersection of arrow types. For example, we can now write $\lambda^{(\mathsf{Int}\to\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool})} x.\, x$ for the identity function that can both be applied to integers (returning integers) and to booleans (returning booleans).

Adding support for interfaces in our semantics is difficult for two major reasons. The first is that the return type of a function is now made explicit. This means that we must take into account this new piece of information when defining the semantics of a function. The second reason is that, to ensure the soundness of the semantics in the presence of type-cases, it is crucial to ensure that for every value v and every type t, then either v has type t or v has type $\neg t$. This means, in particular, that if a function cannot be given type $\mathsf{Nat} \to \mathsf{Nat}$ according to its interface, then it must be of type $\neg(\mathsf{Nat} \to \mathsf{Nat})$, *even* if it maps natural numbers to natural numbers. For example, the function $\lambda^{\mathsf{Nat}\to\mathsf{Int}} x.\, x$ cannot be given type $\mathsf{Nat} \to \mathsf{Nat}$ by subtyping, so the type system must be able to give it type $\neg(\mathsf{Nat} \to \mathsf{Nat})$.

Our solution to these two difficulties lies in the extension of finite relations with a new form of input, and in the addition of explicit negative type annotations to λ-abstractions. This allows us to define a sound denotational semantics for our language, which, we argue, is still a close representation of ℂDuce despite the modifications we introduced.

Finally, in Chapter 12, we introduce gradual types. We start by providing a set-theoretic interpretation of gradual types, based on an entirely new interpretation domain which draws inspiration from the domains presented in Chapter 10 and Chapter 11.

The main idea behind this new interpretation is to distinguish between the values that always belong to a type, independently of how the occurrences of ? are refined, and the values that may belong to a type, provided the occurrences of ? are refined in a certain way. For example, the value 3 always belong to $? \vee \mathsf{Int}$, because it is clear that for every type t, it belongs to $t \vee \mathsf{Int}$. However, it does not always belong to ?, but it *may* do so, if ? is refined to be a type containing

3. To distinguish between the two, we introduce *tags* to our interpretation domain.

Using this interpretation and by interpreting subtyping as set-containment following the semantic subtyping approach, we deduce a subtyping relation on gradual types that satisfies very strong properties. Moreover, we also deduce a set-theoretic interpretation of the *precision* of a type, which yields a semantic definition of a precision relation, which is closely related to the relation of the same name we introduced in the first part of this manuscript. We prove that subtyping and precision are actually strongly related, as both can be expressed in terms of subtyping on static types.

Finally, we apply this work to deduce a denotational semantics for a gradually-typed cast calculus. Due to the complexity of this task, we restrict ourselves to a cast calculus with simple types. We present in particular a denotational interpretation of casts, by formalizing the action of a cast on an element of the interpretation domain. To conclude this part, we prove two soundness results for our semantics.

8.3. Contributions

The main contributions of this part of the thesis can be thus summarized.

- A denotational semantics for functional languages with semantic subtyping (i.e., higher-order functional languages with type-case expressions and union, intersection, and negation types) that we prove to be sound. We moreover prove the adequacy of the semantics restricted to a simple functional core (without type-case expressions).
- A model with a simple domain defined inductively—without resorting to isomorphisms or the resolution of domain equations—whose elements capture results finiteness.
- A new view of continuity: as Scott continuity is a key ingredient to make denotational semantics provide an *extensional* view of functional programs, so it seems to us that our model provides an *extensional* view of Scott continuity, insofar as it explicitly states that the essence of computable functions is to map finite enumerations of basis elements of the argument to finite enumerations of basis elements of the result which, according to Stoy [70, pp. 97-105], is the very nature of computation (see in particular Condition 6.39 of [70] and its implications).
- A technique to reconcile two features that rarely fit together, namely, the usage of sets of finite approximations (used to interpret functions) with the usage of set-theoretic complement (used to interpret negation types).
- An encoding of the language of Frisch et al. [26, 27] into a simpler language in which recursively defined functions are not primitive but encoded.
- A set-theoretic interpretation of gradual types which entails simple, semantic definitions of both gradual subtyping and precision.
- A new view of set-theoretic gradual types as intervals: using the intersection and union connectives, we highlight a bijection between gradual types and pairs of static types.
- A sound denotational semantics for a simply-typed cast language, which features a denotational interpretation of casts and blame.

Chapter 9.

A functional core calculus with set-theoretic types

> *"A wall is happy when it is well designed, when it rests firmly on its foundation, when its symmetry balances its part and produces no unpleasant stresses. Good design can be worked out on the mathematical principles of mechanics."*
>
> Isaac Asimov, *Foundation's Edge*

In this chapter, we define a denotational semantics for a simple λ-calculus with pairs and a limited use of set-theoretic types, by interpreting terms of this calculus into elements of the domain $\mathscr{D}$ defined in Chapter 2. We hightlight several problems with this approach, which will guide our subsequent work in Chapter 10.

Chapter Outline

Section 9.1 We present the syntax, type system and operational semantics of our functional core calculus λ_{F}, a simply-typed λ-calculus that supports a limited use of set-theoretic types through the use of type annotations on the parameters of λ-abstractions.

Section 9.2 We use the interpretation domain of semantic subtyping $\mathscr{D}$ to give a denotational semantics for λ_{F}. This is done in several steps: we first explain how to give a semantics to abstractions, then deal with type errors, before providing the full denotational semantics.

Section 9.3 We state several properties of our denotational semantics, including soundness properties. These properties ensure that our semantics properly implements the operational semantics of λ_{F} for non-diverging expressions. We hightlight some issues with these results, which will motivate the changes presented in the following chapter.

Section 9.4 We present the property of computational adequacy, which is the counterpart of the soundness properties for diverging expressions. The proof of this result, which uses a technique inspired by logical relations, will be presented in the next chapter.

9.1. Presentation of λ_{F}

9.1.1. Syntax of λ_{F}

The functional core calculus λ_{F} is a fairly standard simply typed λ-calculus equipped with pairs and which supports a limited use of set-theoretic types. In particular, it is not possible in λ_{F} to assign intersection types to functions, except for trivial ones obtained by subsumption, such as

$(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathbb{0} \to \mathbb{1})$. This prevents us from using a function with two different types, thus forbidding overloading, which is, arguably, one of the main strengths of set-theoretic types. We will however handle this point in Chapter 11.

Throughout all this part of the manuscript, we consider the set Types of set-theoretic types to be defined as in Chapter 2 (Definition 2.2), except we restrict ourselves to monomorphic types. As a reminder, we have the following definition:

Definition 9.1 (Set-theoretic types)**.** *The set* Types *of* set-theoretic types *is the set of terms t generated coinductively by the following grammar:*

$$\mathsf{Types} \ni t \quad ::= \quad \alpha \mid b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0} \quad \textit{set-theoretic types}$$

where b ranges over $\mathscr{B}$ and that satisfy the following two conditions:

- (regularity) *the term has a finite number of different sub-terms;*
- (contractivity) *every infinite branch of a type contains an infinite number of occurrences of the $\times$ or $\to$ type constructors.*

The *terms* $\mathbf{e} \in \mathsf{Terms}$ and the *values* $\mathbf{v} \in \mathsf{Values}$ of λ_{F} are defined inductively by the following grammar

$$\begin{array}{rcl} \mathsf{Terms} \ni \mathbf{e} & ::= & c \mid x \mid \lambda x{:}t.\, \mathbf{e} \mid \mathbf{e}\,\mathbf{e} \mid \pi_i\, \mathbf{e} \mid (\mathbf{e}, \mathbf{e}) \\ \mathsf{Values} \ni \mathbf{v} & ::= & c \mid \lambda x{:}t.\, \mathbf{e} \mid (\mathbf{v}, \mathbf{v}) \end{array}$$

where, additionally, the set Values is restricted to closed well-typed terms for the type system of Figure 9.1. That is, a term can only be a value if it is closed and well-typed, otherwise it is a stuck term.

In this definition, x ranges over a countable set of variables Vars, c over the set of constants $\mathscr{C}$, and i in π_i ranges over $\{1, 2\}$. Moreover, as we stated above, the parameter of a λ-abstraction can and must be explicitly annotated by a set-theoretic type $t \in \mathsf{Types}$. However, without subsumption, the most precise inferred type for a λ-abstraction $\lambda x{:}t.\, \mathbf{e}$ will always be a single arrow type, with t as the domain.

9.1.2. Operational semantics

The operational semantics of λ_{F} follows a standard call by value strategy, where evaluation contexts implement a left-most outer-most weak reduction strategy. Formally, the reduction rules are stated in a small-step style as follows.

$$\begin{array}{rrcll} [\mathrm{R}^{\mathrm{F}}_{\mathrm{app}}] & (\lambda x{:}t.\, \mathbf{e})\, \mathbf{v} & \rightsquigarrow & \mathbf{e}\,[\mathbf{v}/x] & \\ [\mathrm{R}^{\mathrm{F}}_{\mathrm{proj}_i}] & \pi_i\, (\mathbf{v}_1, \mathbf{v}_2) & \rightsquigarrow & \mathbf{v}_i & \text{for } i \in \{1, 2\} \\ [\mathrm{R}^{\mathrm{F}}_{\mathrm{ctx}}] & \mathscr{E}\,[\mathbf{e}] & \rightsquigarrow & \mathscr{E}\,[\mathbf{e}'] & \text{if } \mathbf{e} \rightsquigarrow \mathbf{e}' \end{array}$$

In these reduction rules, we use the notation $\mathbf{e}\,[\mathbf{v}/x]$ to denote the capture-avoiding substitution of $\mathbf{v}$ for the variable x in the term $\mathbf{e}$. We use $\mathscr{E}\,[\mathbf{e}]$ to denote the term obtained by replacing the hole of the evaluation context $\mathscr{E}$ by the term $\mathbf{e}$. Evaluation contexts are defined inductively by the following grammar, where [] stands for the hole of the context, implementing the strategy we stated before.

$$\mathscr{E} ::= [] \mid \mathscr{E}\,\mathbf{e} \mid \mathbf{v}\,\mathscr{E} \mid (\mathscr{E}, \mathbf{e}) \mid (\mathbf{v}, \mathscr{E}) \mid \pi_i\, \mathscr{E}$$

$$[\mathrm{T}^{\mathrm{F}}_{\mathrm{Cst}}]\ \frac{}{\Gamma \vdash c : b_c} \qquad [\mathrm{T}^{\mathrm{F}}_{\mathrm{Var}}]\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad [\mathrm{T}^{\mathrm{F}}_{\mathrm{Sub}}]\ \frac{\Gamma \vdash \mathbf{e} : t}{\Gamma \vdash \mathbf{e} : t'}\ t \leq t'$$

$$[\mathrm{T}^{\mathrm{F}}_{\mathrm{Abs}}]\ \frac{\Gamma, x : t \vdash \mathbf{e} : t'}{\Gamma \vdash \lambda x{:}t.\, \mathbf{e} : t \to t'} \qquad [\mathrm{T}^{\mathrm{F}}_{\mathrm{App}}]\ \frac{\Gamma \vdash \mathbf{e}_1 : t \to t' \quad \Gamma \vdash \mathbf{e}_2 : t}{\Gamma \vdash \mathbf{e}_1\, \mathbf{e}_2 : t'}$$

$$[\mathrm{T}^{\mathrm{F}}_{\mathrm{Pair}}]\ \frac{\Gamma \vdash \mathbf{e}_1 : t_1 \quad \Gamma \vdash \mathbf{e}_2 : t_2}{\Gamma \vdash (\mathbf{e}_1, \mathbf{e}_2) : t_1 \times t_2} \qquad [\mathrm{T}^{\mathrm{F}}_{\mathrm{Proj_i}}]\ \frac{\Gamma \vdash \mathbf{e} : t_1 \times t_2}{\Gamma \vdash \pi_i\, \mathbf{e} : t_i}$$

FIGURE 9.1. Typing rules for λ_{F}

9.1.3. Type system

To complete the presentation of λ_{F}, we now equip it with a type system. The full declarative definition of the type system is given in Figure 9.1. By declarative, we mean that subtyping is added as a simple, non-syntax-directed subsumption rule.

Once again, this type system is fairly standard. Statements are of the form $\Gamma \vdash \mathbf{e} : t$ where t ranges over set-theoretic types Types and Γ ranges over *type environments*, that is, finite mappings from variables Vars to Types. We use $\Gamma, x : t$ to denote the environment obtained by extending Γ with a mapping from x to t. In practice, we ensure that no mapping is associated to x in Γ beforehand using α-renaming.

The rule $[\mathrm{T}^{\mathrm{F}}_{\mathrm{Cst}}]$ makes use of the function $b_{(.)} : \mathscr{C} \to \mathscr{B}$ from semantic subtyping that associates a base type b_c to every constant c. The rule $[\mathrm{T}^{\mathrm{F}}_{\mathrm{Var}}]$ simply looks up the type of a variable x into the environment Γ, which we denote by $\Gamma(x)$. As we stated before, there is no rule that allows us to deduce intersection types for λ-abstractions, apart from trivial ones obtained by subsumption.

As customary for such a calculus, it satisfies the properties of progress (i.e., every well-typed term is either a value or can be reduced) and subject-reduction (i.e., the type of a term is preserved by reduction). As such, it is sound with respect to its type system in the sense of Wright and Felleisen.

Theorem 9.2 (Type soundness of λ_{F})**.** *Let* $\mathbf{e} \in$ Terms*. If* $\emptyset \vdash \mathbf{e} : t$ *then either* $\mathbf{e}$ *diverges, or there exists* $\mathbf{v} \in$ Values *such that* $\mathbf{e} \rightsquigarrow^* \mathbf{v}$ *and* $\emptyset \vdash \mathbf{v} : t$*.*

Proving the two properties above (and thus the type soundness theorem) for such a standard calculus is a routine exercise, and can be done by induction on the terms at hand. We will not detail these proofs in this manuscript (they can be found in [27]).

9.2. Denotational semantics

Now that the type system and the operational semantics of λ_{F} have been presented, we are ready to provide a denotational semantics of λ_{F}. Our goal is to define an interpretation domain and an interpretation function $[\![.]\!]$ that associates to every term $\mathbf{e}$ of our calculus a set of *denotations* taken from the interpretation domain. However, instead of defining a new interpretation domain, our semantics will map terms of λ_{F} into sets of elements of $\mathscr{D}$, the domain used to define semantic

subtyping presented in Chapter 2. This further emphasizes the correspondence between types and sets of values.

Two main questions arise when defining a denotational semantics for λ_{F} using the domain $\mathscr{D}$, for which we dedicate two subsections. The first one concerns the semantics of annotated λ-abstractions, since the semantics of a λ-abstraction must also depend on its type annotation. The second one is linked to Ω, which, in our interpretation domain, represents a failure. Such a failure must be propagated properly in our semantics.

9.2.1. Dealing with annotated λ-abstractions

Providing a denotational semantics for λ-abstractions is certainly the most difficult part of this work. For well-foundedness reasons, similar to the reasons found in the set-theoretic interpretation of semantic subtyping, the denotations of λ-abstractions must be kept finite. However, it is clear that a function such as $\lambda x{:}\mathbb{1}.\, x$ cannot be represented by a single finite relation, as it maps *every* value into itself, and there is an infinite number of values in our system. Thus, if we are to base our denotational semantics on the interpretation domain $\mathscr{D}$ (in which relations are always finite), we must necessarily interpret λ-abstractions as the *infinite set* of their finite approximations, similarly to arrow types.

Now, recall that we defined the set-theoretic interpretation of arrow types in Definition 2.5 as follows:

$$[\![t_1 \to t_2]\!] = \{R \in \mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega) \mid \forall (d, \partial) \in R,\ d \in [\![t_1]\!] \implies \partial \in [\![t_2]\!]\}$$

Notice in particular how the right hand side of every pair is constrained to $[\![t_2]\!]$ only if its left hand side is an element of $[\![t_1]\!]$, and can be anything otherwise. This is fundamental, as this ensures the contravariance of semantic subtyping with respect to the domain of arrow types. One might be tempted to apply the same principle to deduce the denotational semantics of λ-abstractions, stating that the denotational semantics of an abstraction $[\![\lambda x{:}t.\, \mathbf{e}]\!]$ is the set of relations made of pairs (d, ∂) such that, if the input d belongs to $[\![t]\!]$, then the output ∂ belongs to the semantics of $\mathbf{e}$, provided we somehow bind x to the input d.

This raises two questions. The first one has to do with the binding of variables when computing the denotational semantics. The solution is fairly standard and consists in adding an environment as a parameter to our semantics. This environment will map variables to denotations, and will be denoted by ρ. The formal definition of this environment will be presented later on, along with the full denotational semantics, in Subsection 9.2.3.

The second question is more about personal preference. Consider, for example, the identity function restricted to integers `idInt` $= \lambda x{:}\mathsf{Int}.\, x$, and suppose that we apply this function in an ill-typed context, such as `idInt true`. With the semantics we hinted at, the semantics of this application would be the set of all possible denotations, including Ω, since the semantics of `idInt` would map `true` to anything. Is this reasonable? Well, this would certainly not impact the soundness and the adequacy of our semantics, since these results only apply to well-typed terms. However, it seems quite unexpected that this function can return, for example, `42`, when applied to `true`. We could get rid of the condition on the input type, and simply ask that for every pair (d, ∂), the result ∂ belongs to the semantics of the body of the abstraction; but this then raises the problem of differentiating between functions that share the same body with different type annotations.

With this in mind, the solution we chose is to make use of both type annotations and Ω, to ensure that λ-abstractions used in an ill-typed context necessarily return Ω. The denotational

semantics of abstractions, in terms of an environment $\rho : \text{Vars} \to \mathscr{D}$, can be given as follows:

$$[\![\lambda x{:}t.\, \mathbf{e}]\!]_\rho = \{R \in \mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega) \mid \forall (d, \partial) \in R,\ d \in [\![t]\!] \implies \partial \in [\![\mathbf{e}]\!]_{\rho, x \mapsto d} \\ d \notin [\![t]\!] \implies \partial = \Omega\}$$

Notice how in particular the environment is used to bind the parameter to the argument using the notation $\rho, x \mapsto d$. Moreover, as we stated above, when the input d is not of the input type t, this semantics forces the output to be Ω.

Now that we have given a denotational semantics to λ-abstractions, the semantics of applications is straightforward, and we already heavily hinted at it in the previous paragraphs. Ignoring, for now, the propagation of Ω (which we will handle in the next subsection), computing the semantics of an application $\mathbf{e}_1\, \mathbf{e}_2$ is simply a matter of mapping every denotation of $\mathbf{e}_2$ by every possible denotation of $\mathbf{e}_1$. Formally, this yields the following definition:

$$[\![\mathbf{e}_1\, \mathbf{e}_2]\!]_\rho = \{\partial \in \mathscr{D}_\Omega \mid \exists d \in [\![\mathbf{e}_2]\!]_\rho, R \in [\![\mathbf{e}_1]\!]_\rho, (d, \partial) \in R\} \tag{9.1}$$

In light of the previous explanation, this definition looks quite intuitive. However, it features a peculiarity we ought to highlight. While the semantics ensures that a λ-abstraction applied in an ill-typed context *will* return Ω, it does not guarantee that Ω will be the *only* result of the application. Intuitively, this is due to the fact that an application *may be* well-typed for a particular denotation of the argument, and ill-typed for another.

Consider, as an example, the functions $\texttt{cstApp} = \lambda f{:}\mathsf{Nat} \to \mathsf{Nat}.\ (f\ 42)$ and $\texttt{pred} = \lambda x{:}\mathsf{Int}.\ (x - 1)$, of types $(\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat}$ and $\mathsf{Int} \to \mathsf{Int}$ respectively. Since $\mathsf{Int} \to \mathsf{Int}$ is not a subtype of $\mathsf{Nat} \to \mathsf{Nat}$, the application of $\texttt{cstApp}\ \texttt{pred}$ is not well-typed. This is reflected by the fact that its semantics contains Ω. Indeed, if we consider the relation $\{(0, -1)\}$, which is a denotation of `pred` (it is a finite approximation of this function), it is not in $[\![\mathsf{Nat} \to \mathsf{Nat}]\!]$ since it maps a natural number (0) to a non-natural number (-1). Therefore, according to our semantics, it must be mapped to Ω in the semantics of `cstApp`. However, the relation $\{(42, 41)\}$ is *also* a denotation of `pred`, but this denotation belongs to $[\![\mathsf{Nat} \to \mathsf{Nat}]\!]$. Therefore, our semantics for `cstApp` will map this denotation to 41, and in the end, $[\![\texttt{cstApp}\ \texttt{pred}]\!] = \{\Omega, 41\}$.

9.2.2. Handling failure

We have shown through several examples that the semantics we gave to λ-abstractions and to applications can produce Ω as a result which, intuitively, corresponds to a stuck reduction. The problem here is that we currently do not propagate such failures. Consider the semantics we gave to applications in Formula 9.1, and suppose that $\mathbf{e}_1$ is ill-typed and that its semantics contains Ω. In the current state of the semantics, this denotation will simply be ignored when computing $[\![\mathbf{e}_1\, \mathbf{e}_2]\!]_\rho$.

To solve this problem, we introduce a new operator noted $\Omega^\rho_{(.)}$ that acts on a term under an environment ρ. The goal of this operator is to simply handle Ω, that is, for every term $\mathbf{e} \in \text{Terms}$, $\Omega^\rho_\mathbf{e} = \{\Omega\}$ if and only if the denotational semantics of $\mathbf{e}$ under the environment ρ contains Ω, and $\Omega^\rho_\mathbf{e} = \emptyset$ otherwise.

There are two cases where we want to explicitly add Ω to the semantics of a term $\mathbf{e}$. The first case is when Ω appears in the semantics of one of the sub-expressions of $\mathbf{e}$, which we discussed earlier in the case of the application. The second case occurs when one tries to apply a value that is not a function, or to project a value that is not a pair. Clearly, both cases are not well-typed and should produce a type failure. Formally, this yields the following definition of the operator

$\Omega^{(\cdot)}_{(\cdot)}$.

Definition 9.3 (Failure operator $\Omega^{(\cdot)}_{(\cdot)}$)**.** *For every term* $\mathbf{e} \in \mathsf{Terms}$*, and every environment* $\rho : \mathsf{Vars} \to \mathscr{D}$*, we have* $\Omega^{\rho}_{\mathbf{e}} = \{\Omega\}$ *if any of the following conditions holds*

1. $\mathbf{e} \equiv \mathbf{e}_1\, \mathbf{e}_2$ *and* $\Omega \in [\![\mathbf{e}_1]\!]_\rho$ *or* $[\![\mathbf{e}_1]\!]_\rho \neq \emptyset$ *and* $\Omega \in [\![\mathbf{e}_2]\!]_\rho$
2. $\mathbf{e} \equiv \mathbf{e}_1\, \mathbf{e}_2$ *where* $[\![\mathbf{e}_2]\!]_\rho \neq \emptyset$ *and* $\exists d \in [\![\mathbf{e}_1]\!]_\rho$ *such that* $d \notin \mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega)$
3. $\mathbf{e} \equiv \pi_i\, \mathbf{e}'$ *and* $\Omega \in [\![\mathbf{e}']\!]_\rho$
4. $\mathbf{e} \equiv \pi_i\, \mathbf{e}'$ *and* $\exists d \in [\![\mathbf{e}']\!]_\rho$ *such that* $d \notin \mathscr{D} \times \mathscr{D}$
5. $\mathbf{e} \equiv (\mathbf{e}_1, \mathbf{e}_2)$ *and* $\Omega \in [\![\mathbf{e}_1]\!]_\rho$ *or* $[\![\mathbf{e}_1]\!]_\rho \neq \emptyset$ *and* $\Omega \in [\![\mathbf{e}_2]\!]_\rho$

and $\Omega^{\rho}_{\mathbf{e}} = \emptyset$ *otherwise.*

Cases (1), (3) and (5) correspond to Ω appearing in a sub-expression of an application, a projection, and a pair, respectively. Cases (2) and (4) correspond to expressions which apply a value that is not a function, or project a value that is not a pair, respectively. Note that a peculiarity of case (1) and (5) is that we only propagate a type error occurring in $\mathbf{e}_2$ if $\mathbf{e}_1$ is non-empty. Indeed, since we are in a strict setting, and use a left to right reduction strategy, the application of a diverging expression to an ill-typed argument diverges and will never get stuck. We reflect this fact in case (1) by ensuring that if $\mathbf{e}_1$ diverges, then the semantics of $\mathbf{e}_1\, \mathbf{e}_2$ is empty, (it does not even contain Ω, even if $\mathbf{e}_2$ is stuck). Similarly for case (2) we require that $[\![\mathbf{e}_2]\!]_\rho \neq \emptyset$ since applying a non-functional value to an expression produces a stuck term only if this expression does not diverge.

Finally, notice that we do not propagate errors occurring inside the body of a λ-abstraction: since our calculus uses a weak reduction strategy, a λ-abstraction is always a value and will therefore never be stuck.

9.2.3. Denotational semantics for λ_{F}

Now that we have dealt with the two main difficulties of our semantics, we can give its formal definition. We start with the formal definition of environments.

Definition 9.4 (Semantic environments)**.** *A semantic environment is a function* $\mathsf{Vars} \to \mathscr{D}$*. We use* Envs *to denote the set of such environments, and use* ρ *to range over this set:*

$$\mathsf{Envs} \ni \rho : \mathsf{Vars} \to \mathscr{D}$$

Moreover, given a variable $x \in \mathsf{Vars}$ *and a denotation* $d \in \mathscr{D}$*, we use the notation* $\rho, x \mapsto d$ *to denote the environment obtained by extending* ρ *with a mapping from* x *to* d*.*

As usual, we ensure via α-renaming that no conflict happens when extending environments. We now give the full definition of our denotational semantics.

Definition 9.5 (Set-theoretic interpretation of λ_{F})**.** *Let* $\rho \in \mathsf{Envs}$*. We define the set-theoretic*

interpretation of λ_F as a function $[\![.]\!]_{(.)} : \mathsf{Terms} \to \mathsf{Envs} \to \mathscr{P}_f(\mathscr{D}_\Omega)$ as follows:

$$
\begin{array}{rcl}
[\![x]\!]_\rho & = & \{\rho(x)\} \\
[\![c]\!]_\rho & = & \{c\} \\
[\![\lambda x{:}t.\,\mathbf{e}]\!]_\rho & = & \{R \in \mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega) \mid \forall (d,\partial) \in R, \quad d \in [\![t]\!] \implies \partial \in [\![\mathbf{e}]\!]_{\rho, x \mapsto d} \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad d \notin [\![t]\!] \implies \partial = \Omega\} \\
[\![\mathbf{e}_1\,\mathbf{e}_2]\!]_\rho & = & \{\partial \in \mathscr{D}_\Omega \mid \exists d \in [\![\mathbf{e}_2]\!]_\rho, R \in [\![\mathbf{e}_1]\!]_\rho, (d,\partial) \in R\} \cup \Omega^\rho_{\mathbf{e}_1\,\mathbf{e}_2} \\
[\![\pi_i\,\mathbf{e}]\!]_\rho & = & \{d_i \mid (d_1,d_2) \in [\![\mathbf{e}]\!]_\rho\} \cup \Omega^\rho_{\pi_i\,\mathbf{e}} \\
[\![(\mathbf{e}_1,\mathbf{e}_2)]\!]_\rho & = & ([\![\mathbf{e}_1]\!]_\rho \setminus \{\Omega\}) \times ([\![\mathbf{e}_2]\!]_\rho \setminus \{\Omega\}) \cup \Omega^\rho_{(\mathbf{e}_1,\mathbf{e}_2)}
\end{array}
$$

In light of the preceding explanations, most of the rules should be pretty straightforward. The semantics of a variable x is simply the denotation that has been bound to x in the current environment ρ. The semantics of a constant is the singleton that contains this constant only. We already explained the rules for λ-abstractions and applications, simply notice the use of the operator $\Omega^{(.)}_{(.)}$ in the semantics of the application, which ensures that failures occurring in both sides of the application are correctly propagated. The semantics of a projection $\pi_i\,\mathbf{e}$ simply amounts to taking all the pairs in the semantics of $\mathbf{e}$, and extracting their i-th component. If $\mathbf{e}$ contains some element that is not a pair, then the operator $\Omega^{(.)}_{(.)}$ produces a type failure. Finally, the semantics of a pair is given by the cartesian product of the semantics of each component, provided we remove all pairs containing Ω since they are not elements of $\mathscr{D}$. Once again, the operator $\Omega^{(.)}_{(.)}$ takes care of Ω appearing in any of the components.

9.3. Soundness properties

Now that we presented the denotational semantics of λ_{F}, we express and prove its soundness. The soundness of a denotational semantics is two-sided: it must be sound with respect to both the type system and the operational semantics of the underlying language. We dedicate a subsection to each aspect.

9.3.1. Type soundness

Type soundness is the first part composing the overall soundness of a denotational semantics. It states that, if a term is well-typed of a given type t, then all its denotations "belong to" this type t. It is, in a sense, the denotational counterpart to subject reduction for operational semantics.

For our semantics, this is where semantic subtyping truly shines. Since both types and expressions are interpreted using the same interpretation domain $\mathscr{D}$, checking that a denotation d "belongs to" a given type t is simply a matter of checking whether $d \in [\![t]\!]$. For closed terms, type soundness could therefore simply be stated as:

$$\forall \mathbf{e} \in \mathsf{Terms},\ \vdash \mathbf{e} : t \implies [\![\mathbf{e}]\!] \subseteq [\![t]\!]$$

However, we would like to give a slightly stronger version of the theorem, supporting terms with free variables, well-typed under a type environment Γ. This requires adding a semantic environment ρ to the previous statement, to give a semantics to the free variables occurring in $\mathbf{e}$. This semantics cannot be *any* semantics, they must at least respect the type information given in the environment Γ. For example, should Γ assign type Int to a variable x, then the environment ρ must only assign denotations of integers to x. To achieve this, we define the denotational interpretation of a type environment as follows.

Definition 9.6 (Denotational interpretation of Γ)**.** *Let* $\Gamma \in \mathsf{TEnvs}$*. We define its denotational interpretation, noted* $[\![\Gamma]\!]$*, as the function*

$$[\![.]\!] : \mathsf{TEnvs} \to \mathscr{P}(\mathsf{Envs})$$
$$[\![\Gamma]\!] = \{\rho \in \mathsf{Envs} \mid \forall x \in \mathsf{Dom}(\Gamma),\ \rho(x) \in [\![\Gamma(x)]\!]\}$$

Intuitively, given a type environment Γ, $[\![\Gamma]\!]$ is the set of all the environments $\rho \in \mathsf{Envs}$ that map every variable present in Γ to a denotation compatible with its type. Using this, we can now state the type soundness theorem for our semantics.

Theorem 9.7 (Type soundness for λ_{F})**.** *For every type environment* $\Gamma \in \mathsf{TEnvs}$ *and every term* $\mathbf{e} \in \mathsf{Terms}$*, if* $\Gamma \vdash \mathbf{e} : t$ *then for every* $\rho \in [\![\Gamma]\!]$*,* $[\![\mathbf{e}]\!]_\rho \subseteq [\![t]\!]$*.*

This theorem can be proven by standard induction on $\mathbf{e}$. We will not detail the proof in this chapter, instead we will prove this result for the extension of our denotational semantics in the next chapter, and prove some conservativity results. Note that since Ω does not belong to the interpretation of any type, this theorem yields the following immediate corollary:

Corollary 9.8. *For every type environment* $\Gamma \in \mathsf{TEnvs}$ *and every term* $\mathbf{e} \in \mathsf{Terms}$*, if* $\Gamma \vdash \mathbf{e} : t$ *then for every* $\rho \in [\![\Gamma]\!]$*,* $\Omega \notin [\![\mathbf{e}]\!]_\rho$*.*

This corollary partially formalizes the fact that Ω is a type failure and does not occur in the semantics of a well-typed expression. While interesting, this result is pretty weak since it does not truly give a meaning to the appearance of Ω in the semantics of a term.

In Chapter 13, we will discuss a way to obtain a much stronger result, provided we slightly modify our reduction rules to use a typed β-reduction, using which an application is only reduced if the type of the argument is compatible with the input type of the function. By introducing such a rule, we postulate that it is possible to prove that if the semantics of a term contains Ω, then this term necessarily reduces to a stuck term. Along with Theorem 9.7, such a result would provide a denotational equivalent of the progress property for operational semantics.

9.3.2. Computational soundness

The second part of the soundness of a denotational semantics is the *computational soundness*. It states that, if a well-typed expression reduces to another expression, then their semantics are equal. However, this is where our semantics presents a major issue.

Consider the following function, which constructs a pair of identical components: $\texttt{mkPair} = \lambda x{:}\mathbb{1}.\,(x, x)$. Its semantics can be computed by essentially taking every denotation $d \in \mathscr{D}$, and replacing x by d. That is, its semantics consists in all finite relations containing pairs of the form $(d, (d, d))$, for every $d \in \mathscr{D}$. This is perfectly sound for values that admit only one denotation. For example, consider the application $(\texttt{mkPair}\,3)$: since the denotational semantics of 3 is the singleton $\{3\}$, the semantics of the application is the singleton $\{(3, 3)\}$, which correctly model the fact that the application reduces to the pair $(3, 3)$.

However, consider now the identity function $\texttt{id} = \lambda x{:}\mathbb{1}.\,x$ (or any value with multiple denotations, for that matter). The semantics of the application $(\texttt{mkPair}\,\texttt{id})$ consists in *all* pairs (d, d) where d is a denotation of $\texttt{id}$. Yet, the application reduces to the pair $(\texttt{id}, \texttt{id})$, whose semantics comprises *all* pairs of the form (d_1, d_2) such that both d_1 and d_2 are denotations of $\texttt{id}$ (it is the

cartesian product of the semantics of `id` with itself). This clearly demonstrates that computational soundness does not hold for our semantics, simply because our semantics does not allow us to bind multiple denotations to the same variable, even though these denotations could come from the same value.

Remark 9.9.
In light of this explanation, one could think of imposing an additional restriction on the calculus, such as requiring all terms to be affine*, that is, such that every variable bound in a λ-abstraction appears at most once in its body. This, however, does not solve the problem, as highlighted by the following, more complicated example: if $\mathbf{e} = \lambda x{:}\mathbb{1}.\, \lambda y{:}\mathbb{1}.\, x$, then the semantics of the application $\mathbf{e}\ \texttt{id}$ contains only relations of the form $\{(d_i, \partial) \mid i \in I\}$, that is, relations where the output is always the same. However, it reduces to $\lambda y{:}\mathbb{1}.\, \texttt{id}$ whose semantics contains all relations of the form $\{(d_i, \partial_i) \mid i \in I\}$ where ∂_i denotes the identity function.*

We will show how to solve this problem in the next chapter. Nevertheless, we can still state a weak version of the computational soundness, using set-containment instead of equality:

Theorem 9.10 (Weak computational soundness for λ_{F})**.** *For every term $\mathbf{e} \in \mathsf{Terms}$ and every type environment $\Gamma \in \mathsf{TEnvs}$, if $\Gamma \vdash \mathbf{e} : t$ and $\mathbf{e} \rightsquigarrow \mathbf{e}'$ then for every $\rho \in [\![\Gamma]\!]$, $[\![\mathbf{e}]\!]_\rho \subseteq [\![\mathbf{e}']\!]_\rho$.*

Notice the requirement for the reduced expression to be well-typed. This is because ill-typed expressions can reduce to well-typed expressions, thus eliminating Ω from their semantics. For example, the semantics of the application $(\lambda x{:}\mathsf{Int}.\, x)\ \texttt{true}$ is the singleton $\{\Omega\}$. However, it reduces to `true` by β-reduction, whose semantics is the singleton $\{\texttt{true}\}$.

This theorem immediately yields the following corollary, further emphasizing the fact that the denotations of a program correspond to the value it produces.

Corollary 9.11. *For every term $\mathbf{e} \in \mathsf{Terms}$ and every environment $\Gamma \in \mathsf{TEnvs}$, if $\Gamma \vdash \mathbf{e} : t$ and $\mathbf{e} \rightsquigarrow^* \mathbf{v}$ then for every $\rho \in [\![\Gamma]\!]$, $[\![\mathbf{e}]\!]_\rho \subseteq [\![\mathbf{v}]\!]_\rho$.*

In the following chapter, we will modify the interpretation domain to allow variables to be bound to multiple elements, and we will prove the above results in equality instead of set-containment.

9.4. Computational adequacy

Soundness is just one side of the relation between operational and denotational semantics, since it shows that when an expression converges to a value, then the expression denotes approximations of that value. It does not provide any information about diverging expressions. This is the goal of the property of *computational adequacy*. It states that if an expression diverges, then its denotation is undefined. In our case, this means that its denotation is empty. Formally, this is expressed by the following theorem:

Theorem 9.12 (Computational adequacy for λ_{F})**.** *For every term $\mathbf{e} \in \mathsf{Terms}$ and every environment $\Gamma \in \mathsf{TEnvs}$, if $\Gamma \vdash \mathbf{e} : t$ and $\mathbf{e}$ diverges then for every $\rho \in [\![\Gamma]\!]$, $[\![\mathbf{e}]\!]_\rho = \emptyset$.*

Together with the computational soundness, it fully encompasses the fact that our denotational semantics properly simulates the operational semantics of λ_{F}.

While this theorem may seem trivial in some calculi, it is not the case in our. This is due to the presence of recursive types. Let D be the smallest type that satisfies the equation $D = D \to D$ (or, using standard μ notation, $D = \mu\alpha.\alpha \to \alpha$). This type satisfies all the conditions for it to be a valid recursive type in our system. Now consider the function $\omega = \lambda x{:}D.\, x\, x$: it is well-typed in our system, of type $D \to D$ (or simply D, which is equivalent). Thus, $\omega\, \omega$ is well-typed of type D and diverges. Since, in the semantic subtyping framework, an arrow type is never empty, it holds that $D \neq \mathbb{0}$. Therefore, computational adequacy cannot be proven by showing that all diverging expressions have type $\mathbb{0}$.

The proof of computational adequacy is far from obvious, and we prove it by adapting the technique of *logical relations* to our system. As for the previous theorems, we will not explain the proof in this section, instead redirecting the reader to the next chapter.

Chapter 10.

A second approach to the semantics of λ_{F}

"Pour examiner la vérité il est besoin, une fois dans sa vie, de mettre toutes choses en doute autant qu'il se peut."

René Descartes, *Les Principes de la philosophie*

In this chapter, we extend the previous denotational semantics to solve the problem arising from its computational soundness. The calculus stays the same, however we perform several changes to the interpretation domain, and adapt the semantics accordingly.

Chapter Outline

Section 10.1 We present a new interpretation domain $\mathscr{D}^{\mathrm{F}}$ to interpret the terms of λ_{F}, and give a new interpretation of types into this domain. We then briefly recall some important information about the functional core calculus λ_{F}, and provide its full denotational semantics in $\mathscr{D}^{\mathrm{F}}$.

Section 10.2 We state several properties of our semantics relating the two interpretation domains $\mathscr{D}^{\mathrm{F}}$ and $\mathscr{D}$. This allows us to relate the semantics of λ_{F} presented in Chapter 9 with the one presented in this chapter. We also show that the subtyping relations induced by the two interpretations are equivalent.

Section 10.3 We state and prove the two main properties of our semantics, namely the full computational soundness and the computational adequacy. We show how to use a technique inspired by logical relations to prove the computational adequacy for our semantics.

10.1. The new semantics of λ_{F}

In the previous chapter, we explained that, to ensure the well-foundedness of our semantics and its interpretation domain, we chose to denote λ-abstractions using possibly infinite sets of finite approximations. While this allowed us to deduce type-sound and adequate semantics for our functional core calculus, we demonstrated in Subsection 9.3.2 that our semantics only satisfied a very weak version of the computational soundness property.

The idea that a value can be represented by its finite approximations is intuitively sound: in a non-diverging program (and therefore, in a program whose semantics is non-empty), every value can only be used a finite number of times, and thus can be finitely approximated. Similarly, since every variable can only be used a finite number of times, it should always be possible to denote the argument of an application by a single, large enough finite approximation to deduce its result. As an example, consider, once again, the identity function $\texttt{idInt} = \lambda x{:}\mathsf{Int}.\, x$, and the pair $(\texttt{idInt}\ 2, \texttt{idInt}\ 3)$. The semantics of $\texttt{idInt}\ 2$ is the singleton $\{2\}$, which can be deduced by

considering the finite approximation $\{(2,2)\}$ of `idInt`. Similarly, the finite approximation $\{(3,3)\}$ of `idInt` yields that the semantics of `idInt 3` is $\{3\}$. Therefore, the semantics of the whole pair can be computed using *only* the finite approximation $\{(2,2),(3,3)\}$ of `idInt`. This is a general principle of our semantics: if two relations are denotations of the same function, then their union is also a denotation of this function.

This last fact is correctly handled by the semantics given in Chapter 9. However, what our semantics does not take into account is the fact that, if a relation is a denotation of a function, then so can be some sub-relations. Indeed, as $\{(2,2),(3,3)\}$ is an approximation of `idInt`, so are $\{\}$, $\{(2,2)\}$ and $\{(3,3)\}$. Going back to the example we presented in the last chapter, consider the function `mkPair` $= \lambda x{:}\mathbb{1}.\,(x,x)$, and suppose that we bind x to $\{(2,2),(3,3)\}$. Our semantics only deduces for (x,x) the denotation $(\{(2,2),(3,3)\},\{(2,2),(3,3)\})$. However, if we bind x to, say, the set containing all the approximations $\{\}$, $\{(2,2)\}$, $\{(3,3)\}$ and $\{(2,2),(3,3)\}$, then we can deduce that every combination of these such as $(\{(2,2)\},\{(3,3)\})$ is a denotation of (x,x). Intuitively, since a β-reduction binds a variable to a value, from a denotational point of view the variable must be bound to the set of approximations denoting the value. However, for cardinality reasons (which we presented in Subsection 2.1.3), the inputs of a relation must be kept finite, hence we only bind variables to finite subsets of the approximations denoting a value.

This is the crux of the problem with the semantics presented in Chapter 9. To solve it, we need to allow functions to manipulate sets of denotations, so that the same variable can be interpreted as multiple denotations.

10.1.1. Changing the domain

As we hinted at in the introduction, the solution we propose is to modify the interpretation of functions so that each finite approximation maps *finite sets* of denotations (rather than single denotations) into denotations. Intuitively, these finite sets represent a set of possible approximations of the input. Of course we consider only non-empty finite sets, since there is no value whose semantics is empty. [1] In order to do that, we define a new domain $\mathscr{D}^{\mathrm{F}}$ to interpret the terms of λ_{F}.

Definition 10.1 (Interpretation domain for λ_{F})**.** *The* interpretation domain $\mathscr{D}^F$ *is the set of finite terms d produced inductively by the following grammar*

$$
\begin{aligned}
&d ::= c \mid (d,d) \mid \{(S,\partial),\ldots,(S,\partial)\} \\
&S ::= \{d,\ldots,d\} \qquad\qquad (S \text{ not empty and finite}) \\
&\partial ::= d \mid \Omega
\end{aligned}
$$

where c ranges over the set $\mathscr{C}$ of constants and where Ω is such that $\Omega \notin \mathscr{D}^F$.
We also write $\mathscr{D}^F_\Omega = \mathscr{D}^F \cup \{\Omega\}$.

Notice how we did not impose any restriction on S: it can be any set of denotations. However, two denotations are not necessarily compatible, in the sense that they cannot always denote the same value. For example, no value can be denoted by both 3 and $\{\}$, since no value is both an integer and a function. This is not a problem in our semantics since such sets can simply be ignored.

[1] Confusion must be avoided between a diverging expression—whose semantics is empty—with the expression diverging on all its arguments—whose semantics is the singleton containing the empty set, and is thus not empty.

Technically, we could stop here and start defining the denotational semantics of λ_{F} using $\mathscr{D}^{\mathrm{F}}$. However, types and terms would then be interpreted in two different domains ($\mathscr{D}^{\mathrm{F}}$ for terms and $\mathscr{D}$ for types). This would make reasoning about our semantics much harder. Therefore, we show how to interpret types into this new domain $\mathscr{D}^{\mathrm{F}}$, so that the new interpretation induces the same subtyping relation as the old one of Definition 2.5 induced by $\mathscr{D}$.

The interpretation of set-theoretic types into this new domain $\mathscr{D}^{\mathrm{F}}$ is obtained by a really straightforward modification of Definition 2.5: all we have to do is to modify just the interpretation of arrow types by replacing the input element d by an input set $S \in \mathscr{P}_f(\mathscr{D}^{\mathrm{F}})$, and use intersection instead of membership. For the sake of concision, we use $\mathscr{F}$ to denote $\mathscr{P}_f(\mathscr{D}^{\mathrm{F}})$, the set of finite parts of $\mathscr{D}^{\mathrm{F}}$. Intuitively, it boils down to defining the interpretation so that the following equation holds:

$$[\![t_1 \to t_2]\!]^{\mathrm{F}} = \{R \in \mathscr{P}_f(\mathscr{F} \times \mathscr{D}^{\mathrm{F}}_\Omega) \mid \forall (S,\partial) \in R.\ S \cap [\![t_1]\!]^{\mathrm{F}} \neq \emptyset \implies \partial \in [\![t_2]\!]^{\mathrm{F}}\}$$

One might wonder why we chose to use the condition $S \cap [\![t_1]\!]^{\mathrm{F}} \neq \emptyset$ rather than the apparently much simpler condition $S \subseteq [\![t_1]\!]^{\mathrm{F}}$ in this equation. The reason is that the latter does not induce the same subtyping relation as the one of Definition 2.5. This is discussed in Section 10.2 and in particular in Remark 10.9.

Of course, due to recursive types, we cannot define this new interpretation inductively on types only using this equation, so we provide the following definition:

Definition 10.2 (Set-theoretic interpretation of types in $\mathscr{D}^{\mathrm{F}}$)**.** *We define a binary predicate $(d : t)^F$ ("the element d belongs to the type t") where $d \in \mathscr{D}^F$ and $t \in$ Types, by induction on the pair (d,t) ordered lexicographically. The predicate is defined as follows:*

$$\begin{aligned}
(c : b)^F &= c \in \mathbb{B}(b) \\
((d_1,d_2) : t_1 \times t_2)^F &= (d_1 : t_1)^F \textit{ and } (d_2 : t_2)^F \\
(\{(S_1,\partial_1),\ldots,(S_n,\partial_n)\} : t_1 \to t_2)^F &= \forall i \in \{1..n\}.\ \textit{if } \exists d \in S_i.\ (d : t_1)^F \textit{ then } (\partial_i : t_2)^F \\
(d : t_1 \vee t_2)^F &= (d : t_1)^F \textit{ or } (d : t_2)^F \\
(d : \neg t)^F &= \textit{ not } (d : t)^F \\
(\partial : t)^F &= \textit{ false} \qquad\qquad \textit{otherwise}
\end{aligned}$$

We define the set-theoretic interpretation $[\![.]\!]^F$: Types $\to \mathscr{P}(\mathscr{D}^F)$ *as* $[\![t]\!]^F = \{d \in \mathscr{D}^F \mid (d : t)^F\}$.

As we already declared, both this interpretation and the interpretation of Definition 2.5 induce the same subtyping relation, as defined in Definition 2.6 (that is, $[\![t]\!]^{\mathrm{F}} = \emptyset \iff [\![t]\!] = \emptyset$). Although the differences between the two models are minimal, proving they induce the same subtyping relation is not straightforward. We will discuss and prove this fact later on, in Section 10.2.

10.1.2. A new interpretation of λ-abstractions

Having defined the new domain $\mathscr{D}^{\mathrm{F}}$ and the new interpretation of types into this domain $[\![.]\!]^{\mathrm{F}}$, we can now interpret terms of λ_{F} into elements of $\mathscr{D}^{\mathrm{F}}$. Naturally, the major change between the new semantic interpretation $[\![.]\!]^{\mathrm{F}}_{(.)}$ and the semantics presented in Definition 9.5 will be the interpretation of λ-abstractions. However, as for the interpretation of types, the semantics of

λ-abstractions is obtained by a straightforward modification of Definition 9.5: we replace inputs $d \in \mathscr{D}$ by sets $S \in \mathscr{F}$, and use containment instead of membership:

$$[\![\lambda x{:}t.\,\mathbf{e}]\!]^{\mathrm{F}}_{\rho} \quad = \quad \{R \in \mathscr{P}_f(\mathscr{F} \times \mathscr{D}_\Omega) \mid \forall (S,\partial) \in R, \quad \begin{array}{rl} \text{either} & S \subseteq [\![t]\!]^{\mathrm{F}} \text{ and } \partial \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho, x \mapsto S} \\ \text{or} & S \subseteq [\![\neg t]\!]^{\mathrm{F}} \text{ and } \partial = \Omega\} \end{array}$$

This formula features several important changes compared to the formula given in Definition 9.5. In particular, we now use two conjunctions instead of two implications. The main reason is that, in Definition 9.5, the conditions $d \in [\![t]\!]$ and $d \notin [\![t]\!]$ where complementary: one and only one of them always holds. However, since we now manipulate sets, it is also possible for a set to intersect both $[\![t]\!]^{\mathrm{F}}$ and $[\![\neg t]\!]^{\mathrm{F}}$, and therefore not to be contained in any of the two. We want to avoid such sets, which we do by ensuring one of the two inclusions holds.

Remark 10.3.
We could have followed the same strategy as for the interpretation of arrow types, by defining the semantics of λ-abstractions as:

$$[\![\lambda x{:}t.\,\mathbf{e}]\!]^{F}_{\rho} \quad = \quad \{R \in \mathscr{P}_f(\mathscr{F} \times \mathscr{D}_\Omega) \mid \forall (S,\partial) \in R, \quad \begin{array}{rl} S \cap [\![t]\!]^{F} \neq \emptyset & \implies \partial \in [\![\mathbf{e}]\!]^{F}_{\rho, x \mapsto S \cap [\![t]\!]^{F}} \\ S \cap [\![t]\!]^{F} = \emptyset & \implies \partial = \Omega\} \end{array}$$

It is actually possible to prove the soundness and adequacy of the semantics using this definition. However, it seems counter-intuitive that, for a given function $\lambda x{:}t.\,\mathbf{e}$, this formula allows some sets S that are not subsets of $[\![t]\!]^{F}$ to be mapped into elements distinct from Ω. We chose to emphasize the importance of type annotations by forbidding such sets.

Also notice that, instead of assigning a single denotation to every variable, environments now map variables to sets of denotations. Thus, we now modify the definition of semantic environments to reflect this change.

Definition 10.4 (Semantic environments)**.** *A semantic environment for λ_F is a function* $\mathsf{Vars} \to \mathscr{P}_f(\mathscr{D})$. *We use* Envs^F *to denote the set of such environments, and use ρ to range over this set:*

$$\mathsf{Envs}^F \ni \rho : \mathsf{Vars} \to \mathscr{P}_f(\mathscr{D})$$

Moreover, given a variable $x \in \mathsf{Vars}$ and a set $S \in \mathscr{P}_f(\mathscr{D})$, we use the notation $\rho, x \mapsto S$ to denote the environment obtained by extending ρ with a mapping from x to S.

Finally, it remains to change the semantics of applications to account for the fact that relations now take sets as input. Once again, this is only a matter of using containment instead of membership in Definition 9.5:

$$[\![\mathbf{e}_1\,\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho} = \{\partial \in \mathscr{D}^{\mathrm{F}}_{\Omega} \mid \exists R \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho}, \exists S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho}.\ (S,\partial) \in R\} \cup \Omega^{\rho}_{\mathbf{e}_1\,\mathbf{e}_2}$$

This is where our remark about the absence of restriction on the elements of S in Definition 10.2 comes into play. Nothing prevents us from saying that the function $\texttt{mkPair} = \lambda x{:}\mathbb{1}.\,(x,x)$ maps the set of denotations $\{2,3\}$ into the pair $(2,3)$, for example. However, no expression will ever admit both 2 and 3 as denotations, therefore, this mapping will never be used in our semantics.

10.1.3. New denotational semantics of λ_{F}

We can now present the new semantics of λ_{F}. The terms and type system of this calculus have already been presented in Chapter 9. As a reminder, the reduction rules are as follows.

$$\begin{array}{rrcll} [\mathrm{R}^{\mathrm{F}}_{\mathrm{app}}] & (\lambda x{:}t.\,\mathbf{e})\,\mathbf{v} & \rightsquigarrow & \mathbf{e}\,[\mathbf{v}/x] & \\ [\mathrm{R}^{\mathrm{F}}_{\mathrm{proj}_i}] & \pi_i\,(\mathbf{v}_1,\mathbf{v}_2) & \rightsquigarrow & \mathbf{v}_i & \text{for } i \in \{1,2\} \\ [\mathrm{R}^{\mathrm{F}}_{\mathrm{ctx}}] & \mathscr{E}\,[\mathbf{e}] & \rightsquigarrow & \mathscr{E}\,[\mathbf{e}'] & \text{if } \mathbf{e} \rightsquigarrow \mathbf{e}' \end{array}$$

Integrating the changes we presented in the previous subsection, the denotational semantics of λ_{F} in $\mathscr{D}^{\mathrm{F}}$ is defined as follows:

Definition 10.5 (Set-theoretic interpretation of λ_{F} in $\mathscr{D}^{\mathrm{F}}$)**.** *Let $\rho \in \mathsf{Envs}^F$. We define the set-theoretic interpretation of λ_F as a function $[\![.]\!]^F_{(.)} : \mathsf{Terms} \to \mathsf{Envs}^F \to \mathscr{P}_f(\mathscr{D}^F_\Omega)$ as follows:*

$$\begin{array}{rcl} [\![x]\!]^F_\rho & = & \rho(x) \\ [\![c]\!]^F_\rho & = & \{c\} \\ [\![\lambda x{:}t.\,\mathbf{e}]\!]^F_\rho & = & \{R \in \mathscr{P}_f(\mathscr{F} \times \mathscr{D}^F_\Omega) \mid \forall (S,\partial) \in R, \quad \textit{either} \quad S \subseteq [\![t]\!]^F \quad \textit{and } \partial \in [\![\mathbf{e}]\!]^F_{\rho,x\mapsto S} \\ & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{or} \quad S \subseteq [\![\neg t]\!]^F \quad \textit{and } \partial = \Omega\} \\ [\![\mathbf{e}_1\,\mathbf{e}_2]\!]^F_\rho & = & \{\partial \in \mathscr{D}^F_\Omega \mid \exists S \subseteq [\![\mathbf{e}_2]\!]^F_\rho, R \in [\![\mathbf{e}_1]\!]^F_\rho, (S,\partial) \in R\} \cup \Omega^\rho_{\mathbf{e}_1\,\mathbf{e}_2} \\ [\![\pi_i\,\mathbf{e}]\!]^F_\rho & = & \{d_i \mid (d_1,d_2) \in [\![\mathbf{e}]\!]^F_\rho\} \cup \Omega^\rho_{\pi_i\,\mathbf{e}} \\ [\![(\mathbf{e}_1,\mathbf{e}_2)]\!]^F_\rho & = & ([\![\mathbf{e}_1]\!]^F_\rho \setminus \{\Omega\}) \times ([\![\mathbf{e}_2]\!]^F_\rho \setminus \{\Omega\}) \cup \Omega^\rho_{(\mathbf{e}_1,\mathbf{e}_2)} \end{array}$$

Apart from the changes in the interpretation of λ-abstractions and applications we presented earlier, the denotational semantics of λ_{F} in $\mathscr{D}^{\mathrm{F}}$ mostly identical to its semantics in $\mathscr{D}$ presented in Chapter 9.

The only remaining change is that, now that environments map variables to sets of denotations, the interpretation of a variable is now simply the set it is bound to in the current environment (instead of a singleton as in Definition 9.5). The operator $\Omega^{(.)}_{(.)}$ is defined as in Definition 9.3, except using the new domain $\mathscr{D}^{\mathrm{F}}$ and semantics $[\![.]\!]^{\mathrm{F}}_{(.)}$.

10.2. Basic properties

Before proving the computational soundness and adequacy of the denotational semantics of λ_{F}, we present several properties formalizing the correspondence between the domains $\mathscr{D}$ and $\mathscr{D}^{\mathrm{F}}$, as well as between the semantics presented in this chapter and in Chapter 9.

10.2.1. Equivalence of subtyping

We currently have two interpretations of types: one interpreting types as sets of elements of $\mathscr{D}$, the other interpreting types as sets of elements of $\mathscr{D}^{\mathrm{F}}$. However, as we stated earlier, these two interpretations actually induce the same subtyping relation. This subsection focuses on the proof of this fact, which is not straightforward.

As a first step, we define two functions $I : \mathscr{D}_\Omega \to \mathscr{D}^{\mathrm{F}}_\Omega$ and $B : \mathscr{D}^{\mathrm{F}}_\Omega \to \mathscr{D}_\Omega$, which we will use as a bridge between the two domains. The function I defines the *injection* of an element $\mathscr{D}_\Omega$ into $\mathscr{D}^{\mathrm{F}}_\Omega$. The intuition behind it is simple: if a relation $R \in \mathscr{D}_\Omega$ maps an element d into a result ∂, then we can consider it as a relation of $\mathscr{D}^{\mathrm{F}}_\Omega$ mapping the singleton $\{I(d)\}$ into the injection of the result $I(\partial)$.

Definition 10.6 (Injection of $d \in \mathscr{D}_\Omega$)**.** *We define the injection function $I : \mathscr{D}_\Omega \to \mathscr{D}_\Omega^F$ by induction as follows:*

$$\begin{aligned} I(c) &= c \\ I((d_1, d_2)) &= (I(d_1), I(d_2)) \\ I(\{(d_1, \partial_1), \ldots, (d_n, \partial_n)\}) &= \{(\{I(d_1)\}, I(\partial_1)), \ldots, (\{I(d_n)\}, I(\partial_n))\} \\ I(\Omega) &= \Omega \end{aligned}$$

The second function, which we call the *basis function* $B : \mathscr{D}_\Omega^{\mathrm{F}} \to \mathscr{D}_\Omega$, is slightly more complex. The intuition behind it is to interpret a relation mapping a set S of denotations into a result ∂ as a relation mapping every $d \in S$ into the same ∂. Of course, this may produce non-deterministic relations: for example, the basis of a relation mapping $\{2\}$ into 2 and $\{2, 3\}$ into 3 will be a non-deterministic relation mapping 2 into both 2 and 3 (and 3 into 3). However, this is not a problem as this is not forbidden by our interpretation of types.

Definition 10.7 (Basis of $d \in \mathscr{D}_\Omega^{\mathrm{F}}$)**.** *We define the basis function $B : \mathscr{D}_\Omega^F \to \mathscr{D}_\Omega$ by induction as follows:*

$$\begin{aligned} B(c) &= c \\ B((d_1, d_2)) &= (B(d_1), B(d_2)) \\ B(\{(S_1, \partial_1), \ldots, (S_n, \partial_n)\}) &= \bigcup_{i \in \{1..n\}} \bigcup_{d_i \in S_i} \{(B(d_i), B(\partial_i))\} \\ B(\Omega) &= \Omega \end{aligned}$$

As a remark, it is straightforward to verify that I maps elements of $\mathscr{D}$ into elements of $\mathscr{D}^{\mathrm{F}}$, and that B maps elements of $\mathscr{D}^{\mathrm{F}}$ into elements of $\mathscr{D}$.

The next step is to show that these functions preserve type membership. That is, $d \in [\![t]\!]$ if and only if $I(d) \in [\![t]\!]^{\mathrm{F}}$, and conversely for B. Since subtyping is defined as set-containment, this will ensure that the two definitions of subtyping are equivalent.

Lemma 10.8. *For every $t \in$ Types,*

1. $\forall d \in \mathscr{D}, d \in [\![t]\!] \iff I(d) \in [\![t]\!]^F$
2. $\forall d \in \mathscr{D}^F, d \in [\![t]\!]^F \iff B(d) \in [\![t]\!]$

Proof. Both results are proven by induction on the pair (d, t) lexicographically ordered, following the inductive definition of $(d : t)$ and $(d : t)^{\mathrm{F}}$. For both statements, we prove the whole equivalence using a single induction (rather than separating the two implications), since the equivalence is needed to handle the case of negation types by induction. Moreover, we first handle the cases where t is a union or a negation, to eliminate cases that hold vacuously (for example, if $((d_1, d_2) : t)$ and t is not a union nor a negation, then t is necessary a product).

1. • $(d, t_1 \vee t_2)$. Suppose that $d \in [\![t_1 \vee t_2]\!]$. By definition, there exists $i \in \{1, 2\}$ such that $d \in [\![t_i]\!]$. By induction hypothesis, $I(d) \in [\![t_i]\!]^{\mathrm{F}}$ and thus $I(d) \in [\![t_1 \vee t_2]\!]^{\mathrm{F}}$. The same reasoning can be made if $I(d) \in [\![t_1 \vee t_2]\!]^{\mathrm{F}}$ to prove that $d \in [\![t_1 \vee t_2]\!]$.

- $(d, \neg t)$. Suppose that $d \in [\![\neg t]\!]$. By definition, this means that $d \notin [\![t]\!]$. Using the induction hypothesis, we deduce that $I(d) \notin [\![t]\!]^{\mathrm{F}}$, thus $I(d) \in [\![\neg t]\!]^{\mathrm{F}}$.
  Conversely, if $I(d) \in [\![\neg t]\!]^{\mathrm{F}}$, the same reasoning can be used to prove that $d \in [\![\neg t]\!]$.
- (c, b). Suppose that $c \in [\![t]\!]$, that is, $c \in \mathbb{B}(b)$. By definition, $I(c) = c$, and $[\![b]\!]^{\mathrm{F}} = [\![b]\!]$ thus $I(c) \in [\![b]\!]^{\mathrm{F}}$. The same reasoning can be used to prove the converse.
- $((d_1, d_2), t_1 \times t_2)$. Suppose that $(d_1, d_2) \in [\![t_1 \times t_2]\!]$. Let $i \in \{1, 2\}$. By construction, $d_i \in [\![t_i]\!]$. By induction, $I(d_i) \in [\![t_i]\!]^{\mathrm{F}}$. Thus, $I((d_1, d_2)) = (I(d_1), I(d_2)) \in [\![t_1 \times t_2]\!]^{\mathrm{F}}$. The same reasoning can be used to prove the converse.
- $(\{(d_1, \partial_1) \dots (d_n, \partial_n)\}, t_1 \to t_2)$. Suppose that $\{(d_1, \partial_1) \dots (d_n, \partial_n)\} \in [\![t_1 \to t_2]\!]$. Let $i \in [1..n]$. If $I(d_i) \in [\![t_1]\!]^{\mathrm{F}}$, then by induction, $d_i \in [\![t_1]\!]$. Thus, by hypothesis, $\partial_i \in [\![t_2]\!]$, and by induction, $I(\partial_i) \in [\![t_2]\!]^{\mathrm{F}}$. Therefore, $I(\{(d_1, \partial_1) \dots (d_n, \partial_n)\}) = \{(\{I(d_1)\}, I(\partial_1)) \dots (\{I(d_n)\}, I(\partial_n))\} \in [\![t_1 \to t_2]\!]^{\mathrm{F}}$. The converse can be proven using the same reasoning.

2.
- $(d, t_1 \vee t_2)$. Suppose that $d \in [\![t_1 \vee t_2]\!]^{\mathrm{F}}$. By definition, there exists $i \in \{1, 2\}$ such that $d \in [\![t_i]\!]^{\mathrm{F}}$. By induction hypothesis, $B(d) \in [\![t_i]\!]$ and thus $B(d) \in [\![t_1 \vee t_2]\!]$. The same reasoning can be made if $B(d) \in [\![t_1 \vee t_2]\!]$ to prove that $d \in [\![t_1 \vee t_2]\!]^{\mathrm{F}}$.
- $(d, \neg t)$. Suppose that $d \in [\![\neg t]\!]^{\mathrm{F}}$. By definition, this means that $d \notin [\![t]\!]^{\mathrm{F}}$. Using the induction hypothesis, we deduce that $B(d) \notin [\![t]\!]$, thus $B(d) \in [\![\neg t]\!]$.
  Conversely, if $B(d) \in [\![\neg t]\!]$, the same reasoning can be used to prove that $d \in [\![\neg t]\!]^{\mathrm{F}}$.
- (c, b). Suppose that $c \in [\![t]\!]^{\mathrm{F}}$, that is, $c \in \mathbb{B}(b)$. By definition, $B(c) = c$, and $[\![b]\!]^{\mathrm{F}} = [\![b]\!]$ thus $B(c) \in [\![b]\!]$. The same reasoning can be used to prove the converse.
- $((d_1, d_2), t_1 \times t_2)$. Suppose that $(d_1, d_2) \in [\![t_1 \times t_2]\!]^{\mathrm{F}}$. Let $i \in \{1, 2\}$. By construction, $d_i \in [\![t_i]\!]^{\mathrm{F}}$. By induction, $B(d_i) \in [\![t_i]\!]$. Thus, $B((d_1, d_2)) = (B(d_1), B(d_2)) \in [\![t_1 \times t_2]\!]$. The same reasoning can be used to prove the converse.
- $(\{(S_1, \partial_1) \dots (S_n, \partial_n)\}, t_1 \to t_2)$. Let us write $R = \{(S_1, \partial_1) \dots (S_n, \partial_n)\}$ and suppose that $R \in [\![t_1 \to t_2]\!]^{\mathrm{F}}$. Let $(d, \partial) \in B(R)$, and suppose that $d \in [\![t_1]\!]$. We have to show that $\partial \in [\![t_2]\!]$. By definition of the basis, there exists $i \in [1..n]$ and $d_i \in S_i$ such that $d = B(d_i)$ and $\partial = B(\partial_i)$. By induction hypothesis, we deduce that $d_i \in [\![t_1]\!]^{\mathrm{F}}$. Thus, $[\![t_1]\!]^{\mathrm{F}} \cap S_i \neq \emptyset$ and by definition $\partial_i \in [\![t_2]\!]^{\mathrm{F}}$. By induction hypothesis, this ensures that $\partial = B(\partial_i) \in [\![t_2]\!]$.
  Conversely, if $B(R) \in [\![t_1 \to t_2]\!]$, take any $i \in [1..n]$ such that $S_i \cap [\![t_1]\!]^{\mathrm{F}} \neq \emptyset$. Let $d_i \in S_i \cap [\![t_1]\!]^{\mathrm{F}}$. By definition of the basis, $(B(d_i), B(\partial_i)) \in B(R)$, and by induction hypothesis, $B(d_i) \in [\![t_1]\!]$. Thus, by definition, this yields $B(\partial_i) \in [\![t_2]\!]$ and by induction hypothesis, $\partial_i \in [\![t_2]\!]^{\mathrm{F}}$, hence the result.

□

Remark 10.9.

This proof highlights the reason we defined the interpretation of arrow types using an intersec-

tion:

$$[\![t_1 \to t_2]\!]^F = \{R \in \mathscr{P}_f(\mathscr{F} \times \mathscr{D}_\Omega^F) \mid \forall (S, \partial) \in R.\ S \cap [\![t_1]\!]^F \neq \emptyset \implies \partial \in [\![t_2]\!]^F\}$$

rather than using containment, which would have been more intuitive:

$$[\![t_1 \to t_2]\!]^F = \{R \in \mathscr{P}_f(\mathscr{F} \times \mathscr{D}_\Omega^F) \mid \forall (S, \partial) \in R.\ S \subseteq [\![t_1]\!]^F \implies \partial \in [\![t_2]\!]^F\}$$

However, this second interpretation does not preserves the subtyping relation. For a counter example, consider the types $t_1 = (\mathsf{True} \to \mathsf{True}) \wedge (\mathsf{False} \to \mathsf{True})$ *and* $t_2 = \mathsf{Bool} \to \mathsf{True}$. *It is straightforward to verify that, for the interpretation of Definition 2.5,* $t_1 \simeq t_2$. *However, consider the relation* $R = \{(\{\texttt{true}, \texttt{false}\}, \texttt{false})\}$. *Since* $\{\texttt{true}, \texttt{false}\} \not\subseteq [\![\mathsf{True}]\!]^F$, *it would hold that* $R \in [\![\mathsf{True} \to \mathsf{True}]\!]^F$, *should we use the second formula above. Similarly, using the same reasoning,* $R \in [\![\mathsf{False} \to \mathsf{True}]\!]^F$ *also holds. Thus, this would mean that* $R \in [\![t_1]\!]^F$. *However, since* $\{\texttt{true}, \texttt{false}\} \subseteq [\![\mathsf{Bool}]\!]^F$, *and* $\texttt{false} \notin [\![\mathsf{True}]\!]^F$, *it would not hold that* $R \in [\![t_2]\!]^F$. *Hence,* t_1 *would not be a subtype of* t_2 *for the new interpretation.* ⌟

Finally, using the two functions I and B, as well as the previous lemma, we can easily prove the equivalence of the subtyping relation induced by the two interpretations.

Theorem 10.10 (Equivalence of subtyping over $\mathscr{D}^{\mathrm{F}}$ and $\mathscr{D}$)**.** *For every types* $t_1, t_2 \in \mathsf{Types}$, $[\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!]^F \subseteq [\![t_2]\!]^F$

Proof. Let $t_1, t_2 \in \mathsf{Types}$. We proceed by double implication.

1. Suppose that $[\![t_1]\!] \subseteq [\![t_2]\!]$. Let $d \in [\![t_1]\!]^{\mathrm{F}}$. By Lemma 10.8, this yields $B(d) \in [\![t_1]\!]$. By hypothesis, $B(d) \in [\![t_2]\!]$. And by Lemma 10.8, $d \in [\![t_2]\!]^{\mathrm{F}}$.
2. Suppose that $[\![t_1]\!]^{\mathrm{F}} \subseteq [\![t_2]\!]^{\mathrm{F}}$. Let $d \in [\![t_1]\!]$. By Lemma 10.8, this yields $I(d) \in [\![t_1]\!]^{\mathrm{F}}$. By hypothesis, $I(d) \in [\![t_2]\!]^{\mathrm{F}}$. And by Lemma 10.8, $d \in [\![t_2]\!]$.

□

10.2.2. Relating our two denotational semantics

The only task remaining before proving the soundness and adequacy of our new semantics $[\![.]\!]^{\mathrm{F}}$ for λ_{F} is to relate it to the semantics presented in the previous chapter.

The idea is simple: we show that, for every term $\mathbf{e}$ of λ_{F}, if this term can be denoted by a denotation d using the interpretation of Definition 9.5, then it can be denoted by $I(d)$ using the interpretation of Definition 10.5, where I is the injection function from Definition 10.6.

However, the result must be proven by induction on the given term $\mathbf{e}$, and for the induction hypothesis to be strong enough, it is necessary to generalize the result over arbitrary environments. Therefore, we first need to relate our two definitions of semantic environments. This is straightforward and we achieve it by extending the definition of the function I to environments.

Definition 10.11 (Injection of environments)**.** *We define the* injection of environments *as the function* $I : \mathsf{Envs} \to \mathsf{Envs}^F$ *such that* $I(\rho)(x) = I(\rho(x))$.

We can now use this definition to formalize the property we stated previously.

Theorem 10.12 (Conservativity of the semantics)**.** *For every term* $\mathbf{e} \in \mathsf{Terms}$*, every environment* $\rho \in \mathsf{Envs}$*,*

$$d \in [\![\mathbf{e}]\!]_\rho \implies I(d) \in [\![\mathbf{e}]\!]^F_{I(\rho)}$$

Proof hint. For the full proof, see Theorem A.33 in the Appendix (page 271).
The proof is done by structural induction on $\mathbf{e} \in \mathsf{Terms}$, where the induction hypothesis is generalized over ρ.

- $\underline{\mathbf{e} = c.}$ Immediate.
- $\underline{\mathbf{e} = x.}$ Use Definition 10.11.
- $\underline{\mathbf{e} = \lambda x{:}t.\, \mathbf{e}'.}$ Take $R \in [\![\lambda x{:}t.\, \mathbf{e}']\!]_\rho$ and consider $(S, \partial) \in I(R)$. Definition 10.6 ensures that there exists $(d_0, \partial_0) \in R$ such that $S = \{I(d_0)\}$ and $\partial = I(\partial_0)$. Then distinguish two cases according to whether $d_0 \in [\![t]\!]$ or not, and use Lemma 10.8 and Definition 9.5.
- For all the remaining cases, consider $\partial \in [\![\mathbf{e}]\!]_\rho$, and distinguish the cases where ∂ comes from $\Omega^\rho_{\mathbf{e}}$, and conclude each case by induction hypothesis.

□

Since the reduction rules are the same for both calculi, this result ensures the soundness and adequacy of the semantics presented in Chapter 9 from the corresponding properties of the semantics we present here. In particular, if a term $\mathbf{e}$ diverges, then its semantics $[\![\mathbf{e}]\!]^{\mathrm{F}}_\rho$ must be empty by adequacy of the semantics of Definition 10.5. And the above result guarantees that $[\![\mathbf{e}]\!]_\rho$ is empty too.

10.3. Soundness and adequacy

We can now state and prove the soundness and adequacy of our semantics. We follow the same strategy as in the previous chapter: we start by its type soundness, then follow with its computational soundness, and conclude with its adequacy.

10.3.1. Type soundness

Recall that in Section 9.3, we stated the type soundness of λ_{F} for open terms typed in arbitrary environments. This required the definition of an operation $[\![.]\!]$ which associated to a type environment a set of "compatible" semantic environments. We follow the same strategy here, however, since we changed the definition of semantic environments in the previous section, we also need to modify the definition of $[\![.]\!]$, yielding a new operation $[\![.]\!]^{\mathrm{F}}$.

This new interpretation is straightforward: if a variable x is bound to a type t in an environment Γ, then rather than binding it to a single element d of $[\![t]\!]$, we map it to a subset of $[\![t]\!]^{\mathrm{F}}$. This yields the following definition:

Definition 10.13 (Denotational interpretation of Γ)**.** *Let* $\Gamma \in \mathsf{TEnvs}$*. We define its denota-*

tional interpretation, noted $[\![\Gamma]\!]^F$, as the function

$$[\![.]\!]^F : \mathsf{TEnvs} \to \mathscr{P}(\mathsf{Envs}^F)$$
$$[\![\Gamma]\!]^F = \{\rho \in \mathsf{Envs}^F \mid \forall x \in \mathrm{Dom}(\Gamma).\ \rho(x) \subseteq [\![\Gamma(x)]\!]^F\}$$

Remark 10.14.
According to this definition, there is a "most-general" environment $\rho \in [\![\Gamma]\!]^F$ which is the environment such that for every x, $\rho(x) = [\![\Gamma(x)]\!]^F$. All the results that follow are in particular true for this most-general environment. This once again emphasizes the link between types and denotations.

Using this new definition, we can now state and prove the type soundness theorem for λ_{F}, similarly to Theorem 9.7.

Theorem 10.15 (Type soundness for λ_{F})**.** *For every type environment $\Gamma \in \mathsf{TEnvs}$ and every term $\mathbf{e} \in \mathsf{Terms}$, if $\Gamma \vdash \mathbf{e} : t$ then for every $\rho \in [\![\Gamma]\!]^F$, $[\![\mathbf{e}]\!]^F_\rho \subseteq [\![t]\!]^F$.*

Proof hint. We only give a brief outline of the proof. For the full proof, see Theorem A.34 in the Appendix (page 272).
The proof is done by structural induction on $\mathbf{e} \in \mathsf{Terms}$ generalized over Γ, supposing $\Gamma \vdash \mathbf{e} : t$.

- $\underline{\mathbf{e} = c.}$ By inversion of the typing rules, $b_c \leq t$, therefore $c \in [\![t]\!]^{\mathrm{F}}$.
- $\underline{\mathbf{e} = x.}$ Apply Definition 10.13.
- $\underline{\mathbf{e} = \lambda x{:}t_x.\,\mathbf{e}'.}$ Consider $R \in [\![\lambda x{:}t_x.\,\mathbf{e}']\!]^{\mathrm{F}}_\rho$ and $(S, \partial) \in R$ such that $S \cap [\![t_x]\!]^{\mathrm{F}} \neq \emptyset$. Definition 10.5 ensures that $S \subseteq [\![t_x]\!]^{\mathrm{F}}$ and that $\partial \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho, x \mapsto S}$. Apply the generalized induction hypothesis to $\Gamma, x : t_x \vdash \mathbf{e}' : t_{\mathbf{e}}$ and the semantic environment $(\rho, x \mapsto S)$ to deduce that $\partial \in [\![t_{\mathbf{e}}]\!]^{\mathrm{F}}$.
- $\underline{\mathbf{e}_1\,\mathbf{e}_2.}$ Inversing the typing rules proves that $\Omega^\rho_{\mathbf{e}_1\,\mathbf{e}_2} = \emptyset$. Then consider $\partial \in [\![\mathbf{e}_1\,\mathbf{e}_2]\!]^{\mathrm{F}}_\rho$. By inversion, there exists $R \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_\rho$ and $S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{F}}_\rho$ such that $(S, \partial) \in R$. Apply the induction hypothesis to R and S to deduce the result.
- $\underline{(\mathbf{e}_1, \mathbf{e}_2).}$ Inversing the typing rules proves that $\Omega^\rho_{(\mathbf{e}_1,\mathbf{e}_2)} = \emptyset$. This yields $[\![(\mathbf{e}_1, \mathbf{e}_2)]\!]^{\mathrm{F}}_\rho = [\![\mathbf{e}_1]\!]^{\mathrm{F}}_\rho \times [\![\mathbf{e}_2]\!]^{\mathrm{F}}_\rho$, conclude with the induction hypothesis.
- $\underline{\pi_i\,\mathbf{e}'.}$ Inversing the typing rules proves $\Omega^\rho_{\pi_i\,\mathbf{e}'} = \emptyset$. Then let $d \in [\![\pi_i\,\mathbf{e}']\!]^{\mathrm{F}}_\rho$, and inverse the semantics to prove that there exists $(d_1, d_2) \in [\![\mathbf{e}']\!]^{\mathrm{F}}_\rho$ such that $d = d_i$. Apply the induction hypothesis.

□

Naturally, this theorem immediately yields the same corollary as in the previous chapter, which formalizes the fact that Ω cannot occur in the semantics of a well-typed expression.

Corollary 10.16. *For every type environment $\Gamma \in \mathsf{TEnvs}$ and every term $\mathbf{e} \in \mathsf{Terms}$, if $\Gamma \vdash \mathbf{e} : t$ then for every $\rho \in [\![\Gamma]\!]^F$, $\Omega \notin [\![\mathbf{e}]\!]^F_\rho$.*

10.3.2. Computational soundness

Having proven the type soundness of our semantics, we continue with its computational soundness. Contrary to Chapter 9 which featured a weak version of this property, this time we state it using equality instead of set-containment, which is a much stronger result.

Theorem 10.17 (Computational soundness for λ_{F})**.** *For every term* $\mathbf{e} \in \mathsf{Terms}$ *such that* $\Gamma \vdash \mathbf{e} : t$ *and every environment* $\rho \in [\![\Gamma]\!]^F$*, if* $\mathbf{e} \rightsquigarrow \mathbf{e}'$ *then* $[\![\mathbf{e}]\!]^F_\rho = [\![\mathbf{e}']\!]^F_\rho$.

We cannot tackle the proof of this theorem immediately without first proving two important lemmas, both related to λ-abstractions. When considering a β-reduction $(\lambda x{:}t.\, \mathbf{e})\, \mathbf{v} \rightsquigarrow \mathbf{e}\,[\mathbf{v}/x]$, it is clear that we need some kind of substitution lemma. A first idea could be to prove that $[\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{F}}_\rho = [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho, x \mapsto [\![\mathbf{v}]\!]^{\mathrm{F}}_\rho}$, however, this result, while true, is not strong enough: when computing the semantics of a λ-abstraction, the parameter x is always approximated by a *finite* set of denotations. Therefore, what we have to prove is that every denotation d of $\mathbf{e}\,[\mathbf{v}/x]$ can be obtained by approximating x by a *finite* subset of $[\![\mathbf{v}]\!]^{\mathrm{F}}_\rho$.

This result is proven in two steps: we first prove the monotonicity of $[\![.]\!]^{\mathrm{F}}_{(.)}$ with respect to environments, and then prove the substitution lemma.

Lemma 10.18 (Monotonicity lemma)**.** *For every term* $\mathbf{e} \in \mathsf{Terms}$*,* $x \in \mathsf{Vars}$*,* $\rho \in \mathsf{Envs}^F$*, and* $S_1, S_2 \in \mathscr{P}(\mathscr{D}^F)$*, if* $S_1 \subseteq S_2$ *then* $[\![\mathbf{e}]\!]^F_{\rho, x \mapsto S_1} \subseteq [\![\mathbf{e}]\!]^F_{\rho, x \mapsto S_2}$

Proof hint. See Lemma A.35 in the Appendix (page 273) for the full proof.
The proof is done by a straightforward structural induction on $\mathbf{e} \in \mathsf{Terms}$, generalized over ρ. □

Lemma 10.19 (Substitution lemma)**.** *For every term* $\mathbf{e} \in \mathsf{Terms}$*,* $\mathbf{v} \in \mathsf{Values}$*,* $x \in \mathsf{Vars}$*,* $\rho \in \mathsf{Envs}^F$*,*

$$[\![\mathbf{e}\,[\mathbf{v}/x]]\!]^F_\rho = \bigcup_{S \in \mathscr{P}_f([\![\mathbf{v}]\!]^F_\rho)} [\![\mathbf{e}]\!]^F_{\rho, x \mapsto S}$$

Proof hint. See Lemma A.36 in the Appendix (page 274) for the full proof.
The proof is done by structural induction on $\mathbf{e} \in \mathsf{Terms}$, generalized over ρ.

- $\underline{\mathbf{e} = c.}$ Immediate.
- $\underline{\mathbf{e} = y.}$ If $y \neq x$ the result is immediate. Otherwise, if $y = x$ reason by double inclusion.
- $\underline{\mathbf{e} = \lambda y{:}t.\, \mathbf{e}'.}$ By definition, $[\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{F}}_\rho = [\![\lambda y{:}t.\, (\mathbf{e}'\,[\mathbf{v}/x])]\!]^{\mathrm{F}}_\rho$. Proceed by double inclusion.
  - Let $R \in [\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{F}}_\rho$, and let us write $R = \{(S_i, \partial_i) \mid i \in I\}$. For every $i \in I$, the induction hypothesis yields the existence of a finite set $S_i^{\mathbf{v}} \subseteq [\![\mathbf{v}]\!]^{\mathrm{F}}_\rho$ such that $\partial_i \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho, y \mapsto S_i, x \mapsto S_i^{\mathbf{v}}}$. Then consider the union $S^{\mathbf{v}} = \bigcup_{i \in I} S_i^{\mathbf{v}}$ and apply Lemma 10.18 on this union.
  - The second inclusion is straightforward by a simple application of the induction hypothesis.
- $\underline{\mathbf{e} = \mathbf{e}_1\, \mathbf{e}_2.}$ Reason by double inclusion, and eliminate the cases involving Ω by induc-

tion hypothesis. Then, apply the induction hypothesis on both $\mathbf{e}_1$ and $\mathbf{e}_2$ to deduce the existence of two sets S_1 and S_2 approximating x, and conclude by Lemma 10.18 on $S_1 \cup S_2$.

- The other cases are treated similarly.

□

Equipped with these two lemmas, we can now tackle the proof of Theorem 10.17.

Proof. The proof is done by structural induction on $\mathbf{e} \in \text{Terms}$ and cases over the reduction rule used for $\mathbf{e} \rightsquigarrow \mathbf{e}'$.

- $[\underline{\mathrm{R}^{\mathrm{F}}_{\mathrm{app}}}]$. $(\lambda x{:}t.\,\mathbf{e})\,\mathbf{v} \rightsquigarrow \mathbf{e}\,[\mathbf{v}/x]$. Since $\mathbf{e}$ is well-typed, by inversion of the typing rules, we have $\Gamma \vdash \mathbf{v} : t$. We then proceed by double inclusion.
  - Note that, by Theorem 10.15, it holds that $\Omega \notin [\![\mathbf{v}]\!]^{\mathrm{F}}_{\rho}$. Moreover, by Definition 10.5, $[\![\lambda x{:}t.\,\mathbf{e}]\!]^{\mathrm{F}}_{\rho} \subseteq \mathscr{P}_f(\mathscr{F} \times \mathscr{D}^{\mathrm{F}}_{\Omega})$. Therefore, $\Omega^{\rho}_{(\lambda x:t.\,\mathbf{e})\,\mathbf{v}} = \emptyset$.
    Now let $\partial \in [\![(\lambda x{:}t.\,\mathbf{e})\,\mathbf{v}]\!]^{\mathrm{F}}_{\rho}$. There exists $R \in [\![\lambda x{:}t.\,\mathbf{e}]\!]^{\mathrm{F}}_{\rho}$ and $S \subseteq [\![\mathbf{v}]\!]^{\mathrm{F}}_{\rho}$ such that $(S, \partial) \in R$. By Theorem 10.15, $S \subseteq [\![t]\!]^{\mathrm{F}}$. Thus, by Definition 10.5, it holds that $\partial \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho, x \mapsto S}$. And the result follows from Lemma 10.19.
  - Let $\partial \in [\![\mathbf{e}\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_{\rho}$. By Lemma 10.19, there exists $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{F}}_{\rho})$ such that $\partial \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho, x \mapsto S}$. By Theorem 10.15, it holds that $S \subseteq [\![t]\!]^{\mathrm{F}}$. Therefore, $\{(S, \partial)\} \in [\![\lambda x{:}t.\,\mathbf{e}]\!]^{\mathrm{F}}_{\rho}$ and the result follows.
- $[\underline{\mathrm{R}^{\mathrm{F}}_{\mathrm{proj_i}}}]$. $\pi_i\,(\mathbf{v}_1, \mathbf{v}_2) \rightsquigarrow \mathbf{v}_i$. By hypothesis, $\mathbf{e}$ is well-typed, therefore, by inversion of the typing rules, $\Gamma \vdash \mathbf{v}_i : t_i$ for every $i \in \{1, 2\}$. By Theorem 10.15, $\Omega \notin [\![\mathbf{v}_1]\!]^{\mathrm{F}}_{\rho} \cup [\![\mathbf{v}_2]\!]^{\mathrm{F}}_{\rho}$. By Definition 10.5, $[\![(\mathbf{v}_1, \mathbf{v}_2)]\!]^{\mathrm{F}}_{\rho} \subseteq \mathscr{D}^{\mathrm{F}} \times \mathscr{D}^{\mathrm{F}}$. Therefore, $\Omega^{\rho}_{\pi_i\,(\mathbf{v}_1, \mathbf{v}_2)} = \emptyset$.
  We then deduce that by Definition 10.5, $[\![(\mathbf{v}_1, \mathbf{v}_2)]\!]^{\mathrm{F}}_{\rho} = [\![\mathbf{v}_1]\!]^{\mathrm{F}}_{\rho} \times [\![\mathbf{v}_2]\!]^{\mathrm{F}}_{\rho}$, which immediately gives that $[\![\pi_i\,(\mathbf{v}_1, \mathbf{v}_2)]\!]^{\mathrm{F}}_{\rho} = [\![\mathbf{v}_i]\!]^{\mathrm{F}}_{\rho}$.
- $[\underline{\mathrm{R}^{\mathrm{F}}_{\mathrm{ctx}}}]$. $\mathscr{E}\,[\mathbf{e}] \rightsquigarrow \mathscr{E}\,[\mathbf{e}']$. Straightforward by induction and cases over $\mathscr{E}$, considering Definition 10.5.

□

10.3.3. Computational adequacy

The next property of our semantics is the property of computational adequacy, which states that if an expression diverges, then its semantics is empty. The statement of the computational adequacy only applies to well-typed, closed terms, which we call *programs*. As we will explore in Chapter 13 (Subsection 13.2.1), there are ill-typed terms in our system whose semantics is empty but that do not diverge. Although we conjecture that the adequacy still holds for open terms, proving it would greatly complicate the proofs. This is due to the fact that we prove the adequacy property by contrapositive: we show that if an element d belongs to $[\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho}$, then $\mathbf{e}$ reduces to a value $\mathbf{v}$ such that d belongs to $[\![\mathbf{v}]\!]^{\mathrm{F}}_{\rho}$. In the presence of free variables, this simply does not hold: given an environment ρ such that $\rho(x) = \{d\}$, then it is clear that $[\![x]\!]^{\mathrm{F}}_{\rho}$ contains d, but x does not reduce to a value. This problem could be solved by defining head normal forms, and proving that $\mathbf{e}$ reduces, in this case, to a term in normal form. Since the increased complexity of such a result far outweighs its benefits, we choose to limit ourselves to closed terms.

In the following, we write Prgs for the set of *programs* of our calculus, that is, the subset of Terms of closed well-typed terms:

$$\mathsf{Prgs} = \{\mathbf{e} \in \mathsf{Terms} \mid \mathbf{e} \text{ closed and } \exists t \in \mathsf{Types}.\ \vdash \mathbf{e} : t\}$$

Using this definition, the adequacy theorem is stated as follows:

Theorem 10.20 (Computational adequacy of λ_{F})**.** *For every term* $\mathbf{e} \in \mathsf{Prgs}$ *and every environment* $\rho \in \mathsf{Envs}^F$*, if* $\mathbf{e}$ *diverges then* $[\![\mathbf{e}]\!]^F_\rho = \emptyset$*.*

As we already hinted at in the previous chapter, the proof of this theorem is non trivial. To prove the computational adequacy, we draw our inspiration from the technique of *logical relations*. A way to understand this technique is that by defining a type-index family of relations, it provides a way to split a proof on distinct inductions, such as an induction on the types and an induction on the expressions. In our case, we do not need to define a type-indexed family of relations insofar as the elements of the domain $\mathscr{D}^{\mathrm{F}}$ are inductively defined. We can replace the induction on types by an induction on these elements: we just need a single relation $\mathscr{R}$ that relates every converging expression $\mathbf{e}$ with every element d in the semantics of $\mathbf{e}$.

Formally, the proof proceeds as follows. We define a relation $\mathscr{R} \subseteq \mathscr{D}^{\mathrm{F}} \times \mathsf{Prgs}$ between the semantics and the syntax of our calculus. Then, we prove for all programs $\mathbf{e}$ and elements d that if $d\mathscr{R}\mathbf{e}$, then there exists a value $\mathbf{v}$ such that $\mathbf{e} \rightsquigarrow^* \mathbf{v}$ and that $d\mathscr{R}\mathbf{v}$. Next, we prove for every program $\mathbf{e}$ that if $d \in [\![\mathbf{e}]\!]^{\mathrm{F}}_\rho$ then $d\mathscr{R}\mathbf{e}$. From both facts we deduce that if $\mathbf{e}$ diverges, then necessarily $[\![\mathbf{e}]\!]^{\mathrm{F}}_\rho = \emptyset$: if $[\![\mathbf{e}]\!]^{\mathrm{F}}_\rho$ is not empty, since $\mathbf{e}$ is well-typed, its semantics cannot contain Ω by type soundness. Thus, it must contain another element d. This implies that $d\mathscr{R}\mathbf{e}$ and thus that $\mathbf{e}$ converges to some value, contradicting the hypothesis that $\mathbf{e}$ diverges.

We proceed with the definition of the relation $\mathscr{R}$:

Definition 10.21. *We define the relation* $\mathscr{R} \subseteq \mathscr{D}^F \times \mathsf{Prgs}$*. The relation is noted infix* $d\mathscr{R}\mathbf{e}$ *and is defined by induction on the lexicographically ordered pair* $(d, \mathbf{e})$*, that is, the structure of the element* d *and of the term* $\mathbf{e}$*. This is done via the following cases:*

$$c\mathscr{R}\mathbf{e} \Leftrightarrow \mathbf{e} \rightsquigarrow^* c \tag{10.1}$$

$$(d_1, d_2)\mathscr{R}\mathbf{e} \Leftrightarrow d_1\mathscr{R}\pi_1\,\mathbf{e} \text{ and } d_2\mathscr{R}\pi_2\,\mathbf{e} \tag{10.2}$$

$$\{\}\mathscr{R}\mathbf{e} \Leftrightarrow \mathbf{e} \rightsquigarrow^* \lambda x{:}t.\,\mathbf{e}' \tag{10.3}$$

$$R \cup \{(S, \Omega)\}\mathscr{R}\mathbf{e} \Leftrightarrow R\mathscr{R}\mathbf{e} \tag{10.4}$$

$$R \cup \{(S, d)\}\mathscr{R}\mathbf{e} \Leftrightarrow R\mathscr{R}\mathbf{e} \text{ and } \nexists \mathbf{v} \in \mathsf{Values}.\ \forall d' \in S.\ d'\mathscr{R}\mathbf{v} \tag{10.5}$$

$$\{(S_i, d_i) \mid i \in I\}_{I\neq\emptyset}\mathscr{R}\mathbf{e} \Leftrightarrow \forall i{\in}I. \begin{cases} \exists \mathbf{v} \in \mathsf{Values}.\ \forall d \in S_i.\ d\mathscr{R}\mathbf{v} \\ \forall \mathbf{v} \in \mathsf{Values}.\ (\forall d \in S_i.\ d\mathscr{R}\mathbf{v}) \Rightarrow d_i\mathscr{R}\mathbf{e}\,\mathbf{v} \end{cases} \tag{10.6}$$

As we explained before, this definition can be understood by considering that the relation $\mathscr{R}$ is defined so that, for every element d, it satisfies the following property:

$$d\mathscr{R}\mathbf{e} \quad \Longrightarrow \quad \mathbf{e} \rightsquigarrow^* \mathbf{v} \text{ and } d \in [\![\mathbf{v}]\!]^{\mathrm{F}}_\rho$$

Let us explain each clause in detail. Clause (10.1) relates c with all expressions that reduce to c; clause (10.2) relates (d_1, d_2) to the expressions that reduce to a pair of values whose denotation

contains (d_1, d_2); since the empty relation is a sound approximation for all functions, then clause (10.3) relates it with every expression that reduces to a function.

Now to understand the last three clauses, it is important to understand that, since the computational adequacy only applies to terms whose semantics does not contain Ω, a relation R can carry a lot of useless information. For example, knowing that applying an expression $\mathbf{e}$ to the value 1 produces a stuck term tells us very little about the convergence of $\mathbf{e}$. This can occur when the pair $(\{1\}, \Omega)$ is included in a denotation of $\mathbf{e}$. Hence, clause (10.4) removes such useless pairs from relations. Similarly, a relation can contain inputs which are not "constructible", that is, are not denotations of any value. For example, knowing that the mapping $(\{1, 2\}, 1)$ is included in a relation denoting an expression $\mathbf{e}$ does not give us any information about $\mathbf{e}$ since it is impossible to apply it to the set of denotations $\{1, 2\}$ (no value can be denoted by both 1 and 2). Clause (10.5) removes such pairs from relations. Finally, the last clause (10.6) enters in action only when the other two previous clauses have already "cleansed" an approximation leaving only pairs with constructible well-typed inputs. It then checks that every input set is constructible and that for every value denoting this set, applying the expression to this value produces a result that correctly denotes the output.

Note that the definitions of clauses (10.4) and (10.5) are well-founded: since relations are finite, by removing pairs from a relation we either obtain an empty relation (on which we can apply (10.3)), or a relation whose pairs all satisfy the premises of clause (10.6).

We are now almost ready to state the first important property of our relation, namely that every expression related to some element converges. However, to prove this crucial property, we first need to prove that the relation is "preserved" by reduction.

Lemma 10.22. *For all* $\mathbf{e}_1, \mathbf{e}_2 \in \mathsf{Prgs}$ *and* $d \in \mathscr{D}^F$*, if* $d \mathscr{R} \mathbf{e}_2$ *and* $\mathbf{e}_1 \rightsquigarrow \mathbf{e}_2$ *then* $d \mathscr{R} \mathbf{e}_1$.

Proof. By induction on d and cases over $d \mathscr{R} \mathbf{e}_2$.

- $\underline{c \mathscr{R} \mathbf{e}_2}$. By Definition 10.21, $\mathbf{e}_2 \rightsquigarrow^* c$. Therefore, since $\mathbf{e}_1 \rightsquigarrow \mathbf{e}_2$, we have $\mathbf{e}_1 \rightsquigarrow^* c$, hence the result.
- $\underline{(d_1, d_2) \mathscr{R} \mathbf{e}_2}$. Let $i \in \{1, 2\}$. By Definition 10.21, $d_i \mathscr{R} \pi_i\, \mathbf{e}_2$. By definition of the reduction contexts, $\pi_i\, \mathbf{e}_1 \rightsquigarrow^* \pi_i\, \mathbf{e}_2$. By induction hypothesis, $d_i \mathscr{R} \pi_i\, \mathbf{e}_1$, and the result follows.
- $\underline{\{\} \mathscr{R} \mathbf{e}_2}$. By Definition 10.21, $\mathbf{e}_2 \rightsquigarrow^* \lambda x{:}t.\, \mathbf{e}$. Therefore, $\mathbf{e}_1 \rightsquigarrow^* \lambda x{:}t.\, \mathbf{e}$ and the result follows.
- $\underline{R \cup \{(S, \Omega)\} \mathscr{R} \mathbf{e}_2}$. By Definition 10.21, $R \mathscr{R} \mathbf{e}_2$. By induction hypothesis, $R \mathscr{R} \mathbf{e}_1$. Thus, $R \cup \{(S, \Omega)\} \mathscr{R} \mathbf{e}_1$.
- $\underline{R \cup \{(S, d)\} \mathscr{R} \mathbf{e}_2}$. Where $R \mathscr{R} \mathbf{e}_2$ and $\nexists \mathbf{v} \in \mathsf{Values}.\ \forall d' \in S.\ d' \mathscr{R} \mathbf{v}$. By induction hypothesis, $R \mathscr{R} \mathbf{e}_1$. Thus, by Definition 10.21, $R \cup \{(S, d)\} \mathscr{R} \mathbf{e}_1$.
- $\underline{\{(S_1, d_1), \ldots, (S_n, d_n)\} \mathscr{R} \mathbf{e}_2}$. Let $i \in \{1..n\}$. By Definition 10.21, there exists $\mathbf{v} \in \mathsf{Values}$ such that $\forall d \in S_i$, $d \mathscr{R} \mathbf{v}$, and $d_i \mathscr{R} \mathbf{e}_2\, \mathbf{v}$. By definition of the reduction contexts, $\mathbf{e}_1\, \mathbf{v} \rightsquigarrow^* \mathbf{e}_2\, \mathbf{v}$. Therefore, by induction hypothesis, $d_i \mathscr{R} \mathbf{e}_1\, \mathbf{v}$, and the result follows.

□

We can now state the first lemma needed to prove the adequacy property.

Lemma 10.23. *For all* $\mathbf{e} \in \mathsf{Prgs}$ *and* $d \in \mathscr{D}^F$*, if* $d \mathscr{R} \mathbf{e}$ *then* $\mathbf{e} \rightsquigarrow^* \mathbf{v}$ *and* $d \mathscr{R} \mathbf{v}$.

Proof. By induction on d and cases over $d \mathscr{R} \mathbf{e}$.

- $\underline{c \mathscr{R} \mathbf{e}}$. By Definition 10.21, $\mathbf{e} \rightsquigarrow^* c$, and $c \mathscr{R} c$.
- $\underline{(d_1, d_2) \mathscr{R} \mathbf{e}}$. Let $i \in \{1, 2\}$. By Definition 10.21, $d_i \mathscr{R} \pi_i \, \mathbf{e}$. By induction hypothesis, $\pi_i \, \mathbf{e} \rightsquigarrow^* \mathbf{v}_i$ and $d_i \mathscr{R} \mathbf{v}_i$. By inversion of the reduction rules, $\mathbf{e} \rightsquigarrow^* (\mathbf{v}_1, \mathbf{v}_2)$. And by Definition 10.21, $(d_1, d_2) \mathscr{R} (\mathbf{v}_1, \mathbf{v}_2)$.
- $\underline{\{\} \mathscr{R} \mathbf{e}}$. By Definition 10.21, $\mathbf{e} \rightsquigarrow^* \lambda x{:}t. \, \mathbf{e}'$, and $\{\} \mathscr{R} \lambda x{:}t. \, \mathbf{e}'$.
- $\underline{R \cup \{(S, \Omega)\} \mathscr{R} \mathbf{e}}$. By Definition 10.21, $R \mathscr{R} \mathbf{e}$. By induction hypothesis, $\mathbf{e} \rightsquigarrow^* \mathbf{v}$ such that $R \mathscr{R} \mathbf{v}$. The result follows from Definition 10.21.
- $\underline{R \cup \{(S, d)\} \mathscr{R} \mathbf{e}}$. Where $R \mathscr{R} \mathbf{e}$ and $\nexists \mathbf{v} \in \mathsf{Values}. \, \forall d' \in S. \, d' \mathscr{R} \mathbf{v}$. By induction hypothesis, $\mathbf{e} \rightsquigarrow^* \mathbf{v}'$ such that $R \mathscr{R} \mathbf{v}'$. The result follows from Definition 10.21.
- $\underline{\{(S_1, d_1), \ldots, (S_n, d_n)\} \mathscr{R} \mathbf{e}}$. Let $i \in \{1..n\}$. By Definition 10.21, there exists $\mathbf{v}_i \in \mathsf{Values}$ such that $\forall d \in S_i, \, d \mathscr{R} \mathbf{v}_i$. Moreover, $\mathbf{v}_i$ satisfies $d_i \mathscr{R} \mathbf{e} \, \mathbf{v}_i$.
  By induction hypothesis, there exists $\mathbf{v}_i' \in \mathsf{Values}$ such that $\mathbf{e} \, \mathbf{v}_i \rightsquigarrow^* \mathbf{v}_i'$ and $d_i \mathscr{R} \mathbf{v}_i'$. By inversion of the reduction rules, this entails $\mathbf{e} \rightsquigarrow^* \lambda x{:}t. \, \mathbf{e}'$ where $(\lambda x{:}t. \, \mathbf{e}') \, \mathbf{v}_i \rightsquigarrow^* \mathbf{v}_i'$. By Lemma 10.22, this ensures that $d_i \mathscr{R} (\lambda x{:}t. \, \mathbf{e}') \, \mathbf{v}_i$.
  Since the reduction of $\mathbf{e}$ is independent of i and the choice of $\mathbf{v}_i$, this result holds for every i and every value $\mathbf{v}_i$ such that $\forall d \in S_i, \; d \mathscr{R} \mathbf{v}_i$. Hence, by Definition 10.21, this yields $\{(S_1, d_1), \ldots, (S_n, d_n)\} \mathscr{R} \lambda x{:}t. \, \mathbf{e}'$.

□

The second lemma, which states that if $d \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho}$ then $d \mathscr{R} \mathbf{e}$, cannot be proven immediately. It needs to be strengthened as some form of substitution lemma. The idea behind this substitution lemma is to state that if d is an element of the semantics of a possibly open term $\mathbf{e}$ in an environment where variables have been bound to the semantics of some values, then d is in relation with $\mathbf{e}$ where the variables have been substituted by their corresponding values.

Lemma 10.24 (Adequacy substitution lemma)**.** *For every* $\mathbf{e} \in \mathsf{Terms}$*, for every* $\rho \in \mathsf{Envs}^F$*,* $x_1 \ldots x_n \in \mathsf{Vars}$*,* $\mathbf{v}_1 \ldots \mathbf{v}_n \in \mathsf{Values}$*, and* $S_1 \ldots S_n \subseteq \mathscr{D}^F$*, if the following conditions hold:*

1. $\mathsf{vars}(\mathbf{e}) \subseteq \{x_1, \ldots, x_n\}$
2. $\forall i \in \{1..n\}. \forall d_i \in S_i. \, d_i \mathscr{R} \mathbf{v}_i$

then for all $d \in [\![\mathbf{e}]\!]^F_{\rho, x_1 \mapsto S_1 \ldots x_n \mapsto S_n}$*, we have* $d \mathscr{R} \mathbf{e}[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n]$.

Proof. By induction on $\mathbf{e} \in \mathsf{Terms}$. For the sake of concision, let us write $\rho_n = \rho, x_1 \mapsto S_1 \ldots x_n \mapsto S_n$, and $\mathbf{e}_n = \mathbf{e}[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n]$.

- $\underline{\mathbf{e} = c}$. We have $[\![c]\!]^{\mathrm{F}}_{\rho_n} = \{c\}$ and $\mathbf{e}_n = c$. By Definition 10.21, $c \mathscr{R} c$, hence the result.
- $\underline{\mathbf{e} = x}$. According to the first condition, necessarily $x = x_i$ for some $i \in \{1..n\}$. We then have $[\![x_i]\!]^{\mathrm{F}}_{\rho_n} = S_i$ and $\mathbf{e}_n = \mathbf{v}_i$. By hypothesis, $\forall d \in S_i, \, d \mathscr{R} \mathbf{v}_i$, hence the result.
- $\underline{\mathbf{e} = \lambda x{:}t. \, \mathbf{e}'}$. Let $R \in [\![\lambda x{:}t. \, \mathbf{e}']\!]^{\mathrm{F}}_{\rho_n}$. We distinguish four cases.

1. $R = \{\}$. By Definition 10.21, we immediately have $\{\}\mathscr{R}\mathbf{e}_n$ since $\mathbf{e}_n$ is a λ-abstraction.
2. $R = R' \cup \{(S, \partial)\}$. By Definition 10.5, $R' \in [\![\lambda x{:}t.\, \mathbf{e}']\!]^{\mathrm{F}}_{\rho_n}$. Therefore, by induction hypothesis, $R'\mathscr{R}\mathbf{e}_n$, and the result follows from Definition 10.21.
3. $R = R' \cup \{(S, d)\}$ where $\nexists \mathbf{v} \in \mathsf{Values}$ such that $\forall d' \in S.\ d'\mathscr{R}\mathbf{v}$. By Definition 10.5, $R' \in [\![\lambda x{:}t.\, \mathbf{e}']\!]^{\mathrm{F}}_{\rho_n}$. Therefore, by induction hypothesis, $R'\mathscr{R}\mathbf{e}_n$, and the result follows from Definition 10.21.
4. $R = \{(S_1, d_1), \ldots, (S_n, d_n)\}_{(n\geq 1)}$ where $\forall i \in \{1..n\}$, $\exists \mathbf{v} \in \mathsf{Values}$ such that $\forall d \in S_i.\ d\mathscr{R}\mathbf{v}$.
   Let $i \in \{1..n\}$, and a value $\mathbf{v} \in \mathsf{Values}$ such that $\forall d \in S_i.\ d\mathscr{R}\mathbf{v}$. We need to prove that $d_i\mathscr{R}\mathbf{e}_n\,\mathbf{v}$. By Definition 10.5, we have $S_i \subseteq [\![t]\!]^{\mathrm{F}}$ and $d_i \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho_n, x\mapsto S_i}$. By $[\mathrm{R}^{\mathrm{F}}_{\mathrm{app}}]$, we deduce that $\mathbf{e}_n\,\mathbf{v} \rightsquigarrow \mathbf{e}'[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n][\mathbf{v}/x]$. Since values are closed terms, we have that $x \sharp \mathbf{v}_i$ for every $i \in \{1..n\}$, which ensures that $[\![\mathbf{e}'[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n][\mathbf{v}/x]]\!]^{\mathrm{F}} = [\![\mathbf{e}'[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n, \mathbf{v}/x]]\!]^{\mathrm{F}}$. By induction hypothesis on $\mathbf{e}'$ generalized on ρ, we obtain that $d_i\mathscr{R}\mathbf{e}'[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n, \mathbf{v}/x]$. Since $\mathbf{e}_n\,\mathbf{v} \rightsquigarrow^* \mathbf{e}'[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n, \mathbf{v}/x]$, we deduce from Lemma 10.22 that $d_i\mathscr{R}\mathbf{e}_n\,\mathbf{v}$, hence the result.

- $\underline{\mathbf{e}_1\,\mathbf{e}_2}$. Let $d \in [\![\mathbf{e}_1\,\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho_n}$. By Definition 10.5, there exists $R \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho_n}$ and $S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho_n}$ such that $(S, d) \in R$. By induction hypothesis, we obtain that $R\mathscr{R}\mathbf{e}_1[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n]$ and $\forall d \in S.\ d\mathscr{R}\mathbf{e}_2[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n]$.
  Now, by Lemma 10.23, there exists $\mathbf{v}'_1 \in \mathsf{Values}$ such that $\mathbf{e}_1[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n] \rightsquigarrow^* \mathbf{v}'_1$ and $R\mathscr{R}\mathbf{v}'_1$. Similarly for $\mathbf{e}_2$, there exists $\mathbf{v}'_2$ such that $\mathbf{e}_2[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n] \rightsquigarrow^* \mathbf{v}'_2$ and $\forall d \in S.\ d\mathscr{R}\mathbf{v}'_2$.
  By Definition 10.21, this entails $d\mathscr{R}\mathbf{v}'_1\,\mathbf{v}'_2$. Since $\mathbf{e}_n \rightsquigarrow \mathbf{v}'_1\,\mathbf{v}'_2$, applying Lemma 10.22 ensures that $d\mathscr{R}\mathbf{e}_n$, hence the result.

- $\underline{(\mathbf{e}_1, \mathbf{e}_2)}$. Let $d \in [\![(\mathbf{e}_1, \mathbf{e}_2)]\!]^{\mathrm{F}}_{\rho_n}$. By Definition 10.5, we have $d = (d_1, d_2)$ where for $i \in \{1, 2\}$, $d_i \in [\![\mathbf{e}_i]\!]^{\mathrm{F}}_{\rho_n}$. By IH, we have $d_i\mathscr{R}\mathbf{e}_i$. Additionally, for every $i \in \{1, 2\}$, we have $\pi_i\,\mathbf{e}_n \rightsquigarrow \mathbf{e}_i[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n]$. Hence, by Lemma 10.22, we deduce $d_i\mathscr{R}\pi_i\,\mathbf{e}_n$. By Definition 10.5, this proves $(d_1, d_2)\mathscr{R}(\mathbf{e}_1, \mathbf{e}_2)$, which is the result.

- $\underline{\pi_i\,\mathbf{e}'}$. Consider w.l.o.g. that $i = 1$. Let $d_1 \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho_n}$. By Definition 10.5, there exists $d_2 \in \mathscr{D}^{\mathrm{F}}$ such that $(d_1, d_2) \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho_n}$. By IH, we deduce that $(d_1, d_2)\mathscr{R}\mathbf{e}'[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n]$. Therefore, by Definition 10.21, we deduce that $d_1\mathscr{R}\pi_1\,\mathbf{e}'[\mathbf{v}_1/x_1 \ldots \mathbf{v}_n/x_n]$, hence the result.

□

Equipped with the previous lemma, we can now prove the second part needed to prove the adequacy of our semantics, namely that an expression with non-empty semantics is related to the elements in its semantics:

Corollary 10.25. *For all* $\mathbf{e} \in \mathsf{Prgs}$, $\rho \in \mathsf{Envs}^F$, *and* $d \in \mathscr{D}^F$, *if* $d \in [\![\mathbf{e}]\!]^F_\rho$ *then* $d\mathscr{R}\mathbf{e}$.

Proof. Immediate corollary of Lemma 10.24 taking $n = 0$. □

The adequacy theorem is then a simple corollary of Lemma 10.23 and Corollary 10.25.

Proof of Theorem 10.20. By contrapositive. Let $\mathbf{e} \in \mathsf{Prgs}$ and $\rho \in \mathsf{Envs}^{\mathrm{F}}$ such that $[\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho} \neq \emptyset$. Since $\mathbf{e}$ is well-typed, by Theorem 10.15, we have $\Omega \notin [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho}$. Therefore, there must exist $d \in \mathscr{D}^{\mathrm{F}}$ such that $d \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho}$. By Corollary 10.25, $d \mathscr{R} \mathbf{e}$. By Lemma 10.23, there exists $\mathbf{v} \in \mathsf{Values}$ such that $\mathbf{e} \rightsquigarrow^{*} \mathbf{v}$, which proves that $\mathbf{e}$ does not diverge. □

Chapter 11.

A denotational semantics for CDuce

"Ideals are dangerous things. Realities are better. They wound, but they're better."

Oscar Wilde, *Lady Windermere's Fan*

As we discussed in Chapter 8, if we want the subtyping relation defined by the interpretation of Definition 2.6 to coincide with the model of the values of the language ([27, Theorem 5.2]), then the language must provide enough values to separate every pair of distinct types. In other terms, whenever two types do not have the same interpretation in the denotational model, then there must exist a value in the language that is in one type but not in the other one.

The language defined in the previous chapter does not have enough values to separate every pair of types that have distinct interpretations according to Definition 2.5. In order to provide enough values to distinguish semantically different types we must enrich the system of the previous chapter in three ways: (*i*) we must improve the type system to infer (non-trivial) intersection types for functions, (*ii*) we must enrich the terms with a typecase expression, and (*iii*) we must add to the language a non-deterministic choice operator. The resulting calculus is the functional core of the language ℂDuce [8].

The first point is necessary to ensure that non-trivial intersections of arrow types can be introduced by the type system, and that they can thus be inhabited by some values of the language. The second point ensures that some distinct intersections can be separated from each other. Consider for example the type $(\mathsf{Int} \to \mathsf{Bool}) \wedge (\mathsf{Bool} \to \mathsf{Int})$. Without a form of type introspection, the only function that inhabits this type is the function that always diverge on both integers and booleans, thus making it indistinguishable from the type $(\mathsf{Bool} \vee \mathsf{Int}) \to \mathbb{0}$. On the other hand, the third point ensures that some distinct union types can be separated. This is the case for example with the types $\mathtt{true} \to \mathsf{Bool}$ and $(\mathtt{true} \to \mathtt{true}) \vee (\mathtt{true} \to \mathtt{false})$. Without a means of non-deterministically returning two different results, any function applied to `true` and returning a boolean would either always return `true` (thus having type $\mathtt{true} \to \mathtt{true}$) or always return `false` (thus having type $\mathtt{true} \to \mathtt{false}$).

In this chapter, we extend the denotational semantics presented in the previous chapter, following these three directions. We also highlight several problems related to the interpretation of negations types: denotational semantics often rely on the fact that types are interpreted as ideals, yet, as we anticipated in Chapter 2, negation types are not interpreted as ideals in semantic subtyping (the complement of an ideal *is not* an ideal). However, by forcing negation types to be introduced explicitly via type annotations, we deduce a sound denotational semantics for the functional core of ℂDuce .

Chapter Outline

Section 11.1 In this section, we add support for the inference of intersection types for

functions. We change the syntax of λ-abstractions to add *interfaces*, and adapt the type system in consequence. We show how to deduce negation types for functions while keeping the soundness of the semantics, thus reconciling the interpretation of negation types with the semantics.

Section 11.2 We extend our system to support typecases, which are a form of conditional branching allowing for type introspection, a necessary feature to ensure that some distinct intersection types can be separated from each other.

Section 11.3 We next extend our system to support a form of non-determinism. We show how we can extend the interpretation domain with *marks*, whose goal is to interpret non-deterministic computations.

Section 11.4 We summarize the previous sections by providing the formal syntax, type system, operational semantics, and denotational semantics of the language. We then prove the soundness of our semantics.

Section 11.5 We discuss the introduction of negation types in the interfaces of λ-abstractions. While our language and type system do not exactly correspond to the system of Frisch et al. [27], we show that we can still reconcile the two by inferring negation types based on the static type information present in a program.

Section 11.6 Finally, we conclude by briefly summarizing the main contributions of this chapter.

11.1. Inferring intersection types for functions

As anticipated the only way to deduce an intersection type for a λ-abstraction in the system of Chapter 10 is to use subsumption. So for instance, for $\lambda x{:}\mathsf{Int}\vee\mathsf{Bool}.\, x$, the system of Chapter 10 can deduce the type $\mathsf{Int}\vee\mathsf{Bool}\to\mathsf{Int}\vee\mathsf{Bool}$ but not the type $(\mathsf{Int}\to\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool})$ which is more precise (it is a subtype of the former, and, thus, subsumption cannot be used to infer it), while being a sound abstraction of the function above (since the identity function maps integers to integers and Booleans to Booleans).

As a matter of fact, the only λ-abstractions in the previous system for which we can deduce the type $(\mathsf{Int}\to\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool})$ are all equivalent to $\lambda x{:}t.\,\bot$, that is, they are the functions that diverge on all arguments of type t, for any t supertype of $\mathsf{Int}\vee\mathsf{Bool}$ (equivalently, they are the functions in $\mathsf{Int}\vee\mathsf{Bool}\to\mathbb{0}$). Therefore, if intersection types for λ-abstractions can be deduced only by subsumption, then there are not enough values to distinguish, say, $(\mathsf{Int}\to\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool})$ from $\mathsf{Int}\vee\mathsf{Bool}\to\mathbb{0}$, even though they are two types that have different interpretations according to Definition 2.5.

A first solution could be to modify the typing rule for abstractions so that for a function such as $\lambda x{:}\mathsf{Int}\vee\mathsf{Bool}.\, x$ the rule checks that it has both type $\mathsf{Int}\to\mathsf{Int}$ and type $\mathsf{Bool}\to\mathsf{Bool}$ and deduces their intersection for it. This solution, however, presents two major problems. First, we lose principal types (e.g., $\lambda x{:}\mathbb{1}.\, x$ has type $t\to t$ for every type t and thus a principal type would require infinite intersections). Second, and more importantly, in our case it would yield either an unsound denotational semantics or an undecidable type system: while it is easy to check that, say, $\lambda x{:}\mathsf{Int}.\,42$ has type $\mathsf{Int}\to 42$, checking this type for a (well-typed) function $\lambda x{:}\mathsf{Int}.\, e$ would correspond to check that $e\,[n/x]$ yields 42 for all integers n. If this were not decidable for that particular e, then the only solution for having a decidable type system would be to deduce

that $\lambda x{:}\mathsf{Int}.\, e$ does *not* have type $\mathsf{Int} \to 42$ and, thus, that it has type $\neg(\mathsf{Int} \to 42)$. But if $e\,[n/x]$ actually yields 42 for all integers n, then its denotation would be in the interpretation of $\mathsf{Int} \to 42$ and not in the one of $\neg(\mathsf{Int} \to 42)$.

To solve these problems, Frisch et al. [27] annotate λ-abstractions with their intersection types, thus providing also their return type(s). So the identity function for integer and Booleans can be written in system of Frisch et al. [27] as $\lambda^{(\mathsf{Int}\to\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool})}x.\, x$ and the system deduces for it the type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$, and by subsumption the type $\mathsf{Int} \vee \mathsf{Bool} \to \mathsf{Int} \vee \mathsf{Bool}$, but clearly not the type $\mathsf{Int} \vee \mathsf{Bool}\to\mathbb{0}$, thus distinguishing them.

11.1.1. Syntax and type system

The first modification to the system of Chapter 10 is then to adopt syntax and typing rules adapted from Frisch et al. [27] for λ-abstractions. As anticipated, Frisch et al. [27] introduce the following production for λ-abstractions:

$$\mathbf{e} ::= \lambda^{\bigwedge_{i\in I}(s_i\to t_i)}x.\, \mathbf{e} \qquad\qquad I \text{ finite}$$

as well as the following typing rule:

$$[\mathrm{T}_{\mathsf{Abs}}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash \mathbf{e} : t_i}{\Gamma \vdash \lambda^{\bigwedge_{i\in I}(s_i\to t_i)}x.\, \mathbf{e} : t \wedge t'} \quad \begin{array}{l} t = \bigwedge_{i\in I}(s_i \to t_i) \\ t' = \bigwedge_{n\in N} \neg(s_n \to t_n) \\ t \wedge t' \not\simeq \mathbb{0} \end{array}$$

This rule (taken verbatim from [27]) checks whether a λ-abstraction has all the arrow types listed in its annotation t and deduces for the term this type t intersected with an arbitrary finite number of negated arrow types. These negated arrow types can be chosen freely provided that the type $t \wedge t'$ remains non-empty. The purpose of the rule is to ensure that, given any function and any type t, either the function has type t or it has type $\neg t$. This property not only matches the view of types as sets of values that underpins semantic subtyping, but also it is necessary to ensure subject reduction in the presence of typecases (see [27] for details).[1]

The addition of interfaces and rule $[\mathrm{T}_{\mathsf{Abs}}]$ may look surprising. For example, it allows the system to type $\lambda^{\mathsf{Int}\to\mathsf{Int}}x.\, x$ as $(\mathsf{Int} \to \mathsf{Int}) \wedge \neg(\mathsf{Bool} \to \mathsf{Bool})$ (notice the negation) even though, disregarding the annotation, the function *does* map Booleans to Booleans. But the language is intrisically typed, and thus we cannot ignore the annotations: indeed, the function does not have type $\mathsf{Bool} \to \mathsf{Bool}$ insofar as its application to a Boolean does not return another Boolean but an error Ω. The point is that the theory of semantic subtyping gives expressions an *intrinsic semantics* (in the sense of Reynolds [61]) since the semantics of λ-abstractions depends on their explicit type annotations, and this must be reflected in the denotational semantics of the expressions.

In particular, notice that according to rule $[\mathrm{T}_{\mathsf{Abs}}]$ we have $\lambda^{\mathsf{Int}\to 42}x.\, 42 : \mathsf{Int} \to 42$ while $\lambda^{\mathsf{Int}\to\mathsf{Int}}x.\, 42 : \neg(\mathsf{Int} \to 42)$ (notice the difference in the annotations), even though both functions can be applied to integers, and both return 42 on all integers. Therefore, $\lambda^{\mathsf{Int}\to 42}x.\, 42$ and $\lambda^{\mathsf{Int}\to\mathsf{Int}}x.\, 42$ must have different denotations, which means that the denotational semantics of a λ-abstraction must take into account both the codomain of its annotation and the possible negation types that can be inferred, which the interpretation of the previous chapter does not do.

This, however, poses a major problem for our denotational semantics: by allowing the type of a λ-abstraction to be arbitrarily refined, it breaks the type soundness property as stated in

[1] Although by this rule it is possible to deduce infinite many distinct types for the same expression, the system still has a notion of principality, obtained by the introduction of *type schemes*: see Frisch et al. [27, Section 6.12].

Theorem 10.15. As an example, consider the identity function $f = \lambda^{\mathsf{Int}\to\mathsf{Int}} x.\, x$. Intuitively, it is clear that the empty relation $R = \{\}$ belongs to its denotational semantics, since it is a valid approximation of its behaviour. If we deduce for f the type $\mathsf{Int} \to \mathsf{Int}$ by $[\mathrm{T}_{\mathrm{Abs}}]$, then this is not a problem, as the empty relation effectively belongs to the interpretation of $\mathsf{Int} \to \mathsf{Int}$. However, $[\mathrm{T}_{\mathrm{Abs}}]$ also allows us to deduce the type $(\mathsf{Int} \to \mathsf{Int}) \wedge \neg(\mathbb{1} \to \mathbb{0})$ for f, since this intersection is non-empty. And since $R \in [\![\mathbb{1} \to \mathbb{0}]\!]$, it is clear that $R \notin [\![(\mathsf{Int} \to \mathsf{Int}) \wedge \neg(\mathbb{1} \to \mathbb{0})]\!]$, and thus the denotational semantics of f is not contained in the interpretation of one of its type.

One could argue that it was wrong to assume that the empty relation $\{\}$ belonged to the semantics of f. However, this problem does not only occur with the empty relation. In fact, given any approximation R of a λ-abstraction f, as long as f admits an infinite number of approximations (which happens as long as its codomain is non-empty), one can find a type $s \to t$ such that f has type $\neg(s \to t)$ and $R \notin [\![\neg(s \to t)]\!]$.

To solve this problem, we force λ-abstractions to be annotated in advance with their negation types, and we do not allow the rule $[\mathrm{T}_{\mathrm{Abs}}]$ to further refine the type of a function. We will then be able to use this information to remove unwanted elements from the semantics of an abstraction. While this may seem constraining at first, we will discuss in Section 11.5 how we can automatically deduce negative annotations for λ-abstractions, thus reconciling our semantics with the semantics of Frisch et al. [27].

To summarize, we replace the production rule for λ-abstractions from the previous chapters (page 156) by the following rule:

$$\mathbf{e} ::= \lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)} x.\, \mathbf{e} \qquad\qquad I, N \text{ finite}$$

As anticipated, λ-abstractions are annotated with an intersection of arrow types and negation of arrow types. We will refer to this annotation as its *interface*.

We also replace the typing rule of Figure 9.1 for abstractions $[\mathrm{T}^{\mathrm{F}}_{\mathrm{Abs}}]$ by the following rule:

$$[\mathrm{T}^{\mathrm{C}}_{\mathrm{Abs}}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash \mathbf{e} : t_i}{\Gamma \vdash \lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)} x.\, \mathbf{e} : t \wedge t'} \quad \begin{array}{l} t = \bigwedge_{i\in I}(s_i \to t_i) \\ t' = \bigwedge_{n\in N}\neg(s_n \to t_n) \\ t \wedge t' \not\simeq \mathbb{0} \end{array}$$

This rule is almost exactly the rule of Frisch et al. [27], except the negative part of the type is not inferred but taken from the interface of the function.

11.1.2. Relating interfaces and denotations

We now know in advance which negation types will be given to a function, but we still need to ensure that the denotational semantics of abstractions take their interface into account.

To that end, we modify the interpretation of a λ-abstraction by adding a form of "dummy" inputs, which serve to range over all the results that belong to the codomain of the annotation of the λ-abstraction, but cannot denote any expression. We associate such an input $\mho_d$ (read: "agemo") to every element d of the interpretation domain.

In practice, if R is a relation in the denotation of $\lambda^{s\to t} x.\, \mathbf{e}$ and $(\mho_d, d') \in R$, if $d \in [\![s]\!]^{\mathrm{C}}$ then $d' \in [\![t]\!]^{\mathrm{C}}$. This follows the same principle as the interpretation of types given in the previous chapters: a relation R is in the interpretation of $s \to t$ if for every input that belongs to the interpretation of s, the corresponding output belongs to the interpretation of t. The symbol $\mho$ just serves to ensure the input cannot denote any expression.

Thanks to this marker $\lambda^{(\mathsf{Int}\to\mathsf{Int})\wedge\neg(\mathsf{Int}\to 42)} x.\, 42$ and $\lambda^{\mathsf{Int}\to 42} x.\, 42$ now have different interpretations since any relation in the denotation of the former can contain the pairs $(\mho_n, n')$ for any

integers n and n', while the relations in the denotation of the latter can contain such pairs only for $n' = 42$. A similar modification has to be done to obtain the new interpretation of types $[\![.]\!]^{\mathrm{C}}$, in order to ensure that if $(\mho_d, d') \in R$ for $R \in [\![s \to t]\!]^{\mathrm{C}}$ and $d \in [\![s]\!]^{\mathrm{C}}$, then $d' \in [\![t]\!]^{\mathrm{C}}$.

11.1.3. Denotational semantics

In light of the previous explanation, the new semantics of λ-abstractions can be obtained by extending the semantics presented in the previous chapter to support multiple input types and the new inputs $\mho_d$.

In this new setting functions still denote sets of finite sets of the form $\{(\iota, \partial), \ldots, (\iota, \partial)\}$, but now inputs ι range over $\mathscr{I}^{\mathrm{C}} = \mathscr{P}_f(\mathscr{D}^{\mathrm{C}}) \cup \{\mho_d \mid d \in \mathscr{D}^{\mathrm{C}}\}$ (we will give the complete formal definition of the interpretation domain $\mathscr{D}^{\mathrm{C}}$ and the interpretation of terms in Section 11.4). The new form of λ-abstractions we defined in this section are then interpreted as follows

$$
\begin{aligned}
[\![\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)} x.\,\mathbf{e}]\!]^{\mathrm{C}}_{\rho} = \{R \in \mathscr{P}_f(\mathscr{I}^{\mathrm{C}}\times\mathscr{D}^{\mathrm{C}}_{\Omega}) \mid \forall(\iota,\partial)\in R.\ & \\
\exists i\in I.\iota \subseteq [\![s_i]\!]^{\mathrm{C}} \wedge \partial \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto\iota} \ \textit{or}\ & \\
\forall i\in I.\iota \subseteq [\![\neg s_i]\!]^{\mathrm{C}} \wedge \partial = \Omega \ \textit{or}\ & \\
\iota = \mho_d \text{ where } \forall i\in I.\, d \in [\![s_i]\!]^{\mathrm{C}} \implies \partial \in [\![t_i]\!]^{\mathrm{C}} & \\
\} \cap [\![\bigwedge_{n\in N}\neg(s_n\to t_n)]\!]^{\mathrm{C}} &
\end{aligned}
$$

As anticipated, compared to the semantics presented in Chapter 10, there are three major changes. First, we now force the input ι to either belong to *at least one* input type s_i, or to none of them. Then, in the first case, we compute the output as usual by binding x to ι, while in the second case we simply map the input to Ω. Second, we add $\mho_{(.)}$ inputs following the same principle: for every input of the form $\mho_d$, and for every input type s_i containing d, we ensure that the output belongs to the corresponding output type t_i.

Finally, we simply "filter" the denotation of the abstraction to keep only the elements that belong to the negative part of its interface. While this may seem surprising, this does not cause any loss of information: given a relation R that approximates the λ-abstraction $\lambda^{\bigwedge_{i\in I}(s_i\to t_i)} x.\,\mathbf{e}$, as long as $\bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{n\in N}\neg(s_n \to t_n)$ is non-empty, it is possible to add a finite number of pairs of the form $(\mho_d, d')$ to R and obtain a relation that belongs to $[\![\bigwedge_{n\in N}\neg(s_n \to t_n)]\!]^{\mathrm{C}}$. The resulting relation behaves exactly as R, in the sense that it maps the same denotable inputs to the same outputs. For example, given the relation $\{(\{3\}, 42)\}$ in the denotation of $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.\,42$, it is possible to complete it into the relation $\{(\{3\}, 42), (\mho_2, 43)\}$ which belongs to the denotation of $\lambda^{(\mathsf{Int}\to\mathsf{Int})\wedge\neg(\mathsf{Int}\to 42)} x.\,42$ and still maps 3 to 42. Thus, taking the intersection with $[\![\bigwedge_{n\in N}\neg(s_n \to t_n)]\!]^{\mathrm{C}}$ actually removes the elements that do not contain enough information. We will formalize this result later as Lemma 11.11.

Naturally, we also have to modify the interpretation of function spaces to account for the presence of the new $\mho_{(.)}$ elements. This modification is straightforward and follows the same idea as the semantics of λ-abstractions. The new interpretation must satisfy the following equation

$$
\begin{aligned}
[\![t_1 \to t_2]\!]^{\mathrm{C}} = \{R \in \mathscr{P}_f(\mathscr{I}^{\mathrm{C}}\times\mathscr{D}^{\mathrm{C}}_{\Omega}) \mid & \forall(\iota,\partial)\in R. \\
& ((\iota = \mho_d \wedge d \in [\![t_1]\!]^{\mathrm{C}}) \text{ or } \iota \cap [\![t_1]\!]^{\mathrm{C}} = \emptyset) \implies \partial \in [\![t_2]\!]^{\mathrm{C}}\}
\end{aligned}
$$

which can be obtained by the inductive definition of a predicate $(. : .)^{\mathrm{C}}$ by following Defini-

tion 10.2 and modifying the case for $(\{(S_1, \partial_1), ..., (S_n, \partial_n)\} : t_1 \to t_2)$ as follows:

$$(\{(\iota_i, \partial_i) \mid i \in I\} : t_1 \to t_2)^{\mathrm{C}} = \forall i \in I. \begin{cases} (\iota_i = \mho_d \wedge (d : t_1)^{\mathrm{C}}) \implies (\partial_i : t_2)^{\mathrm{C}} \\ (\exists d \in \iota_i.\ (d : t_1)^{\mathrm{C}}) \implies (\partial_i : t_2)^{\mathrm{C}} \end{cases}$$

(we simply split the definition in two for clarity). It is straightforward to prove that this definition induces the same subtyping relation on types as the previous one (see Proposition 11.4).

11.2. Adding typecases

11.2.1. Syntax and operational semantics

The second ingredient to obtain the system of Frisch et al. [27] is the addition of a typecase. Formally we add to the previous productions the following one:

$$\mathbf{e} ::= (x = \mathbf{e} \in t)?\ \mathbf{e} : \mathbf{e}$$

The expression $(x = \mathbf{e} \in t)?\ \mathbf{e}_1 : \mathbf{e}_2$ binds the value produced by $\mathbf{e}$ to the variable x, checks whether this value is of type t, if so it reduces to $\mathbf{e}_1$, otherwise it reduces to $\mathbf{e}_2$. Therefore to define the reduction semantics we have to determine when a value $\mathbf{v}$ has a given type t, written $\mathbf{v} \in t$. For that, Frisch et al. [27] show that it is not necessary to resort to the type-deduction system: they define a predicate $\mathbf{v} \in t$ that simply inspects the value $\mathbf{v}$ to deduce whether it can be given type t, and show the following two properties, which are necessary to ensure the soundness of their type system:

$$\mathbf{v} \in t \iff \vdash \mathbf{v} : t$$
$$\mathbf{v} \notin t \iff \mathbf{v} \in \neg t$$

Since their system can infer arbitrary negation types for functions, the definition of this predicate is quite involved, as, for example, it needs to ensure that $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.\, x \in \neg(\mathsf{Bool} \to \mathsf{Bool})$ since $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.\, x$ can be given type $\neg(\mathsf{Bool} \to \mathsf{Bool})$ according to their typing rule for abstractions.

However, since we removed the possibility of inferring negation types, we cannot reuse the predicate of Frisch et al. [27]. Even more problematic is the fact that the two properties above cannot hold at the same time in our system without additional hypotheses. Consider, for example, the abstraction $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.\, x$. It is clear that it cannot be given type $\mathsf{Bool} \to \mathsf{Bool}$ in our system. Thus, according to the first property above, we should have $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.\, x \notin \mathsf{Bool} \to \mathsf{Bool}$, but then the second property states that we must have $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.\, x \in \neg(\mathsf{Bool} \to \mathsf{Bool})$, which in turn implies that our type system should be able to type $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.\, x$ with type $\neg(\mathsf{Bool} \to \mathsf{Bool})$, which is not possible.

To solve this, we introduce some restrictions on the terms we consider. The idea is to ensure that every λ-abstraction of a program contains enough negative annotations to ensure that, for every type t appearing in a typecase in the program, the abstraction can either be given type t or type $\neg t$. We will define these restrictions formally in Section 11.4, and show in Section 11.5 how every term of the system of Frisch et al. [27] can be compiled to an operationally equivalent term satisfying these restrictions by adding a finite number of annotations.

With this in mind, we can define our version of the predicate $\in$, by stating that $\mathbf{v} \in t \overset{\text{def}}{\iff} \mathrm{type}(\mathbf{v}) \leq t$ where $\mathrm{type}(\mathbf{v})$ is inductively defined as:

$$\begin{aligned}
\mathsf{type}(c) &\stackrel{\text{def}}{=} b_c \\
\mathsf{type}(\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)}x.\,\mathbf{e}) &\stackrel{\text{def}}{=} \bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n) \\
\mathsf{type}((\mathbf{v}_1,\mathbf{v}_2)) &\stackrel{\text{def}}{=} \mathsf{type}(\mathbf{v}_1)\times\mathsf{type}(\mathbf{v}_2)
\end{aligned}$$

The definition of the type(.) operator is fairly straightforward insofar as it simply gathers the explicit type information present in a program. As anticipated, we will show that, under certain hypotheses, the two aforementioned properties hold for our operator $\in$ (see Propositions 11.8 and 11.13).

The behavior of typecases is then formalized by the following reduction rules, which are taken verbatim from Frisch et al. [27]:

$$\begin{aligned}
(x=\mathbf{v}\in t)?\,\mathbf{e}_1:\mathbf{e}_2 &\quad\rightsquigarrow\quad \mathbf{e}_1\,[\mathbf{v}/x] \quad \text{if } \mathbf{v}\in t \\
(x=\mathbf{v}\in t)?\,\mathbf{e}_1:\mathbf{e}_2 &\quad\rightsquigarrow\quad \mathbf{e}_2\,[\mathbf{v}/x] \quad \text{if } \mathbf{v}\notin t
\end{aligned}$$

Note that, according to these definitions, the test $(x=\mathbf{v}\in \mathsf{Int}\to 42)?\,\mathbf{e}_1:\mathbf{e}_2$ will reduce to $\mathbf{e}_1$ for $\mathbf{v}=\lambda^{\mathsf{Int}\to 42}x.\,42$ but to $\mathbf{e}_2$ for $\mathbf{v}=\lambda^{\mathsf{Int}\to\mathsf{Int}}x.\,42$, further emphasizing the importance for these two functions to have different semantics.

Typecase expressions are needed to define full-fledged overloaded functions as opposed to having just "coherent overloading" as found in Forsythe [60]. Indeed, the rule $[\mathrm{T}^{\mathrm{C}}_{\mathrm{Abs}}]$ we added in the previous section, if taken alone, allows the system to type a limited form of *ad hoc* polymorphism known as coherent overloading. In languages with coherent overloading such as Forsythe or the system defined so far, it is not possible to distinguish the type $(s_1\to t_1)\wedge(s_2\to t_2)$ from $(s_1\vee s_2)\to(t_1\wedge t_2)$, in the sense that they both type exactly the same set of expressions. [2] The equivalence (or indistinguishablity) of the two types above states that it is not possible to have a function with two distinct behaviors chosen according to the type of the argument: the behavior is the same for inputs of type s_1 or s_2 and the intersection of the arrow types is just a way to "refine" this behavior for specific cases. In our model instead the relation:

$$s_1\vee s_2\to t_1\wedge t_2 \quad\leq\quad (s_1\to t_1)\wedge(s_2\to t_2) \tag{11.1}$$

is strict. Therefore, to ensure a true correspondence between types and language expressions, the language must provide a λ-abstraction that is in the larger type but not in the smaller one, for instance because for some argument in s_1 the λ-abstraction returns a result that is in t_1 but not in t_2. This can be done by a typecase, as for foo = $\lambda x{:}\mathsf{Int}\vee\mathsf{Bool}.\,(y=x\in\mathsf{Int})?\,y+1:\mathtt{not}(y)$ which is a function that has type $(\mathsf{Int}\to\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool})$ but not $\mathsf{Int}\vee\mathsf{Bool}\to\mathsf{Int}\wedge\mathsf{Bool}$ (notice that $\mathsf{Int}\wedge\mathsf{Bool}=\mathbb{0}$, therefore a function of this type diverges on all its arguments, which is not the case for the function foo at issue).

[2] It is not possible to prove that the two types are equivalent in the system of [60] but this can be done for the system of [7].

11.2.2. Typing

The typing rules for typecase expressions are, once again, taken verbatim from Frisch et al. [27]:

$$[\mathrm{T}^{\mathrm{C}}_{\mathsf{Case}}]\ \frac{\Gamma \vdash \mathbf{e} : t' \quad \Gamma, x : t\wedge t' \vdash \mathbf{e}_1 : s \quad \Gamma, x : \neg t\wedge t' \vdash \mathbf{e}_2 : s}{\Gamma \vdash (x = \mathbf{e} \in t)?\ \mathbf{e}_1 : \mathbf{e}_2 : s} \qquad [\mathrm{T}^{\mathrm{C}}_{\mathsf{Efq}}]\ \frac{}{\Gamma, x : \mathbb{0} \vdash \mathbf{e} : t}$$

The $[\mathrm{T}^{\mathrm{C}}_{\mathsf{Case}}]$ rule infers the type t' of the tested expression e, and then infers the types of the branch by taking into account the outcome of the test. Namely, it infers the type of e_1 under the hypothesis that x is bound to a value that was produced by e (i.e., of type t') and passed the test (i.e., of type t): that is, a value of type $t\wedge t'$; it infers the type of e_2 under the hypothesis that x is bound to a value that was produced by e (i.e., of type t') and did *not* pass the test (i.e., of type $\neg t$). Thus $[\mathrm{T}^{\mathrm{C}}_{\mathsf{Case}}]$ refines the type of the different occurrences of x in the branches to take into account the outcome of the test, a technique nowadays known as *occurrence typing* [72]. Rule $[\mathrm{T}^{\mathrm{C}}_{\mathsf{Efq}}]$ (ex falso quodlibet) is used for when in the rule $[\mathrm{T}^{\mathrm{C}}_{\mathsf{Case}}]$ either $t\wedge t'$ or $\neg t\wedge t'$ is empty: this means that the corresponding branch cannot be selected whatever the result of e is and therefore, thanks to $[\mathrm{T}^{\mathrm{C}}_{\mathsf{Efq}}]$ the branch is not typed (it is given any type, in particular the type of the other branch). For more discussion on the $[\mathrm{T}^{\mathrm{C}}_{\mathsf{Case}}]$ rule and its various implications, the reader can refer to Section 3.3 of [27] or Section 3.3 of [12].

Remark 11.1.
One could have expected the rule $[T^{C}_{\mathsf{Case}}]$ to introduce a union type so that both branches can be given different types:

$$[T^{C}_{\mathsf{Case2}}]\ \frac{\Gamma \vdash \mathbf{e} : t' \quad \Gamma, x : t\wedge t' \vdash \mathbf{e}_1 : t_1 \quad \Gamma, x : \neg t\wedge t' \vdash \mathbf{e}_2 : t_2}{\Gamma \vdash (x = \mathbf{e} \in t)?\ \mathbf{e}_1 : \mathbf{e}_2 : t_1 \vee t_2}$$

However, supposing that $\mathbf{e}_1$ and $\mathbf{e}_2$ verify the premises of this rule, since $t_1 \leq t_1 \vee t_2$ and $t_2 \leq t_1 \vee t_2$, by $[T^{C}_{\mathsf{Sub}}]$ they can both be given type $t_1 \vee t_2$, and our typing rule $[T^{C}_{\mathsf{Case}}]$ then derives type $t_1 \vee t_2$ for the whole typecase expression. Thus, the above rule can be derived in our system.

11.2.3. Denotational semantics

The definition of the denotational semantics of a typecase expression follows the same idea as for λ-abstractions: given a typecase $(x = \mathbf{e} \in t)?\ \mathbf{e}_1 : \mathbf{e}_2$, if the semantics of $\mathbf{e}$ belongs to the type t, then the result of the typecase is the semantics of $\mathbf{e}_1$ where x has been bound to the semantics of $\mathbf{e}$. Conversely, if the semantics of $\mathbf{e}$ does not belong to the type t, then the result of the typecase is computed using $\mathbf{e}_2$ by binding x to the semantics of $\mathbf{e}$.

There are, however, two problems with this approach. First, proving the substitution lemma for our denotational semantics (Lemma A.41) which is necessary to prove its soundness is only possible if the interpretations of variables in the environment ρ are kept finite. Thus, we do not directly bind x to the semantics of $\mathbf{e}$, but to a finite subset of it.

Second, suppose that $[\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho} \not\subseteq [\![t]\!]^{\mathrm{C}}$. Following the above explanation, we should have $d \in [\![(x = \mathbf{e} \in t)?\ \mathbf{e}_1 : \mathbf{e}_2]\!]^{\mathrm{C}}_{\rho} \iff d \in [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho, x\mapsto S}$ for some $S \in \mathscr{P}_f([\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho})$. However, this is type-sound only if $S \subseteq [\![\neg t]\!]^{\mathrm{C}}$, since the second branch is typed under the hypothesis that x has type $\neg t$ according to rule $[\mathrm{T}^{\mathrm{C}}_{\mathsf{Case}}]$.

We stated that, under certain restrictions, we could prove that for every value $\mathbf{v}$ and every type t, then either $\mathbf{v} \in t$ or $\mathbf{v} \in \neg t$. Provided our semantics satisfies the type soundness property, this ensures that $[\![\mathbf{v}]\!]^{\mathrm{C}} \not\subseteq [\![t]\!]^{\mathrm{C}} \implies [\![\mathbf{v}]\!]^{\mathrm{C}} \subseteq [\![\neg t]\!]^{\mathrm{C}}$. However, the same result for arbitrary expressions, which is what we need here, is a consequence of the computational soundness property (if $\mathbf{e} \rightsquigarrow^* \mathbf{v}$ then $[\![\mathbf{e}]\!]^{\mathrm{C}} = [\![\mathbf{v}]\!]^{\mathrm{C}}$), which in turn relies on the type soundness of our semantics.

Therefore, we have to break this circularity to avoid having to prove both the type soundness and computational soundness properties by mutual induction. We achieve this by simply defining the semantics of a typecase $(x = \mathbf{e} \in t)?\ \mathbf{e}_1 : \mathbf{e}_2$ as empty if the semantics of $\mathbf{e}$ is not included in $[\![t]\!]^{\mathrm{C}}$ nor in $[\![\neg t]\!]^{\mathrm{C}}$. The computational soundness of our semantics will then ensure that this case can never occur, as this would break its adequacy.

To summarize, the semantics of a typecase expression is defined as:

$$[\![(x = \mathbf{e} \in t)?\ \mathbf{e}_1 : \mathbf{e}_2]\!]^{\mathrm{C}}_{\rho} = \begin{cases} \bigcup_{S \in \mathscr{P}_f([\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho})} [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho, x \mapsto S} & \text{if } [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho} \subseteq [\![t]\!]^{\mathrm{C}} \\ \bigcup_{S \in \mathscr{P}_f([\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho})} [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho, x \mapsto S} & \text{if } [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho} \subseteq [\![\neg t]\!]^{\mathrm{C}} \\ \emptyset & \text{otherwise} \end{cases} \cup\ \Omega^{\rho}_{(x=\mathbf{e}\in t)?\ \mathbf{e}_1:\mathbf{e}_2}$$

As we explained before, we simply bind all finite subsets of the denotation of $\mathbf{e}$ to x and return the denotation of the branch selected according to the (semantic) type of the denotation of $\mathbf{e}$.

The only part we did not explain beforehand is the definition of $\Omega^{\rho}_{(x=\mathbf{e}\in t)?\ \mathbf{e}_1:\mathbf{e}_2}$, but it is straightforward. It is clear from its semantics that a typecase reduces to a stuck typecase if and only if the tested expression $\mathbf{e}$ reduces to a stuck term. Therefore, since Ω represents stuck terms, we simply define $\Omega^{\rho}_{(x=\mathbf{e}\in t)?\ \mathbf{e}_1:\mathbf{e}_2} = \{\Omega\}$ whenever $\Omega \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho}$.

11.3. Adding non-determinism

11.3.1. Non-deterministic choice

The very last ingredient to obtain the system of Frisch et al. [27] is the addition of expressions for random choice.

Frisch et al. [27] introduce a random expression generator of the form $\mathsf{rnd}(t)$, which generates randomly an expression of type t. However, defining a denotational semantics for such an expression proves to be very complex. Thus, we instead choose to define a binary choice operator, that just randomly chooses between two expressions. This has no influence on the syntactic theory (i.e., all the results of Frisch et al. [27] still hold), but will greatly simplify the definition of the denotational semantics. For this reason, we still refer to this calculus as the ℂDuce calculus.

Formally we add to the previous grammar the following production:

$$\mathbf{e} ::= \mathsf{choice}(\mathbf{e}, \mathbf{e})$$

The operational semantics of choice just randomly chooses either of its arguments yielding the following set of reductions:

$$\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2) \rightsquigarrow \mathbf{e}_1$$
$$\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2) \rightsquigarrow \mathbf{e}_2$$

The need for a choice operator is clear from considering the interpretation of function spaces. Notice indeed that functions are interpreted as relations, but we do not require them to be de-

terministic, that is, in our finite relations there may be two pairs with the same first projection but different second projections. More concretely, if $\mathbf{e}_1 : t_1$ and $\mathbf{e}_2 : t_2$ then $\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2)$ allows us to define a value that separates the type $s \to t_1 \vee t_2$ from the type $(s \to t_1) \vee (s \to t_2)$ (in our model the interpretation of the latter type is strictly contained in the interpretation of the former type), since $\lambda x{:}s.\, \mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2)$ is a value in the first type that it is not in the second type. This is formalized by the following straightforward typing rule.

$$[\mathrm{T}^{\mathrm{C}}_{\mathrm{Choice}}]\ \frac{\Gamma \vdash \mathbf{e}_1 : t \qquad \Gamma \vdash \mathbf{e}_2 : t}{\Gamma \vdash \mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2) : t}$$

Similarly to the rule $[\mathrm{T}^{\mathrm{C}}_{\mathrm{Case}}]$, this rule does not explicitly introduce a union type, as it can be introduced by using subsumption in both premises if necessary.

11.3.2. Denoting execution paths

The introduction of the random choice requires a last modification to the interpretation domain. Usually, passing from the semantics of a deterministic calculus to the semantics of its nondeterministic version can be done by interpreting expressions as the set of their possible results. Since we already interpret expressions as sets (of approximations), then one should interpret expressions as sets of sets. But since the possible results of an expressions may be infinite, then it would not be possible to define the interpretation domain by induction, as functions should return possibly infinite sets of elements.

To obviate this we could imagine to define the semantics of an expression as the set of all approximations of all its possible results. This is a reasonable solution that would fit the spirit of our approach. However, using this solution would make impossible to distinguish the semantics of $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.\, x$ from the one of $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.\, \mathsf{choice}(\bot, x)$ (where $\bot$ is an always diverging expression). This would be wrong, since the two terms have different operational semantics, insofar as the application of the former to an integer always returns that integer while the same application of the latter function may diverge.

We need a semantics in which different choices are distinguished. Therefore, if we mix in the semantics of an expression the approximations of different results, we still need to distinguish which approximation belongs to which result, like, for instance, by coloring approximations of different values with different colors. This is essentially what we propose here. The solution we choose is to interpret values as in the previous sections: constants are interpreted as singletons, pairs of values as the Cartesian product of the interpretations of the values, and λ-abstractions as the set of their finite approximations. As before, deterministic expressions will be interpreted as the single value they produce.

Multivalued expressions instead, will contain all the interpretations of all the values they may produce, but the single elements of each value will keep track of the choices the expression made to produce the value they interpret. This will be done by marking all elements of the domain by *propagation marks*, which are finite strings on the alphabet $\{l, r\}$ that record the suite of choices (*l*eft or *r*ight) performed to produce a result. The idea is that the semantics of the expression, say, $\mathsf{choice}(1, \mathsf{choice}(2, 3))$ will be the set $\{1^l, 2^{rl}, 3^{rr}\}$: the expression produces the value 1 by a left choice, the value 2 by a right choice followed by a left choice, and the value 3 by a right choice followed by another right choice. Likewise, $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.\, x$ and of $\lambda x{:}\mathsf{Int} \to \mathsf{Int}.\, \mathsf{choice}(\bot, x)$ will have distinct interpretations, since the approximations of the former will contain pairs of the form (n, n) for $n \in \mathsf{Int}$, while in the latter the pairs will be of the form (n, n^r).

The formal definition of the semantics of choice, along with the definition of the new interpretation domain, are presented in the following section.

11.4. Summary and results

We now summarize the previous sections, giving the formal definitions of the language, its type system, and its operational semantics. Then, we formalize the new interpretation domain, and give the full denotational semantics of the language.

11.4.1. A summary of the system

The *terms* $\mathbf{e} \in \mathsf{Terms}^{\mathrm{C}}$ and the *values* $\mathbf{v} \in \mathsf{Values}^{\mathrm{C}}$ of λ_{C} are those defined inductively by the following grammar:

$$\begin{array}{rcl} \mathsf{Terms}^{\mathrm{C}} \ni \mathbf{e} & ::= & c \mid x \mid \lambda^{\mathbb{I}} x.\, \mathbf{e} \mid \mathbf{e}\,\mathbf{e} \mid \pi_i\, \mathbf{e} \mid (\mathbf{e}, \mathbf{e}) \mid (x = \mathbf{e} \in t)?\, \mathbf{e} : \mathbf{e} \mid \mathsf{choice}(\mathbf{e}, \mathbf{e}) \\ \mathsf{Values}^{\mathrm{C}} \ni \mathbf{v} & ::= & c \mid \lambda^{\mathbb{I}} x.\, \mathbf{e} \mid (\mathbf{v}, \mathbf{v}) \end{array}$$

where $\mathbb{I}$ is a shorthand for $\bigwedge_{i \in I}(s_i \to t_i) \wedge \bigwedge_{n \in N} \neg(s_n \to t_n)$. As before, values are supposed to be well-typed. An ill-typed term satisfying the above production rule for $\mathbf{v}$ is considered to be a stuck term.

The operational semantics of λ_{C}, which we gradually introduced in the previous sections, are summarized as follows:

$$\begin{array}{rrcll} [\mathrm{R}^{\mathrm{C}}_{\mathsf{App}}] & (\lambda^{\mathbb{I}} x.\, \mathbf{e})\, \mathbf{v} & \rightsquigarrow & \mathbf{e}\,[\mathbf{v}/x] & \\ [\mathrm{R}^{\mathrm{C}}_{\mathsf{Proj}_i}] & \pi_i\, (\mathbf{v}_1, \mathbf{v}_2) & \rightsquigarrow & \mathbf{v}_i & i \in \{1, 2\} \\ [\mathrm{R}^{\mathrm{C}}_{\mathsf{Choice}_i}] & \mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2) & \rightsquigarrow & \mathbf{e}_i & i \in \{1, 2\} \\ [\mathrm{R}^{\mathrm{C}}_{\mathsf{CaseL}}] & (x = \mathbf{v} \in t)?\, \mathbf{e}_1 : \mathbf{e}_2 & \rightsquigarrow & \mathbf{e}_1\,[\mathbf{v}/x] & \text{if } \mathbf{v} \in t \\ [\mathrm{R}^{\mathrm{C}}_{\mathsf{CaseR}}] & (x = \mathbf{v} \in t)?\, \mathbf{e}_1 : \mathbf{e}_2 & \rightsquigarrow & \mathbf{e}_2\,[\mathbf{v}/x] & \text{if } \mathbf{v} \notin t \\ [\mathrm{R}^{\mathrm{C}}_{\mathsf{Ctx}}] & \mathscr{E}\,[\mathbf{e}] & \rightsquigarrow & \mathscr{E}\,[\mathbf{e}'] & \text{if } \mathbf{e} \rightsquigarrow \mathbf{e}' \end{array}$$

Apart from the reduction rules for typecases and random choices which we already presented earlier, this calculus uses standard reduction rules for pairs and applications.

The evaluation contexts still implement a standard leftmost outermost reduction strategy:

$$\mathscr{E} ::= [\,] \mid \mathscr{E}\,\mathbf{e} \mid \mathbf{v}\,\mathscr{E} \mid (\mathscr{E}, \mathbf{e}) \mid (\mathbf{v}, \mathscr{E}) \mid \pi_i\,\mathscr{E} \mid (x = \mathscr{E} \in t)?\, \mathbf{e} : \mathbf{e}$$

Finally, the full type system is summarized in Figure 11.1.

11.4.2. Denotational semantics

We start by summarizing the changes we introduced to the interpretation domain, to obtain the new domain $\mathscr{D}^{\mathrm{C}}$. There are two changes: first, we added distinguished inputs $\mho_{(.)}$ to finite relations. Second, we added *propagation marks* to memorize the random path that led to a result.

Formally, we define the set of *propagation marks* $\mathscr{M}$, ranged over by $\mathfrak{m}$, as the set of strings on the alphabet $\{l, r\}$. The interpretation domain for the terms of λ_{C} is then defined as follows:

Definition 11.2 (Interpretation domain $\mathscr{D}^{\mathrm{C}}$)**.** *The* interpretation domain $\mathscr{D}^{C}$ *is the set of*

$$[\mathrm{T}^{\mathrm{C}}_{\mathrm{Cst}}]\ \frac{}{\Gamma \vdash c : b_c} \qquad [\mathrm{T}^{\mathrm{C}}_{\mathrm{Var}}]\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad [\mathrm{T}^{\mathrm{C}}_{\mathrm{Sub}}]\ \frac{\Gamma \vdash \mathbf{e} : t}{\Gamma \vdash \mathbf{e} : t'}\ t \leq t'$$

$$[\mathrm{T}^{\mathrm{C}}_{\mathrm{Abs}}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash \mathbf{e} : t_i}{\Gamma \vdash \lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)} x.\,\mathbf{e} : t \wedge t'}\ \begin{array}{l} t = \bigwedge_{i\in I}(s_i \to t_i) \\ t' = \bigwedge_{n\in N} \neg(s_n \to t_n) \\ t \wedge t' \not\simeq \mathbb{0} \end{array}$$

$$[\mathrm{T}^{\mathrm{C}}_{\mathrm{App}}]\ \frac{\Gamma \vdash \mathbf{e}_1 : t \to t' \quad \Gamma \vdash \mathbf{e}_2 : t}{\Gamma \vdash \mathbf{e}_1\,\mathbf{e}_2 : t'} \qquad [\mathrm{T}^{\mathrm{C}}_{\mathrm{Pair}}]\ \frac{\Gamma \vdash \mathbf{e}_1 : t_1 \quad \Gamma \vdash \mathbf{e}_2 : t_2}{\Gamma \vdash (\mathbf{e}_1, \mathbf{e}_2) : t_1 \times t_2} \qquad [\mathrm{T}^{\mathrm{C}}_{\mathrm{Proj}_i}]\ \frac{\Gamma \vdash \mathbf{e} : t_1 \times t_2}{\Gamma \vdash \pi_i\,\mathbf{e} : t_i}$$

$$[\mathrm{T}^{\mathrm{C}}_{\mathrm{Choice}}]\ \frac{\Gamma \vdash \mathbf{e}_1 : t \qquad \Gamma \vdash \mathbf{e}_2 : t}{\Gamma \vdash \mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2) : t}$$

$$[\mathrm{T}^{\mathrm{C}}_{\mathrm{Case}}]\ \frac{\Gamma \vdash \mathbf{e} : t' \quad \Gamma, x : t \wedge t' \vdash \mathbf{e}_1 : s \quad \Gamma, x : \neg t \wedge t' \vdash \mathbf{e}_2 : s}{\Gamma \vdash (x = \mathbf{e} \in t)?\ \mathbf{e}_1 : \mathbf{e}_2 : s} \qquad [\mathrm{T}^{\mathrm{C}}_{\mathrm{Efq}}]\ \frac{}{\Gamma, x : \mathbb{0} \vdash \mathbf{e} : t}$$

FIGURE 11.1. Typing rules for λ_{C}

finite terms d produced inductively by the following grammar

$$\begin{array}{lll} d & ::= & c^{\mathfrak{m}} \mid (d,d)^{\mathfrak{m}} \mid \{(\iota,\partial),\ldots,(\iota,\partial)\}^{\mathfrak{m}} \\ \iota & ::= & \{d,\ldots,d\} \mid \mho_d \qquad\qquad (\iota \text{ not empty}) \\ \partial & ::= & d \mid \Omega \end{array}$$

where c ranges over the set $\mathscr{C}$ of constants, $\mathfrak{m}$ ranges over the set of marks $\mathscr{M}$, and where Ω and $\mho_d$ are elements that do not belong to $\mathscr{D}^C$.
We also write $\mathscr{D}^C_\Omega = \mathscr{D}^C \cup \{\Omega\}$ and $\mathscr{I}^C = (\mathscr{P}_f(\mathscr{D}^C) \setminus \{\emptyset\}) \cup \{\mho_d \mid d \in \mathscr{D}^C\}$.

We introduce some additional notation to ease the manipulation of marks. We denote the empty mark by $\varepsilon \in \mathscr{M}$, and use $\mathfrak{m}_1.\mathfrak{m}_2$ to denote the concatenation of the two marks $\mathfrak{m}_1$ and $\mathfrak{m}_2$. We note $[\partial]^{\mathfrak{m}}$ the element obtained by concatenating the string $\mathfrak{m}$ in front of the top-level mark of ∂ if any. That is, we have the following equalities:

$$\begin{aligned} [c^{\mathfrak{m}}]^{\mathfrak{m}'} &= c^{\mathfrak{m}'.\mathfrak{m}} \\ [(d_1,d_2)^{\mathfrak{m}}]^{\mathfrak{m}'} &= (d_1,d_2)^{\mathfrak{m}',\mathfrak{m}} \\ [\{(\iota_i,\partial_i) \mid i \in I\}^{\mathfrak{m}}]^{\mathfrak{m}'} &= \{(\iota_i,\partial_i) \mid i \in I\}^{\mathfrak{m}',\mathfrak{m}} \\ [\Omega]^{\mathfrak{m}} &= \Omega \end{aligned}$$

We canonically extend this notation to $\mathscr{P}(\mathscr{D}^{\mathrm{C}})$ so that if $S \subseteq \mathscr{D}^{\mathrm{C}}$, then $[S]^{\mathfrak{m}} = \{[d]^{\mathfrak{m}} \mid d \in S\}$.

Finally, we denote by $\mathsf{mark}(d)$ the top-level mark of an element $d \in \mathscr{D}^{\mathrm{C}}$, so that:

$$\mathsf{mark}(c^{\mathfrak{m}}) = \mathsf{mark}((d_1,d_2)^{\mathfrak{m}}) = \mathsf{mark}(\{(\iota_i,\partial_i) \mid i \in I\}^{\mathfrak{m}}) = \mathfrak{m}$$

and for a set $S \subseteq \mathscr{D}^{\mathrm{C}}$, we use $S\big|_{\mathfrak{m}}$ to denote the set of elements in S marked by $\mathfrak{m}$, that is:

$$S\big|_{\mathfrak{m}} = \{d \in S \mid \mathsf{mark}(d) = \mathfrak{m}\}$$

Having defined the new domain $\mathscr{D}^{\mathrm{C}}$, we can now define the new interpretation of types in this domain. We already explained how to adapt the interpretation presented in Definition 10.2 to account for our new interpretation of arrow types. The only other change is the addition of marks, but this is straightforward insofar as the new interpretation simply ignores marks.

Definition 11.3 (Set-theoretic interpretation of types in $\mathscr{D}^{\mathrm{C}}$)**.** *We define a binary predicate $(d : t)^C$ ("the element d belongs to the type t") where $d \in \mathscr{D}^C$ and $t \in \mathsf{Types}$, by induction on the pair (d, t) ordered lexicographically. The predicate is defined as follows:*

$$
\begin{aligned}
(c^{\mathfrak{m}} : b)^C &= c \in \mathbb{B}(b) \\
((d_1, d_2)^{\mathfrak{m}} : t_1 \times t_2)^C &= (d_1 : t_1)^C \text{ and } (d_2 : t_2)^C \\
(\{(\iota_i, \partial_i) \mid i \in I\} : t_1 \to t_2)^C &= \forall i \in I. \begin{cases} (\iota_i = \mho_d \ \wedge \ (d : t_1)^C) \implies (\partial_i : t_2)^C \\ (\exists d \in \iota_i.\ (d : t_1)^C) \implies (\partial_i : t_2)^C \end{cases} \\
(d : t_1 \vee t_2)^C &= (d : t_1)^C \text{ or } (d : t_2)^C \\
(d : \neg t)^C &= \text{not } (d : t)^C \\
(\partial : t)^C &= \text{false} \qquad \text{otherwise}
\end{aligned}
$$

We define the set-theoretic interpretation $[\![.]\!]^C : \mathsf{Types} \to \mathscr{P}(\mathscr{D}^C)$ *as* $[\![t]\!]^C = \{d \in \mathscr{D}^C \mid (d : t)^C\}$.

As anticipated, this new interpretation induces the same subtyping relation as the previous one:

Proposition 11.4. *For every types $t_1, t_2 \in \mathsf{Types}$, we have $t_1 \leq t_2 \iff [\![t_1]\!]^C \subseteq [\![t_2]\!]^C$.*

Proof. It is clear that $\mathscr{D}^{\mathrm{F}} \subseteq \mathscr{D}^{\mathrm{C}}$: the elements of $\mathscr{D}^{\mathrm{F}}$ simply correspond to the elements of $\mathscr{D}^{\mathrm{C}}$ that only have empty marks and do not contain inputs $\mho_d$. Conversely, we can associate to every element of $\mathscr{D}^{\mathrm{C}}$ an element of $\mathscr{D}^{\mathrm{F}}$ by the following function F:

$$
\begin{aligned}
F : \mathscr{D}^{\mathrm{C}}_{\Omega} &\mapsto \mathscr{D}^{\mathrm{F}}_{\Omega} \\
F(c^{\mathfrak{m}}) &= F(c) \\
F((d_1, d_2)^{\mathfrak{m}}) &= (F(d_1), F(d_2)) \\
F(\{(\iota_i, \partial_i) \mid i \in I\}^{\mathfrak{m}}) &= \{(F(\iota_i), F(\partial_i)) \mid i \in I\} \\
F(\Omega) &= \Omega
\end{aligned}
$$

and where F is extended to $\mathscr{I}^{\mathrm{C}}$ as follows:

$$
\begin{aligned}
F : \mathscr{I}^{\mathrm{C}} &\mapsto \mathscr{P}_f(\mathscr{D}^{\mathrm{F}}) \\
F(\{d_1, \ldots, d_n\}) &= \{F(d_1), \ldots, F(d_n)\} \\
F(\mho_d) &= \{d\}
\end{aligned}
$$

From there, it is straightforward to prove by induction on the pair (d, t) lexicographically

ordered that for every type $t \in \mathsf{Types}$:

$$\forall d \in \mathscr{D}^{\mathrm{C}}.\ d \in [\![t]\!]^{\mathrm{C}} \iff F(d) \in [\![t]\!]^{\mathrm{F}}$$
$$\forall d \in \mathscr{D}^{\mathrm{F}}.\ d \in [\![t]\!]^{\mathrm{F}} \iff d \in [\![t]\!]^{\mathrm{C}}$$

Which ensures that $[\![t]\!]^{\mathrm{C}} = \emptyset \iff [\![t]\!]^{\mathrm{F}} = \emptyset$, hence the equivalence of the subtyping relations since semantic subtyping reduces to an emptiness problem: for every types t_1, t_2, we have $t_1 \leq t_2 \overset{\text{def}}{\iff} [\![t_1]\!]^{\mathrm{F}} \subseteq [\![t_2]\!]^{\mathrm{F}} \iff [\![t_1]\!]^{\mathrm{F}} \setminus [\![t_2]\!]^{\mathrm{F}} = \emptyset \iff [\![t_1 \wedge \neg t_2]\!]^{\mathrm{F}} = \emptyset$ and similarly for $[\![.]\!]^{\mathrm{C}}$. □

We now have all the notions required to summarize the denotational semantics of the language λ_{C} in $\mathscr{D}^{\mathrm{C}}$:

Definition 11.5 (Set-theoretic interpretation of λ_{C})**.** *Let* $\mathsf{Envs} \ni \rho : \mathsf{Vars} \to \mathscr{P}_f(\mathscr{D}^C)$*. We define the set-theoretic interpretation of* λ_C *as a function* $[\![.]\!]^C_{(.)} : \mathsf{Terms}^C \to \mathsf{Envs} \to \mathscr{P}_f(\mathscr{D}^C_\Omega)$ *as follows:*

$$[\![x]\!]^C_\rho = \rho(x)$$
$$[\![c]\!]^C_\rho = \{c^\varepsilon\}$$
$$\begin{aligned}[\![\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)} x.\,\mathbf{e}]\!]^C_\rho = \{R^\varepsilon \in \mathscr{P}_f(\mathscr{I}^C \times \mathscr{D}^C_\Omega) \mid \forall(\iota,\partial)\in R.\\ \exists i\in I.\iota \subseteq [\![s_i]\!]^C \wedge \partial \in [\![\mathbf{e}]\!]^C_{\rho,x\mapsto\iota} \text{ or}\\ \forall i\in I.\iota \subseteq [\![\neg s_i]\!]^C \wedge \partial = \Omega \text{ or}\\ \iota = \mho_d \text{ where } \forall i \in I.\ d \in [\![s_i]\!]^C \implies \partial \in [\![t_i]\!]^C\\ \} \cap [\![\textstyle\bigwedge_{n\in N}\neg(s_n\to t_n)]\!]^C\end{aligned}$$
$$\begin{aligned}[\![\mathbf{e}_1\,\mathbf{e}_2]\!]^C_\rho = \textstyle\bigcup_{\mathtt{m}_1,\mathtt{m}_2\in\mathscr{M}}\{[\partial]^{\mathtt{m}_1.\mathtt{m}_2} \in \mathscr{D}^C_\Omega \mid\\ \exists S \subseteq [\![\mathbf{e}_2]\!]^C_\rho\big|_{\mathtt{m}_2}, R^{\mathtt{m}_1} \in [\![\mathbf{e}_1]\!]^C_\rho, (S,\partial)\in R\} \cup \Omega^\rho_{\mathbf{e}_1\,\mathbf{e}_2}\end{aligned}$$
$$[\![\pi_i\,\mathbf{e}]\!]^C_\rho = \{[d_i]^{\mathtt{m}} \mid (d_1,d_2)^{\mathtt{m}} \in [\![\mathbf{e}]\!]^C_\rho\} \cup \Omega^\rho_{\pi_i\,\mathbf{e}}$$
$$[\![(\mathbf{e}_1,\mathbf{e}_2)]\!]^C_\rho = \{(d_1,d_2)^\varepsilon \mid \forall i \in \{1,2\}, d_i \in [\![\mathbf{e}_i]\!]^C_\rho\} \cup \Omega^\rho_{(\mathbf{e}_1,\mathbf{e}_2)}$$
$$[\![\mathsf{choice}(\mathbf{e}_1,\mathbf{e}_2)]\!]^C_\rho = \{[\partial]^l \mid \partial \in [\![\mathbf{e}_1]\!]^C_\rho\} \cup \{[\partial]^r \mid \partial \in [\![\mathbf{e}_2]\!]^C_\rho\}$$
$$\begin{aligned}[\![(x=\mathbf{e}\in t)?\,\mathbf{e}_1:\mathbf{e}_2]\!]^C_\rho = \textstyle\bigcup_{\mathtt{m}\in\mathscr{M}}\{[\partial]^{\mathtt{m}} \mid \partial \in [\![\mathbf{e}_1]\!]^C_{\rho,x\mapsto S},\ S \subseteq [\![\mathbf{e}]\!]^C_\rho\big|_{\mathtt{m}} \subseteq [\![t]\!]^C\} \cup\\ \textstyle\bigcup_{\mathtt{m}\in\mathscr{M}}\{[\partial]^{\mathtt{m}} \mid \partial \in [\![\mathbf{e}_2]\!]^C_{\rho,x\mapsto S},\ S \subseteq [\![\mathbf{e}]\!]^C_\rho\big|_{\mathtt{m}} \subseteq [\![\neg t]\!]^C\}\\ \cup\Omega^\rho_{(x=\mathbf{e}\in t)?\,\mathbf{e}_1:\mathbf{e}_2}\end{aligned}$$

where $\Omega^\rho_{\mathbf{e}}$ *is defined as in Definition 9.3, adding the following condition:*

$$\Omega^\rho_{(x=\mathbf{e}\in t)?\,\mathbf{e}_1:\mathbf{e}_2} = \{\Omega\} \qquad \text{if } \Omega \in [\![\mathbf{e}]\!]^C_\rho$$

We already explained most of these equations in the previous sections, although the addition of marks complicates the formalism. Notice how environments are restricted to finite sets $\mathscr{P}_f(\mathscr{D}^{\mathrm{C}})$, which slightly simplifies the presentation. The propagation of marks follows the following intuition: values are already results and their denotation does not depend on a random choice, therefore, the denotation of a value will only contain empty marks at top-level. Marks are introduced via the semantics of choice expressions, which is quite intuitive: the semantics of

$\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2)$ is obtained by prepending the mark l to the elements denoting $\mathbf{e}_1$, and prepending the mark r to the elements denoting $\mathbf{e}_2$.

Note that the semantics of an expression that does not contain any choice will only contain empty marks, therefore, the above semantics is a conservative extension of the semantics we presented throughout the previous sections, before the introduction of the choice operator.

The main difficulty comes from the semantics of applications and typecases. The input sets S we consider in the semantics of applications are, intuitively, approximations of the argument. However, since we are in a call-by-value setting, for these approximations to be meaningful they must approximate the same possible value. That is, if the argument is a non-deterministic expression such as $\mathsf{choice}(2, 3)$ whose semantics is $\{2^l, 3^r\}$, then we must consider the inputs $\{2^l\}$ and $\{3^r\}$, but not $\{2^l, 3^r\}$ since this input does not correspond to the denotation of a value. Hence, the semantics of an application is defined by taking all finite approximations S of all the possible results of the argument $\mathbf{e}_2$, where a possible result corresponds to a mark $\mathfrak{m}_2$. Hence, we consider all sets $S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho}\big|_{\mathfrak{m}_2}$. Then, we propagate the marks denoting the paths that led to the generation of the input S and the relation R, to obtain the full path that produces a particular result of the application.

Typecases follow the same idea. Recall that, according to the operational semantics, to compute the result of a typecase, we first reduce the tested expression to obtain a value and then test the type of this value. The denotational semantics follows exactly the same principle: we consider all possible results of the tested expression, which we obtain by considering all sets $[\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho}\big|_{\mathfrak{m}}$ for all possible marks $\mathfrak{m} \in \mathscr{M}$. Then, we simply follow the semantics of typecases we defined in Section 11.2 on this particular set. Additionally, we propagate the mark denoting the path that led to the tested result, similarly to applications.

This means that, in particular, it is possible for the semantics to select both branches of a typecase, provided the tested expression is non-deterministic. For example, consider the expression $(x = \mathsf{choice}(\mathtt{42}, \mathtt{true}) \in \mathsf{Int})?\ x + 1 : \neg x$. The semantics of the tested expression is $\{\mathtt{42}^l, \mathtt{true}^r\}$. For the input $\{\mathtt{42}^l\}$, the semantics of the typecase evaluates to $\{\mathtt{43}^l\}$, while for the input $\{\mathtt{true}^r\}$, the semantics evaluates to $\{\mathtt{false}^r\}$. Thus, the semantics of the whole expression is $\{\mathtt{43}^l, \mathtt{false}^r\}$.

11.4.3. Properties

We now state the various properties of our semantics, which follow the results presented in the previous chapters, except they must be slightly modified to account for the non determinism brought by the choice operator.

The first result is the type soundness, which is stated exactly as in Chapter 10. The denotational interpretation of type environments $[\![.]\!]^{\mathrm{C}}$ in $\mathscr{D}^{\mathrm{C}}$ is also defined exactly as in Definition 10.13, by replacing $[\![.]\!]^{\mathrm{F}}$ by $[\![.]\!]^{\mathrm{C}}$, thus we will not restate it here.

Theorem 11.6 (Type soundness for λ_{C})**.** *For every type environment* $\Gamma \in \mathsf{TEnvs}$ *and every term* $\mathbf{e} \in \mathsf{Terms}^C$*, if* $\Gamma \vdash \mathbf{e} : t$ *then for every* $\rho \in [\![\Gamma]\!]^C$*,* $[\![\mathbf{e}]\!]^C_\rho \subseteq [\![t]\!]^C$*.*

Proof. See appendix page 276. □

The next result is the computational soundness, whose statement differs from Chapter 10, for two reasons. The first is the addition of marks and non-determinism, which means that an expression such as $\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2)$ can reduce to, say, $\mathbf{e}_1$ whose semantics is composed of the

elements of the denotation of $\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2)$ whose mark starts with l. Therefore, we do not have $[\![\mathbf{e}_1]\!]^{\mathrm{C}} = [\![\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2)]\!]^{\mathrm{C}}$ but rather $[\![\mathbf{e}_1]\!]^{\mathrm{C}} = \{d \in \mathscr{D}^{\mathrm{C}} \mid [d]^l \in [\![\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2)]\!]^{\mathrm{C}}\}$.

The second reason has been discussed in Section 11.2. Due to the addition of typecases and interfaces but the impossibility of inferring negation types for functions, the computational soundness does not hold for arbitrary terms of our language. As we hinted before, a solution is to only consider terms where λ-abstractions are sufficiently annotated to ensure that every typecase can be decided unambiguously. We do this in three steps. First, we define a predicate $\mathscr{W}_{(.)}(.)$ on terms parameterized by a set of arrow types S, which formalizes the meaning of a *well-annotated term* for S: in essence, it verifies that all the functions appearing in a term are annotated with all the types present in S. Second, we define a function $\mathsf{A}^{\rightarrow}(.)$ on terms which collects all the arrow types occurring in a term (both in interfaces and typecases). Finally, we define the set $\mathsf{Prg}_t(\Gamma)$ of well-typed terms in an environment Γ that are also well-annotated for the arrow types occurring in them.

First of all, we define the set $\mathscr{A}^{\rightarrow}$ of arrow types, or *functional atoms*:

$$\mathscr{A}^{\rightarrow} \stackrel{\text{def}}{=} \{s \rightarrow t \mid s, t \in \mathsf{Types}\}$$

We then define a predicate stating whether a term is well-annotated for a set of functional atoms.

Definition 11.7 (Well-annotated terms)**.** *We define a predicate $\mathscr{W}_{(.)}(.)$ on Terms^C, parameterized by a set $S \in \mathscr{P}_f(\mathscr{A}^{\rightarrow})$, by induction on terms as follows:*

$$\begin{array}{rcl}
\mathscr{W}_S(c) & = & \mathit{true} \\
\mathscr{W}_S(x) & = & \mathit{true} \\
\mathscr{W}_S(\lambda^{\bigwedge_{i\in I}(s_i\rightarrow t_i)\wedge\bigwedge_{n\in N}\neg(s_n\rightarrow t_n)} x.\,\mathbf{e}) & = & \mathscr{W}_S(\mathbf{e}) \wedge \forall (s \rightarrow t) \in S. \\
 & & \bigwedge_{i\in I}(s_i \rightarrow t_i) \leq s \rightarrow t \text{ or } \bigwedge_{n\in N}\neg(s_n \rightarrow t_n) \leq \neg(s \rightarrow t) \\
\mathscr{W}_S(\mathbf{e}_1\,\mathbf{e}_2) = \mathscr{W}_S((\mathbf{e}_1, \mathbf{e}_2)) = \mathscr{W}_S(\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2)) & = & \mathscr{W}_S(\mathbf{e}_1) \wedge \mathscr{W}_S(\mathbf{e}_2) \\
\mathscr{W}_S(\pi_i\,\mathbf{e}) & = & \mathscr{W}_S(\mathbf{e}) \\
\mathscr{W}_S((x = \mathbf{e} \in t)?\,\mathbf{e}_1 : \mathbf{e}_2) & = & \mathscr{W}_S(\mathbf{e}) \wedge \mathscr{W}_S(\mathbf{e}_1) \wedge \mathscr{W}_S(\mathbf{e}_2)
\end{array}$$

Most of the definition of this predicate is straightforward, and simply consists in stating that a term is well-annotated if all of its subterms also are. The only non-trivial statement concerns λ-abstractions. Intuitively, we state that a λ-abstraction is well-annotated for a set of functional atoms S if, for every atom $s \rightarrow t \in S$, the λ-abstraction at issue either has type $s \rightarrow t$ or type $\neg(s \rightarrow t)$, which we do by checking its interface.

The next step is to define a function that "collects" the functional atoms occurring in a term. We suppose given a function $\mathsf{A}^{\rightarrow}(.) : \mathsf{Types} \rightarrow \mathscr{P}_f(\mathscr{A}^{\rightarrow})$ which collects the functional atoms occurring in a type. In essence, it satisfies the following equalities:

$$\begin{array}{rcll}
\mathsf{A}^{\rightarrow}(t_1 \rightarrow t_2) & = & \{t_1 \rightarrow t_2\} \cup \mathsf{A}^{\rightarrow}(t_1) \cup \mathsf{A}^{\rightarrow}(t_2) & \\
\mathsf{A}^{\rightarrow}(t_1 \times t_2) = \mathsf{A}^{\rightarrow}(t_1 \vee t_2) & = & \mathsf{A}^{\rightarrow}(t_1) \cup \mathsf{A}^{\rightarrow}(t_2) & \\
\mathsf{A}^{\rightarrow}(\neg t) & = & \mathsf{A}^{\rightarrow}(t) & \\
\mathsf{A}^{\rightarrow}(t) & = & \emptyset & \text{otherwise}
\end{array}$$

Due to the presence of recursive types, it is not possible to consider these equalities as an inductive definition of the function $\mathsf{A}^{\rightarrow}(.)$. However, since our types are regular, we know that the set $\mathsf{A}^{\rightarrow}(t)$ exists for every type t and is finite (see [27] for more details about the well-foundedness

of this definition).

We then extend this function to obtain a function $\mathsf{A}^{\to}(.) : \mathsf{Terms}^{\mathsf{C}} \to \mathscr{P}_f(\mathscr{A}^{\to})$ by induction on terms. This definition is straightforward insofar as it simply collects all the functional atoms occurring in the interfaces and typecases of a term:

$$\begin{aligned}
\mathsf{A}^{\to}(c) &= \emptyset \\
\mathsf{A}^{\to}(x) &= \emptyset \\
\mathsf{A}^{\to}(\lambda^t x.\, \mathbf{e}) &= \mathsf{A}^{\to}(\mathbf{e}) \cup \mathsf{A}^{\to}(t) \\
\mathsf{A}^{\to}(\mathbf{e}_1\, \mathbf{e}_2) = \mathsf{A}^{\to}((\mathbf{e}_1, \mathbf{e}_2)) = \mathsf{A}^{\to}(\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2)) &= \mathsf{A}^{\to}(\mathbf{e}_1) \cup \mathsf{A}^{\to}(\mathbf{e}_2) \\
\mathsf{A}^{\to}(\pi_i\, \mathbf{e}) &= \mathsf{A}^{\to}(\mathbf{e}) \\
\mathsf{A}^{\to}((x = \mathbf{e} \in t)?\, \mathbf{e}_1 : \mathbf{e}_2) &= \mathsf{A}^{\to}(t) \cup \mathsf{A}^{\to}(\mathbf{e}) \cup \mathsf{A}^{\to}(\mathbf{e}_1) \cup \mathsf{A}^{\to}(\mathbf{e}_2)
\end{aligned}$$

This definition allows us to prove a first result. This result states that if a value is well-annotated for at least all the functional atoms occurring in a type t, then this value can either be given type t or type $\neg t$ statically.

Proposition 11.8. *For every* $\mathbf{v} \in \mathsf{Values}^{C}$, $t \in \mathsf{Types}$, *and* $S \in \mathscr{P}_f(\mathscr{A}^{\to})$, *if* $\mathsf{A}^{\to}(t) \subseteq S$ *and* $\mathscr{W}_S(\mathbf{v})$ *then either* $\mathbf{v} \in t$ *or* $\mathbf{v} \in \neg t$.

Proof. By induction on the pair $(\mathbf{v}, t)$ lexicographically ordered.

- $\underline{\mathbf{v} = c.}$ Immediate.
- $\underline{\mathbf{v} = \lambda^{\bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{n\in N} \neg(s_n \to t_n)} x.\, \mathbf{e}.}$
  - $t = c$. Then $\mathbf{v} \in \neg t$.
  - $t = t_1 \times t_2$. Then $\mathbf{v} \in \neg t$.
  - $t = t_1 \to t_2$. Since $\mathscr{W}_S(\mathbf{v})$, and by definition, $(t_1 \to t_2) \in \mathsf{A}^{\to}(t) \subseteq S$, we have either $\bigwedge_{i\in I}(s_i \to t_i) \leq t_1 \to t_2$ in which case $\mathbf{v} \in t$ or $\bigwedge_{n\in N} \neg(s_n \to t_n) \leq \neg(t_1 \to t_2)$ in which case $\mathbf{v} \in \neg t$.
  - $t = t_1 \vee t_2$. By induction hypothesis, either $\mathbf{v} \in t_1$ in which case $\mathbf{v} \in t$ by subtyping, or $\mathbf{v} \in t_2$ in which case $\mathbf{v} \in t$ by subtyping, or $\mathbf{v} \in \neg t_1$ and $\mathbf{v} \in \neg t_2$ in which case $\mathbf{v} \in \neg t$ since $\neg t \simeq \neg t_1 \wedge \neg t_2$.
  - $t = \neg t'$. By IH, we have that $\mathbf{v} \in t'$ or $\mathbf{v} \in \neg t'$ which immediately gives the result since $\neg t \simeq t'$.
  - $t = \mathbb{0}$. Then $\mathbf{v} \in \neg t$.
- $\underline{\mathbf{v} = (\mathbf{v}_1, \mathbf{v}_2).}$
  - $t = t_1 \to t_2$. Then $\mathbf{v} \in \neg t$.
  - $t = t_1 \times t_2$. By induction hypothesis, either $\mathbf{v}_1 \in t_1$ and $\mathbf{v}_2 \in t_2$ in which case $\mathbf{v} \in t$, or $\exists i \in \{1, 2\}$ such that $\mathbf{v}_i \in \neg t_i$ which yields $\mathbf{v} \in \neg t$ since $\neg t \simeq (\neg t_1 \times \mathbb{1}) \vee (\mathbb{1} \times \neg t_2) \vee \neg(\mathbb{1} \times \mathbb{1})$.
  - The other cases are proven as before.

□

We now define the sets of *programs*. For a given environment Γ and type t, a program of

type t is a well-typed term of type t in the environment Γ, that is also well-annotated for all the functional atoms it contains.

Definition 11.9 (Programs)**.** *Given a type environment Γ and a type t, we define the set* $\mathsf{Prg}_\Gamma(t)$ *of the* programs *of type t in the environment Γ as:*

$$\mathsf{Prg}_\Gamma(t) = \{\mathbf{e} \in \mathsf{Terms}^C \mid \Gamma \vdash \mathbf{e} : t \text{ and } \mathscr{W}_{\mathsf{A}\to(\mathbf{e})}(\mathbf{e})\}$$

We are now ready to state the computational soundness for our semantics. As anticipated, it is stated for programs only, and accounts for non-determinism.

Theorem 11.10 (Computational soundness for λ_C)**.** *For every term $\mathbf{e} \in \mathsf{Prg}_\Gamma(t)$ and every environment $\rho \in [\![\Gamma]\!]^C$, if $\mathbf{e} \rightsquigarrow \mathbf{e}'$ then there exists $\mathfrak{m} \in \mathscr{M}$ such that $[\![\mathbf{e}']\!]^C_\rho = \{d \in \mathscr{D}^C \mid [d]^\mathfrak{m} \in [\![\mathbf{e}]\!]^C_\rho\}$.*

Proof. See appendix page 282. □

The proof of this theorem is mostly identical to the proof of the same theorem presented in Chapter 10, and involves the same monotonicity and substitution lemmas (see Lemmas A.41 and A.40). However, it introduces a new crucial lemma which formalizes the meaning of the new inputs $\mho_{(.)}$ with respect to negation types. This lemma states that, given a function f annotated with positive and negative types, and given an approximation R of the same function but where the negative annotations have been removed, then R can be extended to obtain an approximation of f. This result only holds thanks to the addition of the new inputs $\mho_{(.)}$, and, although this is not stated explicitly, the extension of the relation R can be obtained by only extending it with pairs of the form $(\mho_d, \partial)$.

Lemma 11.11. *For every $\mathbf{v} = \lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)}x.\,\mathbf{e} \in \mathsf{Values}^C$, if $R^\varepsilon \in [\![\lambda^{\bigwedge_{i\in I}(s_i\to t_i)}x.\,\mathbf{e}]\!]^C_\rho$ then there exists $R'^\varepsilon \in [\![\mathbf{v}]\!]^C_\rho$ such that $R \setminus \mho(R) \subseteq R'$ where $\mho(R) = \{(\mho_d, \partial) \mid (\mho_d, \partial) \in R\}$.*[3]

Proof. See appendix page 282. □

The presentation of the properties of our calculus and its semantics would not be complete without a type soundness property for the operational semantics. As for the computational soundness theorem, this property only holds for well-annotated terms, as the reduction of typecases would not satisfy the subject reduction otherwise.

Theorem 11.12 (Operational type soundness of λ_C)**.** *Let $\mathbf{e} \in \mathsf{Prg}_\emptyset(t)$. If $\vdash \mathbf{e} : t$ then either $\mathbf{e}$ diverges, or there exists $\mathbf{v} \in \mathsf{Values}^C$ such that $\mathbf{e} \rightsquigarrow^* \mathbf{v}$ and $\vdash \mathbf{v} : t$.*

While our semantics and type system differs slightly from the semantics of Frisch et al. [27], this theorem is proven identically via subject reduction and progress, and we will not restate the proof here. The only major difference comes from the semantics of typecases which is much simpler in our system, and whose soundness rely on the following lemma:

Lemma 11.13. *For every $\mathbf{v} \in \mathsf{Values}^C$, $\vdash \mathbf{v} : \mathsf{type}(\mathbf{v})$. Moreover, for every $t \in \mathsf{Types}$, if $\vdash \mathbf{v} : t$ then $\mathsf{type}(\mathbf{v}) \leq t$.*

[3] Removing the pairs $(\mho_d, \partial)$ from R is not actually needed to prove the result, but makes the proof much easier, and has no impact on its usefulness.

Proof. Immediate by induction on $\mathbf{v}$ and inversion of the typing rules. □

A consequence of the subject reduction property is that for every type t and environment Γ, the set of programs $\mathsf{Prg}_\Gamma(t)$ is stable by reduction. We formalize this in the following proposition.

Proposition 11.14. *For every* $\mathbf{e} \in \mathsf{Prg}_\Gamma(t)$*, if* $\mathbf{e} \rightsquigarrow \mathbf{e}'$ *then* $\mathbf{e}' \in \mathsf{Prg}_\Gamma(t)$.

Proof. By definition of $\mathscr{W}_{\_}(.)$, it is clear that for any two sets of atoms such that $S \subseteq S'$, if $\mathscr{W}_{S'}(\mathbf{e}')$ then $\mathscr{W}_S(\mathbf{e}')$. Then, it suffices to remark that for every reduction rule $\mathsf{A}^{\rightarrow}(\mathbf{e}') \subseteq \mathsf{A}^{\rightarrow}(\mathbf{e})$, and that $\mathscr{W}_{\mathsf{A}^{\rightarrow}(\mathbf{e})}(\mathbf{e})$ ensures that $\mathscr{W}_{\mathsf{A}^{\rightarrow}(\mathbf{e})}(\mathbf{e}')$. Thus, we have $\mathscr{W}_{\mathsf{A}^{\rightarrow}(\mathbf{e}')}(\mathbf{e}')$ and this proves that $\mathbf{e}' \in \mathsf{Prg}_\Gamma(t)$ by subject reduction. □

This last proposition allows us to recursively apply the computational soundness theorem to a reduction, to deduce the following corollary:

Corollary 11.15. *For every term* $\mathbf{e} \in \mathsf{Prg}_\Gamma(t)$ *and every environment* $\rho \in [\![\Gamma]\!]^C$*, if* $\mathbf{e} \rightsquigarrow^* \mathbf{v}$ *then there exists* $\mathfrak{m} \in \mathscr{M}$ *such that* $[\![\mathbf{v}]\!]^C_\rho = \{d \in \mathscr{D}^C \mid [d]^{\mathfrak{m}} \in [\![\mathbf{e}]\!]^C_\rho\}$.

Proof. Immediate consequence of Theorem 11.10 and Proposition 11.14. □

For complexity reasons, we leave the adequacy of our semantics as a conjecture. Proving this property should be feasible by following the same strategy as in Chapter 10, by adapting the relation to support marks.

11.5. Inferring negative interfaces

This last section focuses on reconciling our system with the system of Frisch et al. [27]. The soundness of our semantics depends on the fact that terms are well-annotated, which means that the programmer must ensure that all the functions of a program are correctly annotated with all the arrow types that appear anywhere in the program. In particular, if the programmer extends a program with a new typecase, he or she must add the appropriate annotations to all the existing functions of the program.

This is, in practice, unacceptable. To solve this issue, we discuss a way to infer the negative part of interfaces automatically. In other words, for every term of the system of Frisch et al. [27] (i.e., where interfaces are conjunctions of arrow types), we can effectively produce a well-annotated term in our system that produces the same result.

For complexity reasons, a major part of our system is presented in a declarative manner, which means that we do not truly obtain a compilation algorithm from the system of Frisch et al. [27] to λ_{C}. However, this presentation gives an idea of how the algorithmic type system of CDuce can be modified to obtain a compilation algorithm to λ_{C}.

11.5.1. Source language

We consider a source language adapted from the language of Frisch et al. [27], in which function interfaces are conjunctions of positive arrow types. To summarize, we consider the terms $\mathsf{Terms}^{\mathsf{FCB}}$ and values $\mathsf{Values}^{\mathsf{FCB}}$ defined inductively by the following grammar:

$$\begin{array}{rcl} \mathsf{Terms}^{\mathsf{FCB}} \ni \mathbf{E} & ::= & c \mid x \mid \lambda^{\bigwedge_{i\in I} s_i \to t_i} x.\, \mathbf{E} \mid \mathbf{E}\,\mathbf{E} \mid \pi_i\,\mathbf{E} \mid (\mathbf{E},\mathbf{E}) \mid (x = \mathbf{E} \in t)?\,\mathbf{E} : \mathbf{E} \mid \mathsf{choice}(\mathbf{E},\mathbf{E}) \\ \mathsf{Values}^{\mathsf{FCB}} \ni \mathbf{V} & ::= & c \mid \lambda^{\bigwedge_{i\in I} s_i \to t_i} x.\, \mathbf{E} \mid (\mathbf{V},\mathbf{V}) \end{array}$$

This grammar differs slightly from the grammar of ℂDuce terms presented by Frisch et al. [27] on two aspects. First, we only consider a binary non-deterministic choice construct $\mathrm{choice}(\mathbf{E}, \mathbf{E})$, while Frisch et al. [27] consider the more general expression $\mathtt{rnd}(t)$ which picks an arbitrary expression of type t. This restriction is made necessary by our encoding of non-determinism in the denotational semantics of λ_C: by restricting non-deterministic choice to a binary construct, we are able to encode non-deterministic paths using words on a finite alphabet. Nevertheless, a binary choice operator is sufficient to ensure that distinct types can be separated, which is the main reason behind the introduction of non-determinism by Frisch et al. [27].

The second difference comes from the definition of λ-abstractions, which do not feature an explicit binder for recursive functions. This is, in fact, unnecessary: as we have discussed in the introduction chapter (§1.1.1), recursive types can be used to encode recursive functions using a fixed-point combinator.

As usual, we consider values to be closed and well-typed terms. The associated type system is defined exactly as in Figure 11.1, except for the rule $[\mathrm{T}^{\mathsf{C}}_{\mathsf{Abs}}]$ in which the negative part is now inferred:

$$[\mathrm{T}^{\mathsf{FCB}}_{\mathsf{Abs}}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash \mathbf{E} : t_i}{\Gamma \vdash \lambda^{\bigwedge_{i\in I}(s_i \to t_i)} x.\, \mathbf{E} : t \wedge t'} \quad \begin{array}{l} t = \bigwedge_{i\in I}(s_i \to t_i) \\ t' = \bigwedge_{n\in N} \neg(s_n \to t_n) \\ t \wedge t' \not\simeq \mathbb{0} \end{array}$$

As seen in the above rule, to distinguish the two type systems, we denote this new type system using a superscript $^{\mathsf{FCB}}$.

The operational semantics of the source language also follows the semantics presented in Section 11.4, except for the reduction of typecases. To define the operational semantics of typecases, Frisch et al. [27] introduce the notion of *type schemes*, so as to avoid invoking the type system in the reduction rules. To ease the presentation, we immediately define the semantics of typecases in terms of the type system, as the soundness properties presented by Frisch et al. [27] guarantee that the two definitions are equivalent.

$$\begin{array}{lllll} [\mathrm{R}^{\mathsf{FCB}}_{\mathsf{CaseL}}] & (x = \mathbf{V} \in t)?\, \mathbf{E}_1 : \mathbf{E}_2 & \rightsquigarrow & \mathbf{E}_1\,[\mathbf{V}/x] & \text{if } \vdash \mathbf{V} : t \\ [\mathrm{R}^{\mathsf{FCB}}_{\mathsf{CaseR}}] & (x = \mathbf{V} \in t)?\, \mathbf{E}_1 : \mathbf{E}_2 & \rightsquigarrow & \mathbf{E}_2\,[\mathbf{V}/x] & \text{if } \nvdash \mathbf{V} : t \end{array}$$

As before, we use a superscript $^{\mathsf{FCB}}$ to distinguish the operational semantics of the source language from the operational semantics of λ_C.

We also introduce the following lemma from Frisch et al. [27] which states that every value in their system is either of type t or of type $\neg t$, for every type t. The fact that this lemma does not hold in our system is the very reason the inference of negative arrows is necessary.

Lemma 11.16. *For every value* $\mathbf{V} \in \mathsf{Values}^{\mathsf{FCB}}$ *and every type* $t \in \mathsf{Types}$*, either* $\vdash V : t$ *or* $\vdash V : \neg t$.

We now proceed with the presentation of the compilation system and its properties.

11.5.2. Annotation and results

Having defined the source language, we now present declaratively a compilation system to the calculus λ_C. The goal of this compilation system is to show that, to every well-typed term of type t of the source language, it is possible to associate a *program* of λ_C that preserves its type, its semantics, and its reduction.

This process is done in two steps. Given a well-typed term $\mathbf{E}$ of the source language, we first follow its type derivation to produce a well-typed term $\mathbf{e}$ of λ_C by adding all the necessary

$$[\mathrm{C}^{\mathrm{C}}_{\mathrm{Cst}}]\ \frac{}{\Gamma \vdash c \rightsquigarrow c : b_c} \qquad [\mathrm{C}^{\mathrm{C}}_{\mathrm{Var}}]\ \frac{}{\Gamma \vdash x \rightsquigarrow x : \Gamma(x)} \qquad [\mathrm{C}^{\mathrm{C}}_{\mathrm{Sub}}]\ \frac{\Gamma \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t}{\Gamma \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t'}\ t \leq t'$$

$$[\mathrm{C}^{\mathrm{C}}_{\mathrm{Abs}}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t_i}{\Gamma \vdash \lambda^{\bigwedge_{i\in I}(s_i \to t_i)} x.\, \mathbf{E} \rightsquigarrow \lambda^{\bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{n\in N} \neg(s_n \to t_n)} x.\, \mathbf{e} : t \wedge t'}\ \begin{array}{l} t = \bigwedge_{i\in I}(s_i \to t_i) \\ t' = \bigwedge_{n\in N} \neg(s_n \to t_n) \\ t \wedge t' \not\simeq \mathbb{0} \end{array}$$

$$[\mathrm{C}^{\mathrm{C}}_{\mathrm{App}}]\ \frac{\Gamma \vdash \mathbf{E}_1 \rightsquigarrow \mathbf{e}_1 : t \to t' \quad \Gamma \vdash \mathbf{E}_2 \rightsquigarrow \mathbf{e}_2 : t}{\Gamma \vdash \mathbf{E}_1\, \mathbf{E}_2 \rightsquigarrow \mathbf{e}_1\, \mathbf{e}_2 : t'} \qquad [\mathrm{C}^{\mathrm{C}}_{\mathrm{Pair}}]\ \frac{\Gamma \vdash \mathbf{E}_1 \rightsquigarrow \mathbf{e}_1 : t_1 \quad \Gamma \vdash \mathbf{E}_2 \rightsquigarrow \mathbf{e}_2 : t_2}{\Gamma \vdash (\mathbf{E}_1, \mathbf{E}_2) \rightsquigarrow (\mathbf{e}_1, \mathbf{e}_2) : t_1 \times t_2}$$

$$[\mathrm{C}^{\mathrm{C}}_{\mathrm{Proj}_i}]\ \frac{\Gamma \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t_1 \times t_2}{\Gamma \vdash \pi_i\, \mathbf{E} \rightsquigarrow \pi_i\, \mathbf{e} : t_i} \qquad [\mathrm{C}^{\mathrm{C}}_{\mathrm{Choice}}]\ \frac{\Gamma \vdash \mathbf{E}_1 \rightsquigarrow \mathbf{e}_1 : t \qquad \Gamma \vdash \mathbf{E}_2 \rightsquigarrow \mathbf{e}_2 : t}{\Gamma \vdash \mathsf{choice}(\mathbf{E}_1, \mathbf{E}_2) \rightsquigarrow \mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2) : t}$$

$$[\mathrm{C}^{\mathrm{C}}_{\mathrm{Case}}]\ \frac{\Gamma \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t' \quad \Gamma, x : t \wedge t' \vdash \mathbf{E}_1 \rightsquigarrow \mathbf{e}_1 : s \quad \Gamma, x : \neg t \wedge t' \vdash \mathbf{E}_2 \rightsquigarrow \mathbf{e}_2 : s}{\Gamma \vdash (x = \mathbf{E} \in t)?\, \mathbf{E}_1 : \mathbf{E}_2 \rightsquigarrow (x = \mathbf{e} \in t)?\, \mathbf{e}_1 : \mathbf{e}_2 : s}$$

$$[\mathrm{C}^{\mathrm{C}}_{\mathrm{Efq}}]\ \frac{}{\Gamma, x : \mathbb{0} \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t}$$

FIGURE 11.2. Type-directed first annotation step of $\mathsf{Terms}^{\mathsf{FCB}}$ to $\mathsf{Terms}^{\mathrm{C}}$

negative interfaces introduced during the derivation. The term we obtain in this way is well-typed but not necessarily well-annotated, since we do not explicitly consider the functional atoms that appear in typecases. The second step consists therefore in gathering all the atoms present in the compiled term $\mathbf{e}$ and adding them to the interfaces appearing in $\mathbf{e}$.

First annotation step

The first step is presented in Figure 11.2 in a declarative style. This presentation introduces statements of the form $\Gamma \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t$, which state that in a type environment Γ, the term $\mathbf{E} \in \mathsf{Terms}^{\mathsf{FCB}}$ compiles to the term $\mathbf{e} \in \mathsf{Terms}^{\mathrm{C}}$ of type t. The definition of this system is fairly straightforward, insofar as it mimics the type system presented in Figure 11.1. The only crucial part is the compilation rule for λ-abstractions $[\mathrm{C}^{\mathrm{C}}_{\mathrm{Abs}}]$, which bridges the gap between the type system of λ_{C} and the type system of Frisch et al. [27] by adding the inferred negative types.

Before continuing with the second step, we prove several properties of this declarative compilation system. The first, most important property, is completeness: every well-typed term of the source language can be annotated to obtain a well-typed term (of the same type) of λ_{C}. As anticipated, the major difficulty in proving this result comes from the annotation of λ-abstractions. This is due to the fact that the rule $[\mathrm{C}^{\mathrm{C}}_{\mathrm{Abs}}]$ requires the body $\mathbf{E}$ of the abstraction to compile to the same expression $\mathbf{e}$, independently of the type assigned to x. Therefore, we need to prove that if an expression compiles to two different terms under two different type environments, then it is possible to "merge" the two terms to obtain a unique annotated term under the two environments. This, in particular, requires to prove that negative interfaces can safely be merged together, which is formalized by the following lemma.

Lemma 11.17. *For every* $t = \bigwedge_{i\in I}(s_i \to t_i) \in \mathsf{Types}$ *and* $t' = \bigwedge_{n\in N} \neg(s_n \to t_n) \in \mathsf{Types}$, *if* $\forall n \in N.\ t \wedge \neg(s_n \to t_n) \not\simeq \mathbb{0}$ *then* $t \wedge t' \not\simeq \mathbb{0}$.

Proof. By induction on the size of N. If $N = \emptyset$ the result is immediate. Otherwise, suppose that $t'' = t' \wedge \neg(s_0 \to t_0)$, and that $\forall n \in N.\ t \wedge \neg(s_n \to t_n) \not\leq \mathbb{0}$ and $t \wedge \neg(s_0 \to t_0) \not\leq \mathbb{0}$. We need to prove that $t \wedge t'' \not\leq \mathbb{0}$.
By IH, we have $t \wedge t' \not\leq \mathbb{0}$. Let $R^{\mathfrak{m}} \in [\![t \wedge t']\!]^{\mathrm{C}}$. Since $t \wedge \neg(s_0 \to t_0) \not\leq \mathbb{0}$, we can also take $R'^{\mathfrak{m}'} \in [\![t \wedge \neg(s_0 \to t_0)]\!]^{\mathrm{C}}$. Consider then $R_0 = R \cup R'$. By Definition 11.3, we immediately verify that $R_0^{\varepsilon} \in [\![t]\!]^{\mathrm{C}}$ by construction. Now let $n \in N$. By hypothesis, $R^{\mathfrak{m}} \in [\![\neg(s_n \to t_n)]\!]^{\mathrm{C}}$. By Definition 11.3, there are two cases:

- There exists $(S, \partial) \in R$ such that $S \cap [\![s_n]\!]^{\mathrm{C}} \neq \emptyset$ and $\partial \notin [\![t_n]\!]^{\mathrm{C}}$. Since $(S, \partial) \in R_0$, we immediately have $R_0^{\varepsilon} \in [\![\neg(s_n \to t_n)]\!]^{\mathrm{C}}$.
- There exists $(\mho_d, \partial) \in R$ such that $d \in [\![s_n]\!]^{\mathrm{C}}$ and $\partial \notin [\![t_n]\!]^{\mathrm{C}}$. The same reasoning yields that $R_0^{\varepsilon} \in [\![\neg(s_n \to t_n)]\!]^{\mathrm{C}}$.

The same reasoning on R' and $\neg(s_0 \to t_0)$ yields that $R_0^{\varepsilon} \in [\![\neg(s_0 \to t_0)]\!]^{\mathrm{C}}$, hence $R_0^{\varepsilon} \in [\![t \wedge t'']\!]^{\mathrm{C}}$. □

Having proven that negative interfaces can be merged without producing an empty type, we can now prove that it is always possible to merge two compiled terms obtained under two different type environments.

Lemma 11.18. *For every term* $\mathbf{E} \in \mathsf{Terms}^{\mathsf{FCB}}$*, if* $\Gamma \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t$ *and* $\Gamma' \vdash \mathbf{E} \rightsquigarrow \mathbf{e}' : t'$ *then there exists* $\hat{\mathbf{e}} \in \mathsf{Terms}^{C}$ *such that* $\Gamma \vdash \mathbf{E} \rightsquigarrow \hat{\mathbf{e}} : t$ *and* $\Gamma' \vdash \mathbf{E} \rightsquigarrow \hat{\mathbf{e}} : t'$.

Proof. By induction on $\mathbf{E}$ and case analysis over the compilation rules used for $\Gamma \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t$ and $\Gamma' \vdash \mathbf{E} \rightsquigarrow \mathbf{e}' : t'$. The only non-trivial case is $\mathbf{E} = \lambda^{\bigwedge_{i\in I}(s_i \to t_i)} x.\, \mathbf{E}'$ when $[\mathrm{C}^{\mathsf{FCB}}_{\mathsf{Abs}}]$ is used for both statements.
In that case, we have $\Gamma \vdash \mathbf{E} \rightsquigarrow \lambda^{\bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{n\in N} \neg(s_n \to t_n)} x.\, \mathbf{e} : t \wedge t'$ and $\Gamma' \vdash \mathbf{E} \rightsquigarrow \lambda^{\bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{n\in N'} \neg(s_n \to t_n)} x.\, \mathbf{e} : t \wedge t''$ where $t \wedge t' \not\leq \mathbb{0}$, $t \wedge t'' \not\leq \mathbb{0}$, and $t = \bigwedge_{i\in I}(s_i \to t_i)$, $t' = \bigwedge_{n\in N} \neg(s_n \to t_n)$, and $t'' = \bigwedge_{n\in N'} \neg(s_n \to t_n)$.
By Lemma 11.17, we have $t \wedge t' \wedge t'' \not\leq \mathbb{0}$. Thus, we deduce that $\Gamma \vdash \mathbf{E} \rightsquigarrow \lambda^{t \wedge t' \wedge t''} x.\, \mathbf{e} : t \wedge t' \wedge t''$ and $\Gamma' \vdash \mathbf{E} \rightsquigarrow \lambda^{t \wedge t' \wedge t''} x.\, \mathbf{e} : t \wedge t' \wedge t''$. The result follows by application of $[\mathrm{C}^{\mathsf{FCB}}_{\mathsf{Sub}}]$. □

Using the above lemma, proving the completeness of the compilation system presented in Figure 11.2 becomes straightforward.

Lemma 11.19. *For every term* $\mathbf{E} \in \mathsf{Terms}^{\mathsf{FCB}}$*, if* $\Gamma \vdash \mathbf{E} : t$ *in* $[T^{\mathsf{FCB}}]$*, then there exists* $\mathbf{e} \in \mathsf{Terms}^{C}$ *such that* $\Gamma \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t$.

Proof. Immediate by induction on the derivation $\Gamma \vdash \mathbf{E} : t$, taking the compilation rules corresponding to the typing rules used, and using Lemma 11.18 to merge the premises of $[\mathrm{T}^{\mathsf{FCB}}_{\mathsf{Abs}}]$. □

The last two lemmas we present for the first step are straightforward results which state that the compilation system properly mimics the type system.

Lemma 11.20. *For every term* $\mathbf{E} \in \mathsf{Terms}^{\mathsf{FCB}}$*, if* $\Gamma \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t$*, then* $\Gamma \vdash \mathbf{e} : t$.

Proof. Immediate by induction on the derivation $\Gamma \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t$, taking the typing rules of [$\mathrm{T^C}$] corresponding to the compilation rules used. □

Lemma 11.21. *For every value* $\mathbf{V} \in \mathsf{Values}^{\mathsf{FCB}}$*, if* $\vdash \mathbf{V} : t$ *and* $\vdash \mathbf{V} \rightsquigarrow \mathbf{v} : t'$ *then* $t \wedge t' \not\leq \mathbb{0}$.

Proof. Once again immediate by cases on $\mathbf{V}$ and over the last rule used to derive $\vdash \mathbf{V} \rightsquigarrow \mathbf{v} : t'$. □

Second annotation step

We now present the second annotation step, whose goal is to produce a well-annotated term (that is, a program) from a well-typed term of λ_{C}. This step is defined algorithmically as a function $(\!|.|\!)_{(.)}$ which takes a term of λ_{C} and a set of functional atoms S, and returns a term that is well-annotated for S. Formally, this function is defined as follows:

Definition 11.22 (Compilation of $\mathsf{Terms}^{\mathrm{C}}$ to programs)**.** *Let* $S = \{s_j \to t_j \mid j \in J\} \in \mathscr{P}_f(\mathscr{A}^{\to})$*. We define the function* $(\!|.|\!)_{(.)} : \mathsf{Terms}^{C} \to \mathscr{P}_f(\mathscr{A}^{\to}) \to \mathsf{Terms}^{C}$ *by induction on* Terms^{C} *as follows:*

$$
\begin{aligned}
(\!|c|\!)_S &\stackrel{\text{def}}{=} c \\
(\!|x|\!)_S &\stackrel{\text{def}}{=} x \\
(\!|\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)} x.\, \mathbf{e}|\!)_S &\stackrel{\text{def}}{=} \lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N'}\neg(s_n\to t_n)} x.\, (\!|\mathbf{e}|\!)_S \\
&\qquad \text{where } N' = \{j \in J \mid \bigwedge_{i\in I}(s_i \to t_i) \wedge \neg(s_j \to t_j) \neq \emptyset\} \cup N \\
(\!|\mathbf{e}_1\, \mathbf{e}_2|\!)_S &\stackrel{\text{def}}{=} (\!|\mathbf{e}_1|\!)_S\, (\!|\mathbf{e}_2|\!)_S \\
(\!|\pi_i\, \mathbf{e}|\!)_S &\stackrel{\text{def}}{=} \pi_i\, (\!|\mathbf{e}|\!)_S \\
(\!|(\mathbf{e}_1, \mathbf{e}_2)|\!)_S &\stackrel{\text{def}}{=} ((\!|\mathbf{e}_1|\!)_S, (\!|\mathbf{e}_2|\!)_S) \\
(\!|(x = \mathbf{e} \in t)?\, \mathbf{e}_1 : \mathbf{e}_2|\!)_S &\stackrel{\text{def}}{=} (x = (\!|\mathbf{e}|\!)_S \in t)?\, (\!|\mathbf{e}_1|\!)_S : (\!|\mathbf{e}_2|\!)_S \\
(\!|\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2)|\!)_S &\stackrel{\text{def}}{=} \mathsf{choice}((\!|\mathbf{e}_1|\!)_S, (\!|\mathbf{e}_2|\!)_S)
\end{aligned}
$$

As for the first step, the definition of this function is fairly straightforward, and the only non-trivial case is the case of λ-abstractions. The idea is behind this case is simple: given an abstraction $\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)} x.\, \mathbf{e}$ and a set of functional atoms S, we simply add to the negative part of the interface all the negations of all the atoms present in S that do not make the interface empty. Note that Lemma 11.17 ensures that this step is properly defined, independently of the order in which we consider the atoms in S.

We first prove that this function properly fulfills its purpose, in the sense that when given a term $\mathbf{e}$ and a set of atoms S, it produces a term that is well-annotated (according to Definition 11.7) for the set S.

Lemma 11.23. *For every term* $\mathbf{e} \in \mathsf{Terms}^{C}$*, for every* $S \in \mathscr{P}_f(\mathscr{A}^{\to})$*, it holds that* $\mathscr{W}_S((\!|\mathbf{e}|\!)_S)$.

Proof. By induction on $\mathbf{e}$. Most cases are immediate by induction following Definition 11.22 and Definition 11.7. The only non-trivial case is $\mathbf{e} = \lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)}x.\,\mathbf{e}'$, where $(\!|\mathbf{e}|\!)_S = \lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N'}\neg(s_n\to t_n)}x.\,(\!|\mathbf{e}'|\!)_S$. By induction hypothesis, we have $\mathscr{W}_S((\!|\mathbf{e}'|\!)_S)$. Moreover, for every $(s\to t)\in S$, we distinguish two cases:

- $\bigwedge_{i\in I}(s_i\to t_i)\wedge\neg(s\to t)\leq\mathbb{0}$. This yields that $\bigwedge_{i\in I}(s_i\to t_i)\leq s\to t$, which satisfies the condition of Definition 11.7.
- $\bigwedge_{i\in I}(s_i\to t_i)\wedge\neg(s\to t)\not\leq\mathbb{0}$. By Definition 11.22, we deduce that there exists $n\in N'$ such that $s_n\to t_n\equiv s\to t$. Thus, $\bigwedge_{n\in N'}\neg(s_n\to t_n)\leq\neg(s\to t)$, hence the result by Definition 11.7.

□

An immediate corollary of this lemma is that, if we take S to be the set of atoms appearing in a term, the term we obtain by this second step is a program, since it is well-annotated for all the atoms appearing in it.

Corollary 11.24. *For every term $\mathbf{e}\in\mathsf{Terms}^C$ such that $\Gamma\vdash\mathbf{e}:t$, for every $S\in\mathscr{P}_f(\mathscr{A}^{\to})$ such that $\mathsf{A}^{\to}(\mathbf{e})\subseteq S$, it holds that $(\!|\mathbf{e}|\!)_S\in\mathsf{Prg}_\Gamma(t)$.*

Proof. By Lemma 11.23 we have that $\mathscr{W}_S((\!|\mathbf{e}|\!)_S)$, which ensures that $\mathscr{W}_{\mathsf{A}^{\to}(\mathbf{e})}((\!|\mathbf{e}|\!)_S)$ since $\mathsf{A}^{\to}(\mathbf{e})\subseteq S$. Then, by cases on $\mathbf{e}$, it is immediate to see by Definition 11.22 that $\Gamma\vdash\mathbf{e}:t$ implies $\Gamma\vdash(\!|\mathbf{e}|\!)_S:t$. Hence, $(\!|\mathbf{e}|\!)_S\in\mathsf{Prg}_\Gamma(t)$. □

We now connect the two steps, by proving that it is possible to associate to every term $\mathbf{E}$ of the source language a program of λ_{C} by using only the system presented in Figure 11.2. Intuitively, this result holds since all the annotations added by the function presented in Definition 11.22 can be added immediately during the first step.

Lemma 11.25. *For every term $\mathbf{E}\in\mathsf{Terms}^{\mathsf{FCB}}$, if $\Gamma\vdash\mathbf{E}\rightsquigarrow\mathbf{e}:t$, then for every $S\in\mathscr{P}_f(\mathscr{A}^{\to})$, $\Gamma\vdash\mathbf{E}\rightsquigarrow(\!|\mathbf{e}|\!)_S:t$.*

Proof. By induction on $\mathbf{E}$ and case disjunction over the last rule used for $\Gamma\vdash\mathbf{E}\rightsquigarrow\mathbf{e}:t$. Once again, all cases are immediate except $[\mathrm{C}^{\mathsf{FCB}}_{\mathsf{Abs}}]$. In this case, we have $\mathbf{E}=\lambda^{\bigwedge_{i\in I}(s_i\to t_i)}x.\,\mathbf{E}'$, $\mathbf{e}=\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)}x.\,\mathbf{e}'$, and $\Gamma\vdash\mathbf{E}\rightsquigarrow\mathbf{e}:\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)$ where $\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)\not\leq\mathbb{0}$ and $\forall i\in I,\ \Gamma,x:s_i\vdash\mathbf{E}'\rightsquigarrow\mathbf{e}':t_i$.
By induction hypothesis, $\forall i\in I,\ \Gamma,x:s_i\vdash\mathbf{E}'\rightsquigarrow(\!|\mathbf{e}'|\!)_S:t_i$. Moreover, by Definition 11.22 and Lemma 11.17, we have $(\!|\mathbf{e}|\!)_S=\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N'}\neg(s_n\to t_n)}x.\,(\!|\mathbf{e}'|\!)_S$ where $\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N'}\neg(s_n\to t_n)\not\leq\mathbb{0}$. Hence $\Gamma\vdash\mathbf{E}\rightsquigarrow(\!|\mathbf{e}|\!)_S:\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N'}\neg(s_n\to t_n)$ and the result follows by application of $[\mathrm{C}^{\mathsf{FCB}}_{\mathsf{Sub}}]$. □

Finally, we prove the main theorem of this section: that for every well-typed term of the source language there exists an annotated term of λ_{C} that preserves its semantics by bisimulation. This theorem guarantees that the denotational semantics of λ_{C} properly model the operational behaviour of ℂDuce , provided we add enough annotations to lift any ambiguity about the resolution of typecases.

Theorem 11.26 (Soundness and completeness of compilation)**.** *For every term* $\mathbf{E} \in \mathsf{Terms}^{\mathsf{FCB}}$*, if* $\vdash \mathbf{E} : t$ *then there exists* $\mathbf{e} \in \mathsf{Prg}_{\emptyset}(t)$ *such that* $\vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t$ *and the following holds:*

1. $\mathbf{E} \rightsquigarrow \mathbf{E}' \implies \exists \mathbf{e}' \in \mathsf{Terms}^{C}.\ \mathbf{e} \rightsquigarrow \mathbf{e}'$ *and* $\vdash \mathbf{E}' \rightsquigarrow \mathbf{e}' : t$
2. $\mathbf{e} \rightsquigarrow \mathbf{e}' \implies \exists \mathbf{E}' \in \mathsf{Terms}^{\mathsf{FCB}}.\ \mathbf{E} \rightsquigarrow \mathbf{E}'$ *and* $\vdash \mathbf{E}' \rightsquigarrow \mathbf{e}' : t$

Proof. See appendix page 284. □

11.6. Summary

In this chapter, we enriched the language presented in the previous chapters with function interfaces, non-deterministic choice, and typecases. These are the three necessary components to ensure that the interpretation of types we presented in Section 2.3 coincides with the interpretation of types as sets of values. We summarize the main contributions of this chapter here.

Encoding of non-determinism. The first, arguably simplest, contribution of this chapter is the encoding of non-determinism in the interpretation domain $\mathscr{D}^{\mathrm{C}}$. By extending the interpretation domain with *marks*, which are finite strings over the alphabet $\{l, r\}$, we were able to provide a denotational semantics of non-deterministic choice expressions where the mark of a denotation represents the random path that led to a result. We showed how to propagate this information through applications and typecases, based on the reduction strategy we adopted for our calculus.

Typecases and programs. With the addition of typecases comes a new difficulty: for the operational semantics of typecases to be sound, it is necessary to ensure that for every value $\mathbf{v}$ and every type t appearing in a typecase, $\mathbf{v}$ can either be given type t or type $\neg t$. To achieve this, Frisch et al. [27] introduce a new typing rule for λ-abstractions that can derive arbitrary negations of arrow types. However, we have shown that this rule is incompatible with the type soundness of our denotational semantics. To solve this problem, we modified both the syntax and the type system of our language to explicitly introduce negative arrow types, and we introduced the notion of *well-annotated* terms. In a well-annotated term, we proved that every λ-abstraction can either be given type t or type $\neg t$ for every type t that appears in a typecase of the term, thus ensuring the soundness of our semantics.

The semantics of functions. Following the previous point, λ-abstractions in λ_{C} are annotated with an intersection of arrow types or negations of arrow types, which we call their *interface*. We showed how to account for the presence of an explicit return type in the denotational semantics of abstractions, so as to ensure that two functions that produce the same results on the same inputs but are annotated differently have distinct denotational semantics. Additionally, we showed that this is strongly tied to the interpretation of the negative part of an interface: as long as this does not make it empty, adding a negation type to the interface of a function simply amounts to taking a subset of its denotational semantics, without any loss of information.

The semantics of $\mathbb{C}$Duce . Finally, since we introduced an important restriction on the type system of $\mathbb{C}$Duce originally presented by Frisch et al. [27], we discussed in Section 11.5 how to

reconcile our semantics with the semantics of ℂDuce . Given a term of ℂDuce , we showed that it is possible to add all the necessary negation of arrow types to obtain a term of λ_C that is well-annotated and that preserves both its reduction and its type.

Chapter 12.

Denotational semantics for gradual typing

"The pursuit of truth and beauty is a sphere of activity in which we are permitted to remain children all our lives."

Albert Einstein

In this chapter, we go back to the interpretation of types and the denotational semantics presented in Chapter 10, and extend them to support gradual typing. We use this new semantics to get an insight into how gradual types behave in the presence of set-theoretic types.

Chapter Outline

Section 12.1 We extend the interpretation domain $\mathscr{D}^{\mathrm{F}}$ with *tags*, to distinguish between *boxed* and *unboxed* values, where unboxed values belong to the static part of a type, and boxed values belong to its dynamic part. We show how to interpret types as elements of this new domain $\mathscr{D}^{\mathrm{G}}$, and deduce several relations with interesting properties, namely *subtyping* and *precision*.

Section 12.2 We present the syntax and declarative type system of a gradually-typed language λ_{G}. We limit ourselves to a simply typed λ-calculus without pairs, to focus on the main difficulty which is the denotational interpretation of applications and casts. After briefly summarizing the operational semantics for λ_{G} (which is a fairly standard operational semantics for a simply typed cast calculus), we conclude this section by giving its formal denotational semantics.

Section 12.3 As in the previous chapters, we state and prove the two main properties of our semantics, namely the type soundness and the computational soundness.

12.1. The set-theoretic semantics of gradual types

In the previous chapters, we have laid the groundwork necessary to give a set-theoretic interpretation of gradual types and the denotational semantics of a gradually-typed language. As adding gradual typing to a fully-featured language such as the language of Chapter 11 would be a huge undertaking, we chose to take a few steps back and use the interpretation domain presented in Chapter 10 as the basis for our work on gradual types.

Throughout this section, we consider gradual set-theoretic types as defined in Chapter 5, except we restrict ourselves to monomorphic types. As a reminder, the set GTypes is defined as follows.

Definition 12.1 (Gradual types)**.** *The set* GTypes *of* gradual types *is the set of terms τ generated coinductively by the following grammar:*

$$\mathsf{GTypes} \ni \tau \quad ::= \quad ? \mid \tau \times \tau \mid \tau \to \tau \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0} \quad \textit{gradual types}$$

where b ranges over $\mathscr{B}$, and that satisfy the following two conditions:

- (regularity) *the term has a finite number of different sub-terms;*
- (contractivity) *every infinite branch of a type contains an infinite number of occurrences of the $\times$ or $\to$ type constructors.*

To give a set-theoretic interpretation of these types, the first step is to extend the interpretation domain $\mathscr{D}^{\mathrm{F}}$. We then study various relations induced by this interpretation, namely, *subtyping* and *precision*. While the definition of subtyping follows the same strategy as in the previous chapters, the set-theoretic definition of precision is entirely novel. By formalizing the notion of *boxed* and *unboxed* values, this new interpretation characterizes set-theoretically the very essence of gradual typing.

12.1.1. An interpretation domain for gradual types

To understand precisely how to interpret gradual types set-theoretically, we first need to think about the behaviour of values in a gradually-typed program. Recall that the elements of our interpretation domain represent the results of computations. In the presence of gradual typing, new results are possible. First, a computation may end by a cast failure that will blame some particular position in the source code. As customary, positions to blame are denoted by labels. Thus, we add "blames" to the possible results of a computation. Formally, we suppose given a set of blame labels $\mathscr{L}$, and we define the set $\mathbb{B}\mathsf{lame}$ of blames as follows:

$$\mathbb{B}\mathsf{lame} \stackrel{\text{def}}{=} \{\mathsf{blame}\ \ell \mid \ell \in \mathscr{L}\} \cup \{\mathsf{blame}\ \overline{\ell} \mid \ell \in \mathscr{L}\}$$

where labels are used to denote the point of failure and have a polarity that indicates whether the failure at point ℓ is due to the context ($\mathsf{blame}\ \overline{\ell}$) or to the expression in that context ($\mathsf{blame}\ \ell$). We write p to denote a blame label independently of its polarity, and use the involutory operation $\overline{p}$ to reverse the polarity of a label.

Secondly, a computation may also return a value of unknown type. For instance, if we feed the function $\lambda x{:}\mathsf{Int}.\, x$ with an integer value, then it will return an integer value (i.e., the same integer), but if we feed the function $\lambda x{:}?.\, x$ with the same integer, it will return a value of unknown type. Even if both functions return the same integer, the one returned by the former can be used only in contexts where an integer is expected, while the one returned by the latter can be used in any context —e.g., one that expects a boolean— even though it may then also dynamically fail. As we see, our old values can play two different roles according to whether they are used with a static or a dynamic type, and the semantics must distinguish these two roles.

Another way to understand this is by seeing a gradually typed program as having two parts: a dynamically typed part and a statically typed one. When a value travels through the statically typed part, it can be safely handled since the type-checker has already ensured that no type error will occur along its path. This is, in essence, the reason we can use type erasure in statically typed programs. However, when this value crosses into the dynamically typed part of the program, anything can happen. Therefore, it must carry its type information so that its type can be

checked at runtime. In other words, the value becomes "boxed", or "annotated" with some runtime type information. When crossing back into the static world, and once this information has been verified as sound at the boundary, it can safely be discarded: the value is then "unboxed".

From a domain point of view, we distinguish between "boxed" and "unboxed" values by annotating them with a tag. This tag can be either ? or !, indicating that the value plays a dynamic (i.e., the value is boxed) or a static role (i.e., the value is unboxed), respectively.

This yields the following definition of the new domain $\mathscr{D}^{\mathrm{G}}$:

Definition 12.2 (Interpretation domain $\mathscr{D}^{\mathrm{G}}$)**.** *The* interpretation domain $\mathscr{D}^{G}$ *is the set of finite terms d produced inductively by the following grammar*

$$
\begin{aligned}
d &::= c^{g} \mid (d, d)^{g} \mid \{(S, \partial), \ldots, (S, \partial)\}^{g} \\
S &::= \{\iota, \ldots, \iota\} && S \text{ non empty} \\
\iota &::= d \mid \mathfrak{V} \\
\partial &::= d \mid \Omega \mid \mathsf{blame}\ p \\
g &::= ! \mid ? \\
p &::= \ell \mid \overline{\ell}
\end{aligned}
$$

where c ranges over the set $\mathscr{C}$ of constants, ℓ ranges over the set $\mathscr{L}$ of blame labels, and where Ω and $\mathfrak{V}$ are such that $\Omega \notin \mathscr{D}^{G}$ and $\mathfrak{V} \notin \mathscr{D}^{G}$.
We also write $\mathscr{D}^{G}_{\Omega} = \mathscr{D}^{G} \cup \{\Omega\} \cup \mathbb{B}\mathsf{lame}$ and $\mathscr{I}^{G} = \mathscr{P}_{f}(\mathscr{D}^{G} \cup \{\mathfrak{V}\}) \setminus \{\emptyset\}$.
We will also commonly use R to range over the set $\mathscr{P}_{f}(\mathscr{I}^{G} \times \mathscr{D}^{G}_{\Omega})$ so that R^{g} ranges over the finite relations in $\mathscr{D}^{G}$ with tag g.
We may also use $\hat{\partial}$ to range over both ι and ∂, that is, over elements of $\mathscr{D}^{G}_{\Omega} \cup \{\mathfrak{V}\}$.

The superscript g that appears in the elements of $\mathscr{D}^{\mathrm{G}}$ is the *gradual tag* of the element. We define $\mathrm{tag}(.) : \mathscr{D}^{\mathrm{G}} \to \{!, ?\}$ the function that returns the topmost tag of a denotation. We also use the involutory operation $\bar{\cdot} : \{!, ?\} \to \{!, ?\}$ which reverses a tag: $\overline{!} = ?$ and $\overline{?} = !$. For $g \in \{!, ?\}$, we also define the two operations $.^{g} : \mathscr{D}^{\mathrm{G}}_{\Omega} \cup \{\mathfrak{V}\} \to \mathscr{D}^{\mathrm{G}}_{\Omega} \cup \{\mathfrak{V}\}$ which modify the topmost tag of a denotation to g (and is the identity on Ω, $\mathfrak{V}$, and blame p). A value d such that $\mathrm{tag}(d) = !$ is said to be *static*, and a value d such that $\mathrm{tag}(d) = ?$ is said to be *dynamic*.

Notice how blame denotations are separated from the rest and are not included in $\mathscr{D}^{\mathrm{G}}$ but in $\mathscr{D}^{\mathrm{G}}_{\Omega}$. Much like Ω, blame is a form of error, and must be propagated as such. For example, when evaluating a pair $(\mathbf{e}_1, \mathbf{e}_2)$, if $\mathbf{e}_1$ produces a blame, then the whole pair must evaluate to a blame instead of to a pair of the form $(\mathsf{blame}\ p, 3)$.

We also draw inspiration from Chapter 11 and introduce a form of dummy input $\mathfrak{V}$, although it behaves differently than in Chapter 11. Contrary to the inputs $\mathfrak{V}_d$ of Chapter 11, this input $\mathfrak{V}$ is not meant to be used in the denotational semantics of our language. Its role, which we will explain in more details in the next sections, is to slightly change the behaviour of semantic subtyping on static types to avoid unsound subsumptions. In particular, the introduction of $\mathfrak{V}$ will make the interpretation of types such as $\mathbb{0} \to \mathbb{1}$ and $\mathbb{0} \to \mathsf{Int}$ different, although in semantic subtyping as presented in Chapter 2, all types $\mathbb{0} \to t$ are equivalent. There is also no need to distinguish $\mathfrak{V}$ inputs from the other inputs of a relation: an input set can mix elements of $\mathscr{D}^{\mathrm{G}}$ with $\mathfrak{V}$. Doing so greatly simplifies the formalism as $\mathfrak{V}$ can simply be treated as any other element in the interpretation of arrow types, as we will see in Definition 12.5.

12.1.2. A set-theoretic interpretation of gradual types

Using this new domain $\mathscr{D}^{\mathrm{G}}$, we can now give a set-theoretic interpretation of gradual types. Overall, our interpretation follows the same strategy as the interpretation presented in Chapter 10, in Definition 10.2. We just need to account for blame and tags.

The interpretation of blame is straightforward: any program of any type can produce a blame as a result. In other words, every denotation of the form blame p is in every type, including $\mathbb{0}$. Note that this is simply an artifact of the definition: the interpretation of $\mathbb{0}$ is still empty, and no value can be given type $\mathbb{0}$. However, a function that has type $\mathbb{1} \to \mathbb{0}$ may now return a blame.

Dealing with tags is less trivial. To understand the interpretation of tags, one can use the same intuition that guided us throughout Part I of this manuscript, namely that every occurrence of the dynamic type ? behaves as some form of existentially-quantified type variable. This leads us to distinguishing between values that are *always* of type τ, independently of how the occurrences of ? in τ are interpreted, and values that are *possibly* of type τ, that is, values that belong to at least one static type that can be obtained by replacing the occurrences of ? in τ. As an example, the value 3 is always of type $\mathsf{Int} \vee ?$, independently of how ? is interpreted: hence $3^{!}$ belongs to the interpretation of $\mathsf{Int} \vee ?$. On the contrary, `true` is of type $\mathsf{Int} \vee \mathsf{Bool}$ but not $\mathsf{Int} \vee \mathsf{Int}$ for example, hence $\texttt{true}^{?}$ belongs to the interpretation of $\mathsf{Int} \vee ?$ but $\texttt{true}^{!}$ does not.

We can connect this intuition with the notion of tags and the interpretation of types given in Chapter 10: a static element $d \in \mathscr{D}^{\mathrm{G}}$ belongs to the interpretation of a type τ if and only if the "untagged" counterpart of d in $\mathscr{D}^{\mathrm{F}}$ belongs to the interpretation of every static type t that can be obtained by replacing the occurrences of ? in τ with static types. Similarly, a dynamic element $d \in \mathscr{D}^{\mathrm{G}}$ belongs to the interpretation of τ if and only if its untagged counterpart belongs to at least one type t that can be obtained from τ.

Throughout the rest of this chapter, we will say that an element d *certainly* belongs to a type τ whenever $d^{!}$ belongs to τ, and that d *possibly* belongs to τ whenever $d^{?}$ belongs to τ. Notice that this informal terminology makes it clear that if a type contains a static element, it also contains its dynamic counterpart: if an element certainly belongs to a type, then it also possibly belongs to it. The converse is, obviously, not true.

This intuition gives us a clear interpretation of the dynamic type. Since ? can contain any value but can be used in any context and does not carry any static information, it is clear that it must contain every element with a dynamic tag, and only these elements. On the contrary, constant types are fully static: for example, the type Int denotes the set of all integers n^{g} independently of their tag, since Int cannot be made more precise and certainly contains all integers.

Remark 12.3.
Following the standard set-theoretic interpretation of intersection types, this interpretation validates our intuition that for every non-empty type τ, $\tau \wedge ?$ is not empty. Indeed, any non-empty type necessarily contains at least one dynamic element: we have already stated that if a type contains a static element, then it also contains its dynamic counterpart, therefore a type cannot contain only static elements. And since ? contains all dynamic elements, the intersection of a non-empty type τ with ? is never empty: it contains all the dynamic elements belonging to τ.

We can continue with this intuition to deduce the interpretation of product types. Product types are more complex in the sense that they can be partially dynamic, such as, for example, $? \times \mathsf{Int}$. Notice that no value is *always* of type $? \times \mathsf{Int}$: whatever pair (d_1, d_2) we consider, it is always possible to interpret ? as a type t that does not contain d_1 (the easiest one being $\mathbb{0}$) such

that the interpretation of $t \times \mathsf{Int}$ does not contain (d_1, d_2). This is because the only way for a pair to always be in a type is if both its components also are. In other words, a pair $(d_1, d_2)^!$ is in the interpretation of a type $\tau_1 \times \tau_2$ if and only if $d_1^!$ and $d_2^!$ are respectively in the interpretations of τ_1 and τ_2.

Remark 12.4.

When it comes to the dynamic tag ?, we could simply drop the condition on tags and say that $(d_1, d_2)^?$ is in the interpretation of a type $\tau_1 \times \tau_2$ if and only if d_1 and d_2 are respectively in the interpretations of τ_1 and τ_2.
However, asking for $d_i^?$ to belong to τ_i is the weakest possible condition about the tag of d_i: if d_i^g belongs to τ_i for some tag g, then it is also the case for $d_i^?$ (as formalized later on in Proposition 12.6). Therefore, it is equivalent and much simpler to say that $(d_1, d_2)^g$ belongs to $\tau_1 \times \tau_2$ if and only if d_i^g belongs to τ_i for $i \in \{1, 2\}$, as in Definition 12.5.

The interpretation of tags gets more complicated when considering arrow and negation types, due to contravariance. When does a relation R always belong to a type $\tau_1 \to \tau_2$? Following the above intuition, this is the case when for every type t_1 and t_2 obtained from τ_1 and τ_2, R belongs to the interpretation of $t_1 \to t_2$. According to the interpretation presented in Definition 10.2, this holds when R maps every input that belongs to t_1 to an output that belongs to t_2. In other words, by contrapositive, it must not be possible to find two types t_1 and t_2 more precise than τ_1 and τ_2 such that R maps an input of type t_1 to an output that is not in the interpretation of t_2. Using our previous terminology, for every input of R that *possibly belongs* to τ_1, the corresponding output *must always belong* to τ_2, otherwise it is possible to find two types t_1 and t_2 that violate the above condition.

Formally, for a relation $\{(S_1, \partial_1), \ldots, (S_n, \partial_n)\}^!$ to belong to a type $\tau_1 \to \tau_2$, it must then hold that for every pair (S_i, ∂_i), if there exists an element $d \in S_i$ that *possibly* belongs to τ_1 —i.e., $d^?$ belongs to τ_1— then $\partial_i^!$ must belong to τ_1.

The same reasoning can be followed to deduce when a relation R possibly belongs to a type $\tau_1 \to \tau_2$. Given a pair of R, there are two possibilities that ensure that the condition of Definition 10.2 holds for some type $t_1 \to t_2$: either the input does not belong to t_1, in which case the implication is vacuously true, or the output belongs to t_2. In other words, R possibly belongs to $\tau_1 \to \tau_2$ if for every pair of R, either the input does not always belong to τ_1, or the output possibly belongs to τ_2. Equivalently, whenever an input always belongs to τ_1, the output possibly belongs to τ_2.

Finally, all that remains to explain is the interpretation of negation types, which is simpler. Intuitively, a value possibly belongs to a type whenever it does not always belong to its negation. Symmetrically, to preserve the equality of the interpretations of $\neg\neg\tau$ and τ, if a value always belongs to the negation of a type, then it cannot possibly belong to this type. Formally, this gives the following formula: $(d : \neg\tau)^{\mathrm{G}} \iff \mathrm{tag}(d) = g$ and not $(d^{\overline{g}} : \tau)^{\mathrm{G}}$.

Putting the previous explanation together, we obtain the following interpretation of gradual types in $\mathscr{D}^{\mathrm{G}}$:

Definition 12.5 (Set-theoretic interpretation of types in $\mathscr{D}^{\mathrm{G}}$)**.** *We define a binary predicate $(\hat{\partial} : \tau)^G$ ("the element $\hat{\partial}$ belongs to the type τ") where $\hat{\partial} \in \mathscr{D}_\Omega^G \cup \{\mathfrak{V}\}$ and $\tau \in \mathrm{GTypes}$, by*

induction on the pair $(\hat{\partial}, \tau)$ *ordered lexicographically. The predicate is defined as follows:*

$$
\begin{aligned}
(\text{blame } p : \tau)^G &= \mathit{true} \\
(d : ?)^G &= \mathsf{tag}(d) = ? \\
(c^g : b)^G &= c \in \mathbb{B}(b) \\
((d_1, d_2)^g : \tau_1 \times \tau_2)^G &= (d_1^g : \tau_1)^G \mathit{\ and\ } (d_2^g : \tau_2)^G \\
(\{(S_1, \partial_1), \ldots, (S_n, \partial_n)\}^g : \tau_1 \to \tau_2)^G &= \forall i \in \{1..n\}.\ \mathit{if}\ \exists \iota \in S_i.\ (\iota^{\overline{g}} : \tau_1)^G\ \mathit{then}\ (\partial_i^g : \tau_2)^G \\
(d : \tau_1 \vee \tau_2)^G &= (d : \tau_1)^G \mathit{\ or\ } (d : \tau_2)^G \\
(d : \neg\tau)^G &= \mathsf{tag}(d) = g \mathit{\ and\ not\ } (d^{\overline{g}} : \tau)^G \\
(\mho : \tau)^G &= \mathit{true} \\
(\partial : \tau)^G &= \mathit{false} \qquad \mathit{otherwise}
\end{aligned}
$$

We define the set-theoretic interpretation *of gradual types* $[\![.]\!]^G : \mathsf{GTypes} \to \mathscr{P}(\mathscr{D}^G)$ *as* $[\![\tau]\!]^G = \{d \in \mathscr{D}^G \mid (d : \tau)^G\}$.

In the end, the set-theoretic interpretation of gradual types is really similar to the interpretation of types presented in Chapter 10. The major difference is the propagation of tags, which amounts to reversing the tag on the element being checked according to the variance of the type at hand.

Note that the interpretation $[\![.]\!]^G$ is defined on $\mathscr{D}^G$. This means in particular that a blame element never belongs to the set-theoretic interpretation of a type, and that while $(\text{blame } p : \mathbb{0})^G$ holds, the set-theoretic interpretation of $\mathbb{0}$ is still empty.

From this interpretation, we can immediately deduce a few interesting properties regarding the behaviour of gradual set-theoretic types. Firstly, the interpretations of ? and $\neg$? are equal, and consist in all values tagged with ?. While this may seem surprising, this behaviour is expected: if the dynamic type ? is seen as *the absence* of type information, then $\neg$? provides no information either. Another way to see this is by thinking of ? as a type that stands for any type: if ? stands for any type τ, then it also stands for $\neg\tau$, and thus $\neg$? stands for $\neg\neg\tau$, which is equivalent to τ.

Secondly, the interpretations of $\mathbb{0}$, $\mathbb{1}$ and ? are very different: $\mathbb{0}$ denotes the empty set of values, $\mathbb{1}$ the set of all values (static *and* dynamic), while ? denotes the set containing all dynamic values. This is further emphasized by the fact that $\tau \wedge$? does *not*, generally, have the same interpretation as τ. For example, Int $\wedge$? denotes the set of all *boxed* integers, while Int denotes the set of all integers, boxed or not.

More formally, a crucial property of our interpretation (which we already hinted at previously), is that if a type contains an unboxed value (i.e., a value with tag !) then it contains the boxed version of this value. The converse only holds for static types: if a static type contains a boxed value, then unboxing it produces a value that is still in this type. This yields the following proposition:

Proposition 12.6. *For all type* $\tau \in \mathsf{GTypes}$*, for all static type* $t \in \mathsf{Types}$*, for all* $d \in \mathscr{D}^G$*,*

$$
\begin{aligned}
d^! \in [\![\tau]\!]^G &\implies d^? \in [\![\tau]\!]^G \\
d^! \in [\![t]\!]^G &\iff d^? \in [\![t]\!]^G
\end{aligned}
$$

Proof. We prove the following two stronger results for every $\hat{\partial} \in \mathscr{D}^{\mathrm{G}}_{\Omega} \cup \{\mho\}$:

$$(\hat{\partial}^{!} : \tau)^{\mathrm{G}} \implies (\hat{\partial}^{?} : \tau)^{\mathrm{G}}$$
$$(\hat{\partial}^{!} : t)^{\mathrm{G}} \iff (\hat{\partial}^{?} : t)^{\mathrm{G}}$$

Both results are proven by induction on the pair $(\hat{\partial}, \tau)$ or $(\hat{\partial}, t)$ lexicographically ordered, following Definition 12.5. Notice that if $\hat{\partial} \notin \mathscr{D}^{\mathrm{G}}$, the result is immediate since the operation $.^{g}$ is the identity on blame p, Ω and $\mho$. Hence we only treat the cases where $\hat{\partial} = d \in \mathscr{D}^{\mathrm{G}}$.

1. $\underline{(\hat{\partial}^{!} : \tau)^{\mathrm{G}} \implies (\hat{\partial}^{?} : \tau)^{\mathrm{G}}}$.
   - $(d^{!} : ?)^{\mathrm{G}}$. Vacuously true.
   - $(c^{!} : b)^{\mathrm{G}}$. By Definition 12.5, this implies $c \in \mathbb{B}(b)$ and thus $(c^{?} : b)^{\mathrm{G}}$.
   - $((d_1, d_2)^{!} : \tau_1 \times \tau_2)^{\mathrm{G}}$ By Definition 12.5, $(d_1^{!} : \tau_1)^{\mathrm{G}}$ and $(d_2^{!} : \tau_2)^{\mathrm{G}}$. By induction hypothesis, this implies that $(d_1^{?} : \tau_1)^{\mathrm{G}}$ and $(d_2^{?} : \tau_2)^{\mathrm{G}}$. By Definition 12.5, this yields $((d_1, d_2)^{?} : \tau_1 \times \tau_2)^{\mathrm{G}}$.
   - $(\{(S_1, \partial_1), \ldots, (S_n, \partial_n)\}^{!} : \tau_1 \to \tau_2)^{\mathrm{G}}$. Let $i \in \{1..n\}$ such that $\exists \iota \in S_i$. $(\iota^{!} : \tau_1)^{\mathrm{G}}$. By induction hypothesis, $(\iota^{?} : \tau_1)^{\mathrm{G}}$. By hypothesis and Definition 12.5, this ensures that $(\partial_i^{!} : \tau_2)^{\mathrm{G}}$. By induction hypothesis, $(\partial_i^{?} : \tau_2)^{\mathrm{G}}$, hence the result.
   - $(d^{!} : \tau_1 \vee \tau_2)^{\mathrm{G}}$. By Definition 12.5, $(d^{!} : \tau_1)^{\mathrm{G}}$ or $(d^{!} : \tau_2)^{\mathrm{G}}$. By induction hypothesis, $(d^{?} : \tau_1)^{\mathrm{G}}$ or $(d^{?} : \tau_2)^{\mathrm{G}}$. Thus, by Definition 12.5, $(d^{?} : \tau_1 \vee \tau_2)^{\mathrm{G}}$.
   - $(d^{!} : \neg\tau)^{\mathrm{G}}$. By Definition 12.5, it does not hold that $(d^{?} : \tau)^{\mathrm{G}}$. Therefore, by induction hypothesis, necessarily $(d^{!} : \tau)^{\mathrm{G}}$ does not hold. Thus, $(d^{?} : \neg\tau)^{\mathrm{G}}$.
2. $\underline{(\hat{\partial}^{!} : t)^{\mathrm{G}} \impliedby (\hat{\partial}^{?} : t)^{\mathrm{G}}}$.
   - $(d^{?} : ?)^{\mathrm{G}}$. Vacuously true since $? \notin$ Types.
   - $(c^{?} : b)^{\mathrm{G}}$. By Definition 12.5, this implies $c \in \mathbb{B}(b)$ and thus $(c^{!} : b)^{\mathrm{G}}$.
   - $((d_1, d_2)^{?} : t_1 \times t_2)^{\mathrm{G}}$ By Definition 12.5, $(d_1^{?} : t_1)^{\mathrm{G}}$ and $(d_2^{?} : t_2)^{\mathrm{G}}$. By induction hypothesis, this implies that $(d_1^{!} : t_1)^{\mathrm{G}}$ and $(d_2^{!} : t_2)^{\mathrm{G}}$. By Definition 12.5, this yields $((d_1, d_2)^{!} : t_1 \times t_2)^{\mathrm{G}}$.
   - $(\{(S_1, \partial_1), \ldots, (S_n, \partial_n)\}^{?} : t_1 \to t_2)^{\mathrm{G}}$. Let $i \in \{1..n\}$ such that $\exists \iota \in S_i$. $(\iota^{?} : t_1)^{\mathrm{G}}$. By induction hypothesis, $(\iota^{!} : t_1)^{\mathrm{G}}$. By hypothesis and Definition 12.5, this ensures that $(\partial_i^{?} : t_2)^{\mathrm{G}}$. By induction hypothesis, $(\partial_i^{!} : t_2)^{\mathrm{G}}$, hence the result.
   - $(d^{?} : t_1 \vee t_2)^{\mathrm{G}}$. By Definition 12.5, $(d^{?} : t_1)^{\mathrm{G}}$ or $(d^{?} : t_2)^{\mathrm{G}}$. By induction hypothesis, $(d^{!} : t_1)^{\mathrm{G}}$ or $(d^{!} : t_2)^{\mathrm{G}}$. Thus, by Definition 12.5, $(d^{!} : t_1 \vee t_2)^{\mathrm{G}}$.
   - $(d^{?} : \neg t)^{\mathrm{G}}$. By Definition 12.5, it does not hold that $(d^{!} : t)^{\mathrm{G}}$. Therefore, by induction hypothesis, necessarily $(d^{?} : t)^{\mathrm{G}}$ does not hold. Thus, $(d^{!} : \neg t)^{\mathrm{G}}$.

□

While we have already shown that, in general, $\tau \wedge \neg\tau$ is not empty (and thus, it is not true anymore that a value always belongs to a type or its negation), our interpretation still satisfies the De Morgan's laws. This is a crucial property since it ensures that, for example, a value belongs to the intersection of two types if and only if it belongs to both types.

Proposition 12.7 (De Morgan's laws)**.** *For all types* $\tau, \tau_1, \tau_2 \in \mathrm{GTypes}$,

$$[\![\neg\neg\tau]\!]^G = [\![\tau]\!]^G$$
$$[\![\neg(\tau_1 \vee \tau_2)]\!]^G = [\![\neg\tau_1 \wedge \neg\tau_2]\!]^G$$
$$[\![\neg\mathbb{0}]\!]^G = \mathscr{D}^G$$

Proof. We prove the three equalities separately.

1. $(d : \neg\neg\tau)^{\mathrm{G}} \iff \text{not } (d^{\overline{\mathrm{tag}(d)}} : \neg\tau)^{\mathrm{G}} \iff \text{not not } (d^{\mathrm{tag}(d)} : \tau)^{\mathrm{G}} \iff (d : \tau)^{\mathrm{G}}$
2. $$\begin{aligned}(d : \neg(\tau_1 \vee \tau_2))^{\mathrm{G}} &\iff \text{not } (d^{\overline{\mathrm{tag}(d)}} : \tau_1 \vee \tau_2)^{\mathrm{G}} \\ &\iff \text{not } ((d^{\overline{\mathrm{tag}(d)}} : \tau_1)^{\mathrm{G}} \text{ or } (d^{\overline{\mathrm{tag}(d)}} : \tau_2)^{\mathrm{G}}) \\ &\iff (\text{not } (d^{\overline{\mathrm{tag}(d)}} : \tau_1)^{\mathrm{G}}) \text{ and } (\text{not } (d^{\overline{\mathrm{tag}(d)}} : \tau_2)^{\mathrm{G}}) \\ &\iff (d : \neg\tau_1 \wedge \neg\tau_2))^{\mathrm{G}}\end{aligned}$$
3. $(d : \neg\mathbb{0})^{\mathrm{G}} \iff \text{not } (d^{\overline{\mathrm{tag}(d)}} : \mathbb{0})^{\mathrm{G}} \iff \text{true}$

□

12.1.3. Subtyping and precision

Having defined the interpretation of gradual types and stated some of its most important properties, we can now proceed with the study of two relations induced by this interpretation.

The first relation is subtyping, for which we use the definition of semantic subtyping presented in Chapter 2: subtyping of two gradual types is just defined as the set-containment of their interpretations:

Definition 12.8 (Subtyping on $\mathscr{D}^{\mathrm{G}}$)**.** *We define the* subtyping *relation* $\leq$ *and the* subtyping equivalence *relation* $\simeq$ *on* $\mathscr{D}^G$ *as* $\tau_1 \leq \tau_2 \overset{\text{def}}{\iff} [\![\tau_1]\!]^G \subseteq [\![\tau_2]\!]^G$ *and* $\tau_1 \simeq \tau_2 \overset{\text{def}}{\iff} (\tau_1 \leq \tau_2)$ and $(\tau_2 \leq \tau_1)$.

The role of subtyping is to dictate whether it is *statically sound* to pass a value to a context, knowing the type of the value and the type expected by the context. This is, in essence, the reason we use set-containment to define subtyping: if a context can accept values from a set S, then it can accept values from any smaller set $S' \subseteq S$.

However, the whole point of gradual typing is to allow some statically unsound operations to take place, provided we insert the dynamic checks necessary to ensure that this does not result in a stuck computation. In other words, while subtyping facilitates the movement of values inside the static world or inside the dynamic world, it does not allow values to cross the boundary between the two, which is the role of gradual typing.

We therefore define a new relation, *precision*, which formalizes the ability to cross between the static world and the dynamic world. Formally, we say that τ_2 *is more precise than* τ_1, noted $\tau_1 \preccurlyeq \tau_2$, if τ_1 is "more dynamic" than τ_2. The idea being that values that go from type τ_1 to type τ_2 will cross (even partially) from the dynamic world into the static world: they may be unboxed, but will never become boxed. In other words, τ_2 is more precise than τ_1 if all the static values of

τ_1 are preserved and are still static values of τ_2, and if the dynamic values of τ_2 are also dynamic values of τ_1. Formally, this yields the following definition:

Definition 12.9 (Precision). *We define the* precision *relation* $\preccurlyeq$ *on* $\mathscr{D}^G$ *as*

$$\tau_1 \preccurlyeq \tau_2 \overset{\text{def}}{\iff} \forall d \in \mathscr{D}^G, \begin{cases} d^? \in [\![\tau_2]\!]^G & \implies d^? \in [\![\tau_1]\!]^G \\ d^! \in [\![\tau_1]\!]^G & \implies d^! \in [\![\tau_2]\!]^G \end{cases}$$

As in the first part of this manuscript, we may also say that τ_1 *materializes into* τ_2 whenever $\tau_1 \preccurlyeq \tau_2$. We may also refer to τ_2 as a *materialization* of τ_1.

The above definition strongly resembles some form of set-containment. Therefore, we can wonder if precision can be expressed using set-containment, or even subtyping. And it is indeed possible: the first condition in Definition 12.9 amounts to checking whether the dynamic values of τ_2 are also dynamic values of τ_1. Recall that the dynamic values of a type can be obtained by taking its intersection with $?$. Therefore, this condition is equivalent to checking whether $\tau_2 \wedge ? \leq \tau_1 \wedge ?$.

The second condition is trickier. It amounts to checking whether the static values of τ_1 are also static values of τ_2. But since $\neg ? \simeq ?$, we have no way of selecting *only* the static values of a type using connectives. What we can do, however, is remove dynamic values from the equation by adding the set of all dynamic values to both types: if the set containing all static values of τ_1 and all dynamic values is contained in the set containing all static values of τ_2 and all dynamic values, then our second condition holds. And this set can be obtained easily by simply taking the union of both types with $?$.

Putting all of this together, we obtain the following syntactic characterization of precision:

Proposition 12.10 (Syntactic characterization of precision). *For every types* $\tau_1, \tau_2 \in \mathrm{GTypes}$,

$$\tau_1 \preccurlyeq \tau_2 \iff \begin{cases} \tau_2 \wedge ? & \leq \tau_1 \wedge ? \\ \tau_1 \vee ? & \leq \tau_2 \vee ? \end{cases}$$

Proof. We proceed by double implication.

- $\underline{\tau_1 \preccurlyeq \tau_2.}$ We prove the two statements.
  1. Let $d \in [\![\tau_2 \wedge ?]\!]^{\mathrm{G}}$. By Definition 12.5 and Proposition 12.7, $d \in [\![\tau_2]\!]^{\mathrm{G}} \cap [\![?]\!]^{\mathrm{G}}$. By Definition 12.5, this yields $\mathrm{tag}(d) = ?$, thus $d^? = d \in [\![\tau_2]\!]^{\mathrm{G}}$. By Definition 12.9, we deduce that $d \in [\![\tau_1]\!]^{\mathrm{G}}$. Since $\mathrm{tag}(d) = ?$, we deduce that $d \in [\![\tau_1 \wedge ?]\!]^{\mathrm{G}}$.
  2. Let $d \in [\![\tau_1 \vee ?]\!]^{\mathrm{G}}$. If $\mathrm{tag}(d) = ?$ then immediately $d \in [\![?]\!]^{\mathrm{G}}$ and thus $d \in [\![\tau_2 \vee ?]\!]^{\mathrm{G}}$ by Definition 12.5. Otherwise, if $\mathrm{tag}(d) = !$ then necessarily $d \in [\![\tau_1]\!]^{\mathrm{G}}$. And since $d = d^!$, by Definition 12.9, we deduce that $d \in [\![\tau_2]\!]^{\mathrm{G}}$ and the result follows.
- $\underline{\tau_2 \wedge ? \leq \tau_1 \wedge ? \text{ and } \tau_1 \vee ? \leq \tau_2 \vee ?.}$ We prove the two statements of Definition 12.9.
  1. Let $d^? \in [\![\tau_2]\!]^{\mathrm{G}}$. By Definition 12.5, $d^? \in [\![?]\!]^{\mathrm{G}}$. Thus, by Proposition 12.7, $d^? \in [\![? \wedge \tau_2]\!]^{\mathrm{G}}$. By hypothesis, this yields $d^? \in [\![? \wedge \tau_1]\!]^{\mathrm{G}}$, and Proposition 12.7 then yields $d^? \in [\![\tau_1]\!]^{\mathrm{G}}$.

2. Let $d^! \in [\![\tau_1]\!]^G$. By Definition 12.5, $d^! \in [\![\tau_1 \vee ?]\!]^G$. Thus, by hypothesis, $d^! \in [\![\tau_2 \vee ?]\!]^G$. By Definition 12.5, $d^! \notin [\![?]\!]^G$ thus necessarily $d^! \in [\![\tau_2]\!]^G$.

□

We have given set-theoretic definitions of subtyping and precision on gradual types, however, a problem remains: the decidability of these relations. Subtyping, on non-gradual set-theoretic types, reduces to an emptiness problem: t_1 is a subtype of t_2 if and only if $t_1 \wedge \neg t_2$ is empty. And since the emptiness of a set-theoretic type is decidable, so is subtyping. However, as we have already shown, this reduction is not feasible in our interpretation of gradual types: ? is a subtype of itself, but $? \wedge \neg ?$ is not empty since it is equivalent to ?.

There is, however, an additional property of gradual types induced by our interpretation that can help us: the set of the materializations of a gradual type always contains two extremal *static* types. That is, every gradual type τ materializes in two static types that are, respectively, a subtype and a supertype of every other materialization of τ. The extremal materializations of a gradual types are defined informally as follows:

Definition 12.11 (Gradual extrema)**.** *For every gradual type $\tau \in$ GTypes, we define the minimal (resp. maximal) materialization of τ, noted $\tau^{\Downarrow}$ (resp. $\tau^{\Uparrow}$), as the static type obtained by replacing every covariant (resp. contravariant) occurrence of ? in τ by $\mathbb{0}$ and every contravariant (resp. covariant) occurrence of ? in τ by $\mathbb{1}$.*

For a more formal definition of the gradual extrema which properly formalizes the substitution of the occurrences of ? and ensures that extrema are well-defined for recursive types, we redirect the reader to Chapter 5 and Definition 5.24 in particular.

It is straightforward to verify that we have the following equalities:

$$\begin{array}{rclcrcl}
(\tau_1 \to \tau_2)^{\Downarrow} &=& \tau_1^{\Uparrow} \to \tau_2^{\Downarrow} & \qquad & (\tau_1 \to \tau_2)^{\Uparrow} &=& \tau_1^{\Downarrow} \to \tau_2^{\Uparrow} \\
(\neg\tau)^{\Downarrow} &=& \neg(\tau^{\Uparrow}) & & (\neg\tau)^{\Uparrow} &=& \neg(\tau^{\Downarrow}) \\
(\tau_1 \vee \tau_2)^{\Downarrow} &=& \tau_1^{\Downarrow} \vee \tau_2^{\Downarrow} & & (\tau_1 \vee \tau_2)^{\Uparrow} &=& \tau_1^{\Uparrow} \vee \tau_2^{\Uparrow} \\
(\tau_1 \times \tau_2)^{\Downarrow} &=& \tau_1^{\Downarrow} \times \tau_2^{\Downarrow} & & (\tau_1 \times \tau_2)^{\Uparrow} &=& \tau_1^{\Uparrow} \times \tau_2^{\Uparrow}
\end{array}$$

Gradual extrema can be understood as follows: according to Definition 12.9, materializing a type can remove dynamic values from its interpretation (possibly converting them to static values), but can never remove static values. Therefore, the finest materialization of a type τ can be obtained by removing all its dynamic values (i.e., all values tagged with ?). This produces a type containing only the values d such that $d^!$ belongs to τ: this is the minimal materialization $\tau^{\Downarrow}$ of τ.

The maximal materialization $\tau^{\Uparrow}$ is obtained similarly, except this time, we simply convert all the dynamic values of τ into static values: we obtain a type containing all the values d such that $d^?$ belongs to τ. This is formalized by the following proposition:

Proposition 12.12 (Denotational interpretation of extrema)**.** *For every gradual type $\tau \in$ GTypes,*

$$[\![\tau^{\Downarrow}]\!]^G = \{d \in \mathscr{D}^G \mid d^! \in [\![\tau]\!]^G\}$$
$$[\![\tau^{\Uparrow}]\!]^G = \{d \in \mathscr{D}^G \mid d^? \in [\![\tau]\!]^G\}$$

Proof. We prove the equivalent result that for every $d \in \mathscr{D}^{\mathrm{G}}$, we have the following equivalences:

$$(d : \tau^{\Downarrow})^{\mathrm{G}} \iff (d^{!} : \tau)^{\mathrm{G}}$$
$$(d : \tau^{\Uparrow})^{\mathrm{G}} \iff (d^{?} : \tau)^{\mathrm{G}}$$

This allows us to reason by induction on the pair (d, τ) lexicographically ordered. Due to the variance involved in the definition of the extrema, the two equivalences must be proven in a single induction. However, since the two are proven almost identically, we only explicitly prove the first one.

- $\underline{(d, \neg\tau).}$ We have $(\neg\tau)^{\Downarrow} = \neg(\tau^{\Uparrow})$. Thus, by Definition 12.5, if we write $g = \mathrm{tag}(d)$, we have that $(d : (\neg\tau)^{\Downarrow})^{\mathrm{G}} \iff \neg(d^{\overline{g}} : \tau^{\Uparrow})^{\mathrm{G}}$. By IH, using the second equivalence, we have that $\neg(d^{\overline{g}} : \tau^{\Uparrow})^{\mathrm{G}} \iff \neg(d^{?} : \tau)^{\mathrm{G}}$. By Definition 12.5, we have $\neg(d^{?} : \tau)^{\mathrm{G}} \iff (d^{!} : \neg\tau)^{\mathrm{G}}$, hence the result.

- $\underline{(d, \tau_1 \vee \tau_2).}$ We have $(\tau_1 \vee \tau_2)^{\Downarrow} = \tau_1{}^{\Downarrow} \vee \tau_2{}^{\Downarrow}$. By Definition 12.5, we have $(d : (\tau_1 \vee \tau_2)^{\Downarrow})^{\mathrm{G}} \iff (d : \tau_1{}^{\Downarrow})^{\mathrm{G}} \vee (d : \tau_2{}^{\Downarrow})^{\mathrm{G}}$. By IH, we deduce that $(d : (\tau_1 \vee \tau_2)^{\Downarrow})^{\mathrm{G}} \iff (d^{!} : \tau_1)^{\mathrm{G}} \vee (d^{!} : \tau_2)^{\mathrm{G}}$, and the result follows by Definition 12.5.

- $\underline{(d, ?).}$ We have $?^{\Downarrow} = \mathbb{0}$, hence $(d : ?^{\Downarrow})^{\mathrm{G}}$ cannot hold. By Definition 12.5, we also have that $(d : ?)^{\mathrm{G}} \iff \mathrm{tag}(d) = ?$, which ensures that $(d^{!} : ?)^{\mathrm{G}}$ does not hold either.

- $\underline{((d_1, d_2)^g, \tau_1 \times \tau_2).}$ We have $(\tau_1 \times \tau_2)^{\Downarrow} = \tau_1{}^{\Downarrow} \times \tau_2{}^{\Downarrow}$. By Definition 12.5, we have $((d_1, d_2)^g : (\tau_1 \times \tau_2)^{\Downarrow})^{\mathrm{G}} \iff (d_1^g : \tau_1{}^{\Downarrow})^{\mathrm{G}} \wedge (d_2^g : \tau_2{}^{\Downarrow})^{\mathrm{G}}$. By IH, we deduce that $((d_1, d_2)^g : (\tau_1 \times \tau_2)^{\Downarrow})^{\mathrm{G}} \iff (d_1^{!} : \tau_1)^{\mathrm{G}} \wedge (d_2^{!} : \tau_2)^{\mathrm{G}}$. By Definition 12.5, we have $(d_1^{!} : \tau_1)^{\mathrm{G}} \wedge (d_2^{!} : \tau_2)^{\mathrm{G}} \iff ((d_1, d_2)^{!} : \tau_1 \times \tau_2)^{\mathrm{G}}$, hence the result.

- $\underline{(\{(S_i, \partial_i) \mid i \in I\}^g, \tau_1 \to \tau_2).}$ We have $(\tau_1 \to \tau_2)^{\Downarrow} = \tau_1{}^{\Uparrow} \to \tau_2{}^{\Downarrow}$. By Definition 12.5, we have $(\{(S_i, \partial_i) \mid i \in I\}^g : (\tau_1 \to \tau_2)^{\Downarrow})^{\mathrm{G}} \iff (\forall i \in I.\ (\exists \iota \in S_i.\ (\iota^{\overline{g}} : \tau_1{}^{\Uparrow})^{\mathrm{G}}) \implies (\partial_i^g : \tau_2{}^{\Downarrow})^{\mathrm{G}})$. By IH, we have $(\iota^{\overline{g}} : \tau_1{}^{\Uparrow})^{\mathrm{G}} \iff (\iota^{?} : \tau_1)^{\mathrm{G}}$ and $(\partial_i^g : \tau_2{}^{\Downarrow})^{\mathrm{G}} \iff (\partial_i^{!} : \tau_2)^{\mathrm{G}}$. Therefore, the above condition is equivalent to $\forall i \in I.\ (\exists \iota \in S_i.\ (\iota^{?} : \tau_1)^{\mathrm{G}}) \implies (\partial_i^{!} : \tau_2)^{\mathrm{G}}$. By Definition 12.5, this is equivalent to $(\{(S_i, \partial_i) \mid i \in I\}^{!} : \tau_1 \to \tau_2)^{\mathrm{G}}$, hence the result.

- All other cases vacuously hold since neither $(d : \tau^{\Downarrow})^{\mathrm{G}}$ nor $(d^{!} : \tau)^{\mathrm{G}}$ can hold.

□

This last property allows us to prove the fundamental property of gradual extrema as stated in Chapter 5 (Theorem 5.25), which formalizes our previous explanation: the types $\tau^{\Downarrow}$ and $\tau^{\Uparrow}$ are materializations of τ that are respectively a subtype and a supertype of every other materialization of τ. Thanks to our set-theoretic interpretation of gradual types and Proposition 12.12, the proof of this theorem is much simpler than in Chapter 5 and does not rely on the manipulation of type substitutions.

Theorem 12.13 (Fundamental property of gradual extrema)**.** *For every gradual type* $\tau \in \mathrm{GTypes}$*, the following holds:*

- $\tau \preccurlyeq \tau^{\Downarrow}$ *and* $\tau \preccurlyeq \tau^{\Uparrow}$*;*
- *for every* $\tau' \in \mathsf{GTypes}$ *such that* $\tau \preccurlyeq \tau'$, $\tau^{\Downarrow} \leq \tau' \leq \tau^{\Uparrow}$.

Proof. We prove the two properties independently.

- Let $d \in \mathscr{D}^{\mathrm{G}}$ such that $d^{?} \in [\![\tau^{\Downarrow}]\!]^{\mathrm{G}}$. By Proposition 12.12, we have $d^{!} \in [\![\tau]\!]^{\mathrm{G}}$. By Proposition 12.6, we deduce that $d^{?} \in [\![\tau]\!]^{\mathrm{G}}$.
  Now let $d \in \mathscr{D}^{\mathrm{G}}$ such that $d^{!} \in [\![\tau]\!]^{\mathrm{G}}$. By Proposition 12.12, we have $d^{g} \in [\![\tau^{\Downarrow}]\!]^{\mathrm{G}}$ independently of g, thus in particular $d^{!} \in [\![\tau^{\Downarrow}]\!]^{\mathrm{G}}$. Hence, by Definition 12.9, we deduce that $\tau \preccurlyeq \tau^{\Downarrow}$.
  The same reasoning can be followed to deduce that $\tau \preccurlyeq \tau^{\Uparrow}$.
- Let $\tau' \in \mathsf{GTypes}$ such that $\tau \preccurlyeq \tau'$. Let $d \in [\![\tau^{\Downarrow}]\!]^{\mathrm{G}}$. By Proposition 12.12, we have $d^{!} \in [\![\tau]\!]^{\mathrm{G}}$. By Definition 12.9, this yields $d^{!} \in [\![\tau']\!]^{\mathrm{G}}$. Hence, by Proposition 12.6, necessarily $d \in [\![\tau']\!]^{\mathrm{G}}$, which proves that $\tau^{\Downarrow} \leq \tau'$. The same reasoning proves that $\tau' \leq \tau^{\Uparrow}$.

□

We can now start to see how gradual extrema can help us deciding subtyping and precision. We have shown in Proposition 12.10 that precision reduces to gradual subtyping, by comparing separately the static and the dynamic parts of types. We can of course follow the same strategy for subtyping: a type τ_1 is a subtype of τ_2 if and only if the static (resp. dynamic) values of τ_1 are also static (resp. dynamic) values of τ_2.

This is where gradual extrema come into play. As Proposition 12.12 shows, $\tau^{\Downarrow}$ and $\tau^{\Uparrow}$ represent *exactly* the static part and the dynamic part of τ, respectively. Thus, we can compute both subtyping and precision by simply comparing extremal materializations. This allows us to deduce the following theorem, which forms the basis of the work we presented in Chapter 6:

Theorem 12.14 (Decidability of subtyping and precision)**.** *For every types* $\tau_1, \tau_2 \in \mathsf{GTypes}$*,*

$$\tau_1 \leq \tau_2 \iff {\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow} \text{ and } {\tau_1}^{\Uparrow} \leq {\tau_2}^{\Uparrow}$$
$$\tau_1 \preccurlyeq \tau_2 \iff {\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow} \text{ and } {\tau_2}^{\Uparrow} \leq {\tau_1}^{\Uparrow}$$

Proof. We first prove the first equivalence.

- Suppose that $\tau_1 \leq \tau_2$ and let $d \in [\![{\tau_1}^{\Downarrow}]\!]^{\mathrm{G}}$. By Proposition 12.12, we have $d^{!} \in [\![\tau_1]\!]^{\mathrm{G}}$. By hypothesis, this yields $d^{!} \in [\![\tau_2]\!]^{\mathrm{G}}$. By Proposition 12.12, we deduce $d \in [\![{\tau_2}^{\Downarrow}]\!]^{\mathrm{G}}$, hence ${\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow}$. Now let $d \in [\![{\tau_1}^{\Uparrow}]\!]^{\mathrm{G}}$. By Proposition 12.12, we have $d^{?} \in [\![\tau_1]\!]^{\mathrm{G}}$. By hypothesis, this yields $d^{?} \in [\![\tau_2]\!]^{\mathrm{G}}$. By Proposition 12.12, we deduce $d \in [\![{\tau_2}^{\Uparrow}]\!]^{\mathrm{G}}$, hence ${\tau_1}^{\Uparrow} \leq {\tau_2}^{\Uparrow}$.
- Now suppose that ${\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow}$ and ${\tau_1}^{\Uparrow} \leq {\tau_2}^{\Uparrow}$, and let $d \in [\![\tau_1]\!]^{\mathrm{G}}$. Suppose that $\mathrm{tag}(d) = ?$. By Proposition 12.12, we have that $d \in [\![{\tau_1}^{\Uparrow}]\!]^{\mathrm{G}}$. By hypothesis, this yields $d \in [\![{\tau_2}^{\Uparrow}]\!]^{\mathrm{G}}$. Hence, by Proposition 12.12, $d^{?} \in [\![\tau_2]\!]^{\mathrm{G}}$, but since $d^{?} = d$ we deduce that $\tau_1 \leq \tau_2$. The same reasoning can be followed using ${\tau_1}^{\Downarrow}$ if $\mathrm{tag}(d) = !$, yielding the result.

Now, for the second equivalence, by Proposition 12.10, we have that $\tau_1 \preccurlyeq \tau_2 \iff \tau_1 \vee ? \leq$

$\tau_2 \vee ?$ and $\tau_2 \wedge ? \leq \tau_1 \wedge ?$. Moreover, it holds that $(\tau_1 \vee ?)^{\Downarrow} \simeq {\tau_1}^{\Downarrow}$, $(\tau_1 \vee ?)^{\Uparrow} \simeq \mathbb{1}$, $(\tau_1 \wedge ?)^{\Downarrow} \simeq \mathbb{0}$, and $(\tau_1 \wedge ?)^{\Uparrow} = {\tau_1}^{\Uparrow}$, and similarly for τ_2. Thus, using the first equivalence, we deduce that:

$$\tau_1 \preccurlyeq \tau_2 \iff {\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow} \text{ and } \mathbb{1} \leq \mathbb{1} \text{ and } \mathbb{0} \leq \mathbb{0} \text{ and } {\tau_2}^{\Uparrow} \leq {\tau_1}^{\Uparrow}$$

Since $\mathbb{0} \leq \mathbb{0}$ and $\mathbb{1} \leq \mathbb{1}$ trivially hold, we have the result. □

This theorem drives a very strong message. It states that to add gradual typing to a language, it is not necessary to define the subtyping and precision relations as we did in the previous subsections: all is needed is an initial subtyping relation on static types, together with the definition of gradual extrema. Provided the subtyping relation on static types is decidable, Theorem 12.14 also guarantees the decidability of both subtyping and precision on gradual types, by reducing them to subtyping on static types. And since for every static type t, $t^{\Uparrow} = t^{\Downarrow} = t$, defining gradual subtyping this way guarantees that it is a conservative extension of its static counterpart.

We finish this section by an even more drastic property than Theorem 12.14, which proves we do not even need the full syntax of gradual types:

Theorem 12.15 (Representation of gradual types)**.** *For every gradual type τ,*

$$\tau \simeq \tau^{\Downarrow} \vee (? \wedge \tau^{\Uparrow})$$

Proof. We have $(\tau^{\Downarrow} \vee (? \wedge \tau^{\Uparrow}))^{\Downarrow} = \tau^{\Downarrow} \vee (\mathbb{0} \wedge \tau^{\Uparrow}) \simeq \tau^{\Downarrow}$ and $(\tau^{\Downarrow} \vee (? \wedge \tau^{\Uparrow}))^{\Uparrow} = \tau^{\Downarrow} \vee (\mathbb{1} \wedge \tau^{\Uparrow}) \simeq \tau^{\Downarrow} \vee \tau^{\Uparrow} \simeq \tau^{\Uparrow}$ since $\tau^{\Downarrow} \leq \tau^{\Uparrow}$. Hence the result follows immediately from Theorem 12.14. □

According to this theorem, every gradual type τ is equivalent to the ? type as long as we bound it with the two extrema $\tau^{\Downarrow}$ and $\tau^{\Uparrow}$. Therefore, every gradual type can be represented by a pair of static types, and to add gradual typing to a system, it suffices to add a single constant ? to types that will always appear at top level, and never under an arrow or product.

12.1.4. About semantic subtyping

While the last few theorems provide very strong results about gradual types, there is, however, one caveat that must be taken into account. In semantic subtyping as defined in Chapter 2, arrow types do not always verify the usual variance properties. In particular, all types $\mathbb{0} \to t$ are equivalent, since they all contain all finite relations of the interpretation domain. This means that the variance properties of arrow types can also be broken when ? occurs contravariantly, since taking its minimal concretization makes the domain of the arrow empty.

Consider for example the type $? \to ?$. Its minimal concretization is $\mathbb{1} \to \mathbb{0}$. Its maximal concretization if $\mathbb{0} \to \mathbb{1}$, which, under the definition of semantic subtyping given in Chapter 2, is equivalent to $\mathbb{0} \to \mathbb{0}$. According to Theorem 12.14, this proves that $? \to ?$ is a subtype of $? \to \mathbb{0}$. This is clearly unsound, as it allows to deduce type $\mathbb{0}$ for the application of a function of type $? \to ?$ to a value of type ?, even though such an application *can* return a result.

This is the reason we introduced the new element $\mathfrak{d}$, which changes the subtyping relation on static types. In our new interpretation, the interpretation of the type $\mathbb{0} \to \mathbb{1}$ contains all relations, whereas the interpretation of, for example, $\mathbb{0} \to \mathsf{Int}$ does not contain the relation $\{\{\mathfrak{d}\}, \mathtt{true}\}$, thus making the two distinct. This ensures that all arrow types, including those where ? occurs contravariantly, verify the usual variance properties for subtyping. This is a crucial aspect that will be needed in the proof of Proposition 12.17, on which the soundness of our semantics relies.

$$[\mathrm{T}^{\mathrm{G}}_{\mathrm{Cst}}]\ \frac{}{\Gamma \vdash c : b_c} \qquad [\mathrm{T}^{\mathrm{G}}_{\mathrm{Var}}]\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad [\mathrm{T}^{\mathrm{G}}_{\mathrm{Sub}}]\ \frac{\Gamma \vdash \mathbf{E} : \tau \quad \tau \leq \tau'}{\Gamma \vdash \mathbf{E} : \tau'}$$

$$[\mathrm{T}^{\mathrm{G}}_{\mathrm{Abs}}]\ \frac{\Gamma, x : \tau \vdash \mathbf{E} : \tau'}{\Gamma \vdash \lambda x{:}\tau.\, \mathbf{E} : \tau \to \tau'} \qquad [\mathrm{T}^{\mathrm{G}}_{\mathrm{App}}]\ \frac{\Gamma \vdash \mathbf{E}_1 : \tau' \to \tau \quad \Gamma \vdash \mathbf{E}_2 : \tau'}{\Gamma \vdash \mathbf{E}_1\, \mathbf{E}_2 : \tau}$$

$$[\mathrm{T}^{\mathrm{G}}_{\mathrm{Cast+}}]\ \frac{\Gamma \vdash \mathbf{E} : \tau}{\Gamma \vdash \mathbf{E}\langle \tau \Rightarrow_{\ell} \tau' \rangle : \tau'}\ \tau \preccurlyeq \tau' \qquad [\mathrm{T}^{\mathrm{G}}_{\mathrm{Cast-}}]\ \frac{\Gamma \vdash \mathbf{E} : \tau}{\Gamma \vdash \mathbf{E}\langle \tau \Rightarrow_{\bar{\ell}} \tau' \rangle : \tau'}\ \tau' \preccurlyeq \tau$$

FIGURE 12.1. Typing rules for λ_{G}

12.2. Semantics of a simply-typed gradual calculus

In this section, we use the formalism we developed in the previous section to present a denotational semantics for a cast calculus. However, due to the complexity of this task, and for reasons we will highlight throughout this section, we restrict ourselves to a cast calculus with simple types.

12.2.1. Presentation of λ_{G}

In the rest of this chapter, we restrict the set of types GTypes to simple gradual types: we remove the set-theoretic connectives and the product constructor from the grammar of types. In the following, the set GTypes of gradual types therefore refers to the types τ defined inductively as follows:

$$\mathsf{GTypes} \ni \tau ::= b \mid \tau \to \tau \mid ?$$

where b ranges over the set of basic types $\mathscr{B}$.

We consider a standard cast calculus, similar to the cast calculus we presented in Chapter 4, except we do not consider polymorphism. This yields a calculus that is close to the calculus presented by Siek et al. [68]. The *terms* $\mathbf{E} \in \mathsf{Terms}^{\mathrm{G}}$ of λ_{G} are those defined inductively by the following grammar:

$$\mathsf{Terms}^{\mathrm{G}} \ni \mathbf{E} \quad ::= \quad c \mid x \mid \lambda x{:}\tau.\, \mathbf{E} \mid \mathbf{E}\, \mathbf{E} \mid \mathbf{E}\langle \tau \Rightarrow_p \tau \rangle$$

The type system of λ_{G} is presented in Figure 12.1. This type system is identical to the type system of the cast calculus presented in Chapter 4. In particular, we establish a correspondence between the direction of a cast and the polarity of its label: a cast with a positive label goes from a type to a more precise type, while a cast with a negative label goes from a type to a less precise type.

The *values* $\mathbf{v} \in \mathsf{Values}^{\mathrm{G}}$ and the *reduction contexts* $\mathscr{E}$ of λ_{G} are those defined inductively by the following grammars:

$$\begin{aligned} \mathsf{Values}^{\mathrm{G}} \ni \mathbf{v} \quad &::= \quad c \mid \lambda x{:}\tau.\, \mathbf{E} \mid \mathbf{v}\langle \eta \Rightarrow_p ? \rangle \mid \mathbf{v}\langle \tau \to \tau \Rightarrow_p \tau \to \tau \rangle \\ \mathscr{E} \quad &::= \quad [\,] \mid \mathscr{E}\, \mathbf{E} \mid \mathbf{v}\, \mathscr{E} \mid \mathscr{E}\langle \tau \Rightarrow_p \tau \rangle \end{aligned}$$

$$
\begin{array}{rrcll}
[\mathrm{R}^{\mathrm{G}}_{\mathrm{App}}] & (\lambda x{:}\tau.\,\mathbf{E})\,\mathbf{v} & \rightsquigarrow & \mathbf{E}\,[\mathbf{v}/x] & \\
[\mathrm{R}^{\mathrm{G}}_{\mathrm{CApp}}] & (\mathbf{v}\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle)\,\mathbf{v}' & \rightsquigarrow & (\mathbf{v}\mathbf{v}'\langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle)\langle \tau_2 \Rightarrow_p \tau_2' \rangle & \\
[\mathrm{R}^{\mathrm{G}}_{\mathrm{Id}}] & \mathbf{v}\langle ? \Rightarrow_p ? \rangle & \rightsquigarrow & \mathbf{v} & \\
[\mathrm{R}^{\mathrm{G}}_{\mathrm{ExpandL}}] & \mathbf{v}\langle \tau_1 \to \tau_2 \Rightarrow_p ? \rangle & \rightsquigarrow & \mathbf{v}\langle \tau_1 \to \tau_2 \Rightarrow_p ? \to ? \rangle\langle ? \to ? \Rightarrow_p ? \rangle & \text{if } \tau_1 \to \tau_2 \neq ? \to ? \\
[\mathrm{R}^{\mathrm{G}}_{\mathrm{ExpandR}}] & \mathbf{v}\langle ? \Rightarrow_p \tau_1 \to \tau_2 \rangle & \rightsquigarrow & \mathbf{v}\langle ? \Rightarrow_p ? \to ? \rangle\langle ? \to ? \Rightarrow_p \tau_1 \to \tau_2 \rangle & \text{if } \tau_1 \to \tau_2 \neq ? \to ? \\
[\mathrm{R}^{\mathrm{G}}_{\mathrm{Collapse}}] & \mathbf{v}\langle \eta \Rightarrow_p ? \rangle\langle ? \Rightarrow_q \eta' \rangle & \rightsquigarrow & \mathbf{v} & \text{if } \mathrm{gnd}(\mathbf{v}) \leq \eta' \\
[\mathrm{R}^{\mathrm{G}}_{\mathrm{Blame}}] & \mathbf{v}\langle \eta \Rightarrow_p ? \rangle\langle ? \Rightarrow_q \eta' \rangle & \rightsquigarrow & \text{blame } q & \text{if } \mathrm{gnd}(\mathbf{v}) \not\leq \eta' \\
[\mathrm{R}^{\mathrm{G}}_{\mathrm{Ctx}}] & \mathscr{E}\,[\mathbf{E}] & \rightsquigarrow & \mathscr{E}\,[\mathbf{E}'] & \text{if } \mathbf{E} \rightsquigarrow \mathbf{E}' \\
[\mathrm{R}^{\mathrm{G}}_{\mathrm{CtxBlame}}] & \mathscr{E}\,[\mathbf{E}] & \rightsquigarrow & \text{blame } p & \text{if } \mathbf{E} \rightsquigarrow \text{blame } p
\end{array}
$$

FIGURE 12.2. Operational semantics of λ_{G}

where η ranges over *ground types* defined as follows:

$$\eta ::= ? \to ? \mid b$$

As anticipated, this definition of values is fairly standard and follows closely the definition we gave in Chapter 4.

To conclude the presentation of λ_{G}, its operational semantics is summarized in Figure 12.2. Once again, most of this semantics is common in the gradual typing literature. The only peculiarity comes from the rules $[\mathrm{R}^{\mathrm{G}}_{\mathrm{Collapse}}]$ and $[\mathrm{R}^{\mathrm{G}}_{\mathrm{Blame}}]$ which make use of an operator noted $\mathrm{gnd}(.)$. This operator returns the ground type associated to a value, and is defined inductively as follows:

$$
\begin{array}{rclrcl}
\mathrm{gnd}(c) & = & b_c & \mathrm{gnd}(\lambda x{:}\tau.\,\mathbf{E}) & = & ? \to ? \\
\mathrm{gnd}(\mathbf{v}\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle) & = & ? \to ? & \mathrm{gnd}(\mathbf{v}\langle \eta \Rightarrow_p ? \rangle) & = & \mathrm{gnd}(\mathbf{v})
\end{array}
$$

The reason these rules use the operator $\mathrm{gnd}(.)$ instead of directly comparing the two ground types η and η' is to avoid some unnecessary error that may arise due to subtyping. For example, consider the expression $2\langle \mathsf{Int} \Rightarrow_p ? \rangle\langle ? \Rightarrow_q \mathsf{Nat} \rangle$. If the result of the reduction were decided by comparing the source type of the first cast and the target type of the second cast, this expression would fail since Int is not a subtype of Nat. However, it is clear that 2 is of type Nat, thus failing is not necessary here[1], and the expression can be safely reduced to 2. This is the role of the operator $\mathrm{gnd}(.)$: it extracts the most precise ground type associated to a value, which ensures that rule $[\mathrm{R}^{\mathrm{G}}_{\mathrm{Blame}}]$ is applied as little as possible.

12.2.2. A new interpretation of types

The interpretation of types we presented in Section 12.1 (Definition 12.5) is based on the intuition that the values belonging to a gradual type τ can be separated into two categories: the values that belong to *every* materialization of τ (we annotate these values with the tag `!`), and the values that belong to *some* materializations of τ (we annotate these values with the tag `?`). Using this intuition, we obtained that, for example, the interpretation of the type $? \to ?$ contains in particular all possible relations tagged with `?`, since every relation belongs to at least one arrow type.

This interpretation of types is particularly useful to define and reason about the subtyping and precision relations set-theoretically. This makes it well-suited to the definition of a gradually-

[1] This corresponds to what is sometimes called *forgetful semantics for casts* in the gradual typing litterature.

typed source language, in which the use of precision is implicit. However, in a cast language, every use of the precision relation is made explicit by the presence of a cast, and the dynamic type ? behaves similarly to a base type: the values of the cast language that can be given type ? are precisely the values that have been explicitly cast to ?. Similarly, the values of the cast language that have type ? → ? are exactly the functions that, when applied to a value that has been explicitly cast to ?, return a value cast to ?. This contrasts with the static semantics of a source language where applying a function of type ? → ? to a value of type ? can return a result of any type, since both the type of the function and the type of the value can be materialized to anything.

This leads us to defining a new interpretation of gradual types which is guided by the behaviour of the values of our cast language λ_{G}. This new interpretation is given as a function $\langle\!\langle . \rangle\!\rangle : \mathsf{GTypes} \to \mathscr{P}(\mathscr{D}^{\mathrm{G}})$ defined as follows.

Definition 12.16 (Concrete interpretation of gradual types)**.** *We define a binary predicate $\langle \hat{\partial} : \tau \rangle$ ("the element $\hat{\partial}$ belongs to the type τ") where $\hat{\partial} \in \mathscr{D}^{G}_{\Omega} \cup \{\mathfrak{O}\}$ and $\tau \in \mathsf{GTypes}$, by induction on the pair $(\hat{\partial}, \tau)$ ordered lexicographically. The predicate is defined as follows:*

$$
\begin{aligned}
\langle \mathsf{blame}\ p : \tau \rangle &= \mathit{true} \\
\langle \mathfrak{O} : \tau \rangle &= \mathit{true} \\
\langle c^{\iota} : b \rangle &= c \in \mathbb{B}(b) \\
\langle c^{?} : ? \rangle &= \mathit{true} \\
\langle \{(S_1, \partial_1), \ldots, (S_n, \partial_n)\}^{\iota} : \tau_1 \to \tau_2 \rangle &= \forall i \in \{1..n\}.\ \mathit{if}\ \exists \iota \in S_i.\ \langle \iota : \tau_1 \rangle\ \mathit{then}\ \langle \partial_i : \tau_2 \rangle \\
\langle \{(S_1, \partial_1), \ldots, (S_n, \partial_n)\}^{?} : ? \rangle &= \forall i \in \{1..n\}.\ \mathit{if}\ \exists \iota \in S_i.\ \langle \iota : ? \rangle\ \mathit{then}\ \langle \partial_i : ? \rangle \\
\langle \partial : \tau \rangle &= \mathit{false} \qquad \mathit{otherwise}
\end{aligned}
$$

We define the concrete interpretation *of gradual types $\langle\!\langle . \rangle\!\rangle : \mathsf{GTypes} \to \mathscr{P}(\mathscr{D}^{G})$ as $\langle\!\langle \tau \rangle\!\rangle = \{d \in \mathscr{D}^{G} \mid \langle d : \tau \rangle\}$.*

Under this interpretation, tags take on another meaning: an element tagged with ? corresponds to a value that has been explicitly cast to ?, whereas an element tagged with ! corresponds to a value that has not been cast, or that has been cast to a type different from ?. Following this idea, the values that can be given type b are exactly the constants c such that $c \in \mathbb{B}(b)$ and that have not been cast to ?, hence the concrete interpretation of b as the set $\{c^{!} \mid c \in \mathbb{B}(b)\}$. Similarly, every constant that has been cast to ? can be given type ?, hence, the concrete interpretation of ? contains all elements $c^{?} \in \mathscr{D}^{\mathrm{G}}$.

The interpretation of arrow types is fairly straightforward and closely mimics the interpretation of arrow types presented in Chapter 10 (Definition 10.2). The only peculiarity comes from the fact that since a function can only be given type $\tau_1 \to \tau_2$ if it has not been cast to ? (as ? is not a subtype of $\tau_1 \to \tau_2$), a relation can only belong to the concrete interpretation of $\tau_1 \to \tau_2$ if its tag is !.

Finally, the second-to-last rule of Definition 12.16, which formalizes whether a relation belongs to the concrete interpretation of ?, is probably the most surprising. Its definition comes from the operational semantics of λ_{G} presented in Figure 12.2: for a functional value to be cast to ?, it must first be cast to ? → ?, as enforced by Rule $[\mathrm{R}^{\mathrm{G}}_{\mathsf{ExpandL}}]$. Therefore, a relation that belongs to the concrete interpretation of ? must satisfy two conditions: it must be tagged with ?, and it must be a function that maps values of type ? to values of type ?.

While this new interpretation of types is based on a completely different intuition than the interpretation presented in Section 12.1, the two are not incompatible. In particular, they both induce the same subtyping relation on GTypes, as formalized by the following result. This result will be crucial to prove that our semantics is sound with respect to the type system of Figure 12.1 which uses the subtyping relation defined in Section 12.1.

Proposition 12.17. *For every types* $\tau, \tau' \in \mathsf{GTypes}$, $\langle\!\langle \tau \rangle\!\rangle \subseteq \langle\!\langle \tau' \rangle\!\rangle \iff \tau \leq \tau'$.

Proof hint. See appendix page 285 for the full proof.
The main difficulty comes from arrow types, and highlights the reason why $\mho$ is necessary. Let $\tau = \tau_1 \to \tau_2$ and $\tau' = \tau'_1 \to \tau'_2$, and suppose that $\tau \leq \tau'$. We can show that τ and τ' necessarily verify the usual variance properties.
Suppose that $\tau'_1 \not\leq \tau_1$. Then there exists $d \in [\![\tau'_1]\!]^{\mathrm{G}} \setminus [\![\tau_1]\!]^{\mathrm{G}}$. By Proposition 12.6, we have $d^? \in [\![\tau'_1]\!]^{\mathrm{G}} \setminus [\![\tau_1]\!]^{\mathrm{G}}$. By Definition 12.5, we have that $\{\{d^?\}, \Omega\}^! \in [\![\tau]\!]^{\mathrm{G}} \setminus [\![\tau']\!]^{\mathrm{G}}$, which contradicts the hypothesis that $\tau \leq \tau'$. Hence, $\tau'_1 \leq \tau_1$.
Similarly, suppose that $\tau_2 \not\leq \tau'_2$. There exists $d \in [\![\tau_2]\!]^{\mathrm{G}} \setminus [\![\tau'_2]\!]^{\mathrm{G}}$. By Proposition 12.6, we have $d^? \in [\![\tau_2]\!]^{\mathrm{G}}$. By Definition 12.5, we have that $\{\{\mho\}, d^?\}^! \in [\![\tau]\!]^{\mathrm{G}} \setminus [\![\tau']\!]^{\mathrm{G}}$, which once again contradicts the hypothesis. Hence, $\tau_2 \leq \tau'_2$.
Then, using these variance properties, we easily prove that $\langle\!\langle \tau \rangle\!\rangle \subseteq \langle\!\langle \tau' \rangle\!\rangle$ by induction hypothesis and Definition 12.16. □

As a final remark, one of the main reasons we restricted this calculus to simple types is that this interpretation of types cannot be easily extended to set-theoretic types. A first issue comes from the fact that, in the presence of set-theoretic types, ? does not behave exactly like a base type anymore, since its intersection with any other type (apart from the empty type) is non-empty. As such, it is possible to cast 3 to, for example, the type Int ∧ ?. However, if we were to interpret the intersection set-theoretically in the above interpretation, the intersection of Int (whose interpretation only contains values tagged with !) with ? (whose interpretation only contains values tagged with ?) would be empty. This suggests that in a cast language with set-theoretic types, values behave entirely differently than in a cast language with simple types, and thus the concrete interpretation of types must be changed accordingly. We will discuss this point in more details in Chapter 13.

12.2.3. The denotational semantics of casts

The central part of our semantics is the definition of the semantics of casts from a denotational perspective. Operationally, the role of a cast is simple: it converts a value of type τ into a value of another type τ' if this operation is possible, or raises an error otherwise. For constants, there are only two operations possible: converting a constant into a "boxed" constant (a constant cast to ?), and converting a "boxed" constant into an "unboxed" constant (a constant without any cast attached). For functions, this is more complex, since it is also possible to cast a function from a type $\tau_1 \to \tau_2$ to another type $\tau'_1 \to \tau'_2$, which produces another function that can now be applied to arguments of type τ'_1 and returns results of type τ'_2. Moreover, "boxing" a function by casting it to the dynamic type introduces an intermediate cast to the type $? \to ?$, whose role is to ensure that casting a function from a type $\tau_1 \to \tau_2$ to ? and later from type ? to another type $\tau'_1 \to \tau'_2$ always goes through an intermediate type that is compatible (with respect to precision) with both $\tau_1 \to \tau_2$ and $\tau'_1 \to \tau'_2$.

We formalize these same operations denotationally, by seeing a cast as an operation that converts an element $\partial \in \mathscr{D}_\Omega^{\mathrm{G}}$ into a set of elements of $\mathscr{D}_\Omega^{\mathrm{G}}$. The reason this operation can return multiple elements (possibly infinitely many) is due to the fact that casting a relation to a type $\tau_1' \to \tau_2'$ can return many relations of this type. For example, consider the relation of type $? \to ?$ that maps $3^?$ to $3^?$. Casting it to $\mathsf{Int} \to \mathsf{Int}$ should intuitively produce the relation that maps $3^!$ to $3^!$, since it can now be applied to unboxed arguments, and it now returns unboxed results. However, it can also produce the relation that maps $\mathtt{true}^!$ to Ω, since applying a relation cast to $\mathsf{Int} \to \mathsf{Int}$ to an argument of type Bool should clearly return a type error.

The formal definition of this operation, which we call the *coercion* of an element $\partial \in \mathscr{D}_\Omega^{\mathrm{G}}$ by a cast, is presented in the following definition. To ease the notation, we define $\mathscr{F}^{\mathrm{G}} = \mathscr{P}_f(\mathscr{D}^{\mathrm{G}})$ to range over finite sets that do not contain $\mho$, as $\mho$ does not intervene in the denotational semantics of the language.

Definition 12.18 (Coercion of denotations)**.** *Given a cast $\langle \tau \Rightarrow_p \tau' \rangle$, we define the* coercion *function $(.)\langle \tau \Rightarrow_p \tau' \rangle : \mathscr{D}_\Omega^G \to \mathscr{P}(\mathscr{D}_\Omega^G)$ by induction on its argument $\partial \in \mathscr{D}_\Omega^G$ and cases on τ and τ' as follows:*

$$
\begin{array}{rcll}
c^g\langle \tau \Rightarrow_p ? \rangle & = & \{c^?\} & (1)\\
c^g\langle \tau \Rightarrow_p \tau' \rangle & = & \{c^!\} \quad \textit{if } \tau' \neq ? \textit{ and } b_c \leq \tau' & (2)\\
c^g\langle \tau \Rightarrow_p \tau' \rangle & = & \{\mathsf{blame}\ p\} \quad \textit{if } \tau' \neq ? \textit{ and } b_c \not\leq \tau' & (3)\\
R^!\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle & = & \{R'^! \mid R' \in \mathscr{P}_f(\mathscr{F}^G \times \mathscr{D}_\Omega^G),\ \forall (S', \partial') \in R', & \\
 & & \quad \textit{either } (S' \subseteq \langle\!\langle \tau_1' \rangle\!\rangle \textit{ and } \exists (S, \partial) \in R,\ S \subseteq S'\langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle,\ \partial' \in \partial\langle \tau_2 \Rightarrow_p \tau_2' \rangle) & \\
 & & \quad \textit{or } (S' \subseteq \langle\!\langle \tau_1' \rangle\!\rangle \textit{ and } \mathsf{blame}\ q \in S'\langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle \textit{ and } \partial' = \mathsf{blame}\ q) & \\
 & & \quad \textit{or } (S' \cap \langle\!\langle \tau_1' \rangle\!\rangle = \emptyset \textit{ and } \partial' = \Omega)\} & (4)\\
R^!\langle ? \to ? \Rightarrow_p ? \rangle & = & \{R^?\} & (5)\\
R^?\langle ? \Rightarrow_p ? \to ? \rangle & = & \{R^!\} & (6)\\
R^!\langle \tau_1 \to \tau_2 \Rightarrow_p ? \rangle & = & \{\partial^? \mid \partial \in R^!\langle \tau_1 \to \tau_2 \Rightarrow_p ? \to ? \rangle\} \quad \textit{if } (\tau_1, \tau_2) \neq (?, ?) & (7)(*)\\
R^?\langle ? \Rightarrow_p \tau_1 \to \tau_2 \rangle & = & R^!\langle ? \to ? \Rightarrow_p \tau_1 \to \tau_2 \rangle \quad \textit{if } (\tau_1, \tau_2) \neq (?, ?) & (8)(*)\\
R^?\langle ? \Rightarrow_p ? \rangle & = & \{R^?\} & (9)\\
R^?\langle ? \Rightarrow_p b \rangle & = & \{\mathsf{blame}\ p\} & (10)\\
\mathsf{blame}\ q\langle \tau \Rightarrow_p \tau' \rangle & = & \{\mathsf{blame}\ q\} & (11)\\
\partial\langle \tau \Rightarrow_p \tau' \rangle & = & \{\Omega\} \quad \textit{otherwise} & (12)
\end{array}
$$

where $S\langle \tau \Rightarrow_p \tau' \rangle$ is defined for $S \in \mathscr{F}^G$ as:

$$S\langle \tau \Rightarrow_p \tau' \rangle = \bigcup_{d \in S} d\langle \tau \Rightarrow_p \tau' \rangle$$

$(*)$ *For a proper inductive definition, case* (4) *should be expanded in cases* (7) *and* (8)*. However, we choose to keep this informal notation for the sake of concision.*

In essence, given a cast $\langle \tau \Rightarrow_p \tau' \rangle$, the role of this definition is to convert every element belonging to $\langle\!\langle \tau \rangle\!\rangle$ into a set of elements that belong to $\langle\!\langle \tau' \rangle\!\rangle$. Let us explain the various rules this definition comprises. First, remark that they are split into two main categories: rules that deal with constants (rules (1) to (3)), and rules that deal with relations (rules (4) to (10)). Rules (11) and (12) are just there to propagate blame through casts, and to produce a type error if the element that is being coerced does not belong to the concrete interpretation of the source type of the cast, respectively.

As anticipated, we distinguish three cases for constants. First, coercing a constant c^g (independently of its tag) to ? returns the same constant tagged with ? (rule (1)). Note that if $g = ?$, this simply corresponds to an identity cast of the form $c\langle b \Rightarrow_p ?\rangle\langle ? \Rightarrow_q ?\rangle$, which is simply moved operationally by Rule $[\mathrm{R}^{\mathrm{G}}_{\mathsf{Id}}]$ to return the boxed constant $c\langle b \Rightarrow_p ?\rangle$. Second, coercing a constant c^g to a type τ different from ? can produce two different results: either the type of c is a subtype of τ, in which case the constant can be unboxed, producing the element $c^!$ (rule (2)), or the type of c is not a subtype of τ, in which case the cast must be blamed (rule (3)).

For functions, the heart of the definition comes from rule (4), which may seem complex. However, this rule is a simple reinterpretation of the reduction rule $[\mathrm{R}^{\mathrm{G}}_{\mathsf{CApp}}]$ of Figure 12.2 from a denotational perspective. Consider a relation $R^!$ that belongs to the concrete interpretation of a type $\tau_1 \to \tau_2$. When coercing it to another type $\tau_1' \to \tau_2'$, we obtain the relations $R'^!$ whose inputs and outputs satisfy several conditions. First, if an input of R' belongs to the concrete interpretation of τ_1' and can be coerced to an input of R by a cast going from τ_1' to τ_1, then the result of R' can be obtained by casting the corresponding result of R from τ_2 to τ_2'. This is, in essence, the denotational equivalent of Rule $[\mathrm{R}^{\mathrm{G}}_{\mathsf{CApp}}]$, and this corresponds to the first part of rule (4) of Definition 12.18. However, there is a corner case we need to take into account, which is the case when the coercion of an input of R' from τ_1' to τ_1 fails. In this case, according to Rule $[\mathrm{R}^{\mathrm{G}}_{\mathsf{Capp}}]$ and the propagation of blame by Rule $[\mathrm{R}^{\mathrm{G}}_{\mathsf{CtxBlame}}]$, the application must fail: hence R' outputs a blame in this case. This is the second part of rule (4). Finally, as anticipated, if an input of R' does not belong to the concrete interpretation of τ_1', then the output is simply a type error Ω, as stated by the last part of rule (4).

The rest of the rules is easier. Rules (5) and (6) correspond, respectively, to the boxing (tagging with ?) and unboxing (tagging with !) of relations that have been cast from and to $? \to ?$. When considering a cast that goes immediately from a type $\tau_1 \to \tau_2$ to ? without going through $? \to ?$ as an intermediate type, we simply merge rules (4) and (5) to mimic the introduction of $? \to ?$ as an intermediate type. This yields rule (7). Similarly, rule (8) deals with the other direction, where a cast goes from a type ? to a type $\tau_1 \to \tau_2$ distinct from $? \to ?$. Rule (9) is straightforward and simply ignores an identity cast going from ? to ?. Finally, rule (10) produces a blame when a boxed function is being cast to a type that is not an arrow type.

Using this operation, defining the denotational semantics of a cast expression $\mathbf{E}\langle\tau \Rightarrow_p \tau'\rangle$ is only a matter of coercing every element of the denotation of $\mathbf{E}$ by the cast $\langle\tau \Rightarrow_p \tau'\rangle$. Formally, we obtain that:

$$[\![\mathbf{E}\langle\tau \Rightarrow_p \tau'\rangle]\!]^{\mathrm{G}}_{\rho} = \bigcup_{\partial \in [\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho}} \partial\langle\tau \Rightarrow_p \tau'\rangle$$

Note that it is not necessary to handle and propagate Ω separately if it occurs in the semantics of $\mathbf{E}$, as we have done before (in the case of the application in Definition 10.5 for example) with the help of the operator $\Omega^{(.)}_{(.)}$. Definition 12.18 already takes care of correctly propagating errors including blame, since $\Omega\langle\tau \Rightarrow_p \tau'\rangle = \{\Omega\}$ and $\mathsf{blame}\ q\langle\tau \Rightarrow_p \tau'\rangle = \{\mathsf{blame}\ q\}$.

12.2.4. Denotational semantics of $\lambda_{\mathbf{G}}$

We now present the rest of the denotational semantics of λ_{G}, which is fairly straightforward and mostly follows the semantics presented in the previous chapters, the only major difference being the addition of blame and tags.

Definition 12.19 (Set-theoretic interpretation of λ_{G})**.** *Let $\rho \in \mathsf{Envs}$. We define the set-*

theoretic interpretation of λ_G as a function $[\![.]\!]^G_{(.)} : \mathsf{Terms}^G \to \mathsf{Envs} \to \mathscr{P}_f(\mathscr{D}^G_\Omega)$ as follows:

$$
\begin{array}{rcl}
[\![x]\!]^G_\rho & = & \rho(x) \\
[\![c]\!]^G_\rho & = & \{c^!\} \\
[\![\lambda x{:}\tau.\,\mathbf{E}]\!]^G_\rho & = & \{R^! \mid R \in \mathscr{P}_f(\mathscr{F}^G \times \mathscr{D}^G_\Omega),\ \forall (S,\partial) \in R, \\
 & & \quad \textit{either} \quad S \subseteq \langle\!\langle \tau \rangle\!\rangle \qquad \textit{and } \partial \in [\![\mathbf{E}]\!]^G_{\rho, x \mapsto S} \\
 & & \quad\ \ \textit{or} \quad S \cap \langle\!\langle \tau \rangle\!\rangle = \emptyset \quad \textit{and } \partial = \Omega\} \\
[\![\mathbf{E}_1\,\mathbf{E}_2]\!]^G_\rho & = & \{\partial \in \mathscr{D}^G_\Omega \mid \exists S \subseteq [\![\mathbf{E}_2]\!]^G_\rho, R^! \in [\![\mathbf{E}_1]\!]^G_\rho, (S,\partial) \in R\} \cup \Omega^\rho_{\mathbf{E}_1\,\mathbf{E}_2} \cup \mathbb{B}^\rho_{\mathbf{E}_1\,\mathbf{E}_2} \\
[\![\mathbf{E}\langle \tau \Rightarrow_p \tau' \rangle]\!]^G_\rho & = & \bigcup_{\partial \in [\![\mathbf{E}]\!]^G_\rho} \partial\langle \tau \Rightarrow_p \tau' \rangle
\end{array}
$$

As anticipated, elements tagged with ? correspond to values that have been explicitly cast to ?. Hence, λ-abstractions and constants, which are uncast values, are respectively interpreted as relations and constants tagged with ! only. Apart from the addition of the tag, the interpretation of λ-abstractions is identical to the interpretation we gave in Chapter 10, except we now use the concrete interpretation of the annotation $\langle\!\langle \tau \rangle\!\rangle$ to determine whether the input is well-typed or not.

The interpretation of applications is also mostly identical to the interpretation of Chapter 10, apart from two additions. First, notice that we only consider relations that are tagged with !: this is because a function that has been cast to ? cannot be applied, as ? is not an arrow type. Hence, the function must be unboxed by being cast to an arrow type before it can be applied, and this operation will produce relations that are tagged with !. Second, we introduced a new operator, $\mathbb{B}^{(.)}_{(.)}$, which is similar to $\Omega^{(.)}_{(.)}$, but whose role is to propagate blame instead of Ω. It is defined as follows.

Definition 12.20 (Blame operator $\mathbb{B}^{(.)}_{(.)}$)**.** *For every term $\mathbf{E} \in \mathsf{Terms}^G$, and every environment $\rho \in \mathsf{Envs}$, we define the set $\mathbb{B}^\rho_{\mathbf{E}} \subseteq \mathbb{B}\mathsf{lame}$ as follows:*

1. *if $\mathbf{E} \equiv \mathbf{E}_1\,\mathbf{E}_2$ then*
$$
\begin{cases}
\mathbb{B}^\rho_{\mathbf{E}} = [\![\mathbf{E}_1]\!]^G_\rho \cap \mathbb{B}\mathsf{lame} & \textit{if } [\![\mathbf{E}_1]\!]^G_\rho \cap \mathbb{B}\mathsf{lame} \neq \emptyset \\
\mathbb{B}^\rho_{\mathbf{E}} = [\![\mathbf{E}_2]\!]^G_\rho \cap \mathbb{B}\mathsf{lame} & \textit{if } [\![\mathbf{E}_1]\!]^G_\rho \neq \emptyset \\
\mathbb{B}^\rho_{\mathbf{E}} = \emptyset & \textit{otherwise}
\end{cases}
$$
2. *$\mathbb{B}^\rho_{\mathbf{E}} = \emptyset$ otherwise*

The idea is similar to the definition of the operator $\Omega^{(.)}_{(.)}$ given in the previous chapters. Given an application $\mathbf{E}_1\,\mathbf{E}_2$, if the semantics of $\mathbf{E}_1$ produces a blame, then this blame is propagated in the semantics of the application. If the semantics of $\mathbf{E}_2$ produces a blame *and* the semantics of $\mathbf{E}_1$ is non-empty (that is, $\mathbf{E}_1$ does not diverge), then the blame is propagated in the semantics of the application. Otherwise, no blame is propagated by the operator.

Lastly, we also need to modify the operation of the operator $\Omega^{(.)}_{(.)}$ to account for tags. In particular, as we explained before, an application in which the function has been cast to ? is not well-typed. We reflect this is the definition of $\Omega^{(.)}_{(.)}$ by stating that if, in an application $\mathbf{E}_1\,\mathbf{E}_2$, the semantics of $\mathbf{E}_1$ contains an element that is not a relation tagged with !, then the application is ill-typed and its semantics contains Ω.

Definition 12.21 (Failure operator $\Omega^{(.)}_{(.)}$ for λ_G)**.** *For every term $\mathbf{E} \in \mathsf{Terms}^G$, and every environment $\rho : \mathsf{Envs}$, we have $\Omega^\rho_{\mathbf{E}} = \{\Omega\}$ if any of the following conditions holds*

1. $\mathbf{E} \equiv \mathbf{E}_1\,\mathbf{E}_2$ *and* $\Omega \in [\![\mathbf{E}_1]\!]^G_\rho$ *or* $[\![\mathbf{E}_1]\!]^G_\rho \neq \emptyset$ *and* $\Omega \in [\![\mathbf{E}_2]\!]^G_\rho$
2. $\mathbf{E} \equiv \mathbf{E}_1\,\mathbf{E}_2$ *where* $[\![\mathbf{E}_2]\!]^G_\rho \neq \emptyset$ *and* $\exists d \in [\![\mathbf{E}_1]\!]^G_\rho$ *such that* $\nexists R \in \mathscr{P}_f(\mathscr{F}^G \times \mathscr{D}^G_\Omega)$, $d = R^!$

and $\Omega^\rho_{\mathbf{E}} = \emptyset$ *otherwise.*

As anticipated, the two operators defined above only apply to applications, as the propagation of errors through cast expressions is already handled by the coercion operation presented in Definition 12.18.

12.3. Soundness properties

Having presented the denotational semantics of λ_{G}, we now state and prove its properties. We prove the same properties as in the previous chapters, namely its type soundness and its computational soundness, while leaving its adequacy as a conjecture.

12.3.1. Type soundness

The first property of λ_{G} we prove is its type soundness. It is stated exactly as in Chapter 10, except it uses the concrete interpretation of types both in the denotational interpretation of type environments and in the statement of the theorem itself. The denotational interpretation of type environments is defined exactly as in Definition 10.13, except we replace $[\![.]\!]^{\mathrm{F}}$ by $\langle\!\langle . \rangle\!\rangle$.

Definition 12.22 (Denotational interpretation of Γ)**.** *Let* $\Gamma \in \mathsf{TEnvs}$*. We define its denotational interpretation, noted* $[\![\Gamma]\!]^G$*, as the function*

$$[\![.]\!]^G : \mathsf{TEnvs} \to \mathscr{P}(\mathsf{Envs})$$
$$[\![\Gamma]\!]^G = \{\rho \in \mathsf{Envs} \mid \forall x \in \mathrm{Dom}\,(\Gamma).\ \rho(x) \subseteq \langle\!\langle \Gamma(x) \rangle\!\rangle\}$$

To prove the type soundness of λ_{G}, we rely on the following lemma which formalizes the intuition that coercing an element of type τ to a type τ' produces a set of elements of type τ'.

Lemma 12.23. *For every cast* $\langle \tau \Rightarrow_p \tau' \rangle$ *and every* $\partial \in \mathscr{D}^G_\Omega$*, if* $\partial \in \langle\!\langle \tau \rangle\!\rangle$ *then* $\partial\langle \tau \Rightarrow_p \tau' \rangle \subseteq \langle\!\langle \tau' \rangle\!\rangle$*.*

Proof. See appendix page 286. □

This lemma ensures the type soundness of the semantics of cast expressions. The type soundness of the rest of the semantics is fairly straightforward and its proof follows the same strategy as for the semantics presented in Chapter 10.

Theorem 12.24 (Type soundness for λ_{G})**.** *For every type environment* $\Gamma \in \mathsf{TEnvs}$ *and every term* $\mathbf{E} \in \mathsf{Terms}^G$*, if* $\Gamma \vdash \mathbf{E} : \tau$ *then for every* $\rho \in [\![\Gamma]\!]^G$*,* $[\![\mathbf{E}]\!]^G_\rho \subseteq \langle\!\langle \tau \rangle\!\rangle$*.*

Proof. See appendix page 287. □

12.3.2. Computational soundness

The second property we prove is the computational soundness property, whose statement differs slightly from the previous chapters, in the sense that we must now account for blame. That is, if an expression $\mathbf{E}$ reduces to $\mathbf{E}'$, then we still show that the semantics of $\mathbf{E}$ and $\mathbf{E}'$ are equal. However, if an expression $\mathbf{E}$ reduces to blame p, then we show that the semantics of $\mathbf{E}$ is exactly equal to $\{\mathsf{blame}\ p\}$.

As in Chapter 10, the proof of this property relies on a monotonicity lemma and a substitution lemma which we state here, and whose proofs are fairly straightforward by induction on the term at hand.

Lemma 12.25. *For every term* $\mathbf{E} \in \mathsf{Terms}^G$, $x \in \mathsf{Vars}$, $\rho \in \mathsf{Envs}$, *and* $S_1, S_2 \in \mathscr{P}(\mathscr{D}^G)$, *if* $S_1 \subseteq S_2$ *then* $[\![\mathbf{E}]\!]^G_{\rho,x\mapsto S_1} \subseteq [\![\mathbf{E}]\!]^G_{\rho,x\mapsto S_2}$

Proof. See appendix page 287. □

Lemma 12.26. *For every term* $\mathbf{E} \in \mathsf{Terms}^G$, $\mathbf{v} \in \mathsf{Values}^G$, $x \in \mathsf{Vars}$, $\rho \in \mathsf{Envs}$,

$$[\![\mathbf{E}\,[\mathbf{v}/x]]\!]^G_\rho = \bigcup_{S \in \mathscr{P}_f([\![\mathbf{v}]\!]^G_\rho)} [\![\mathbf{E}]\!]^G_{\rho,x\mapsto S}$$

Proof. See appendix page 288. □

The proof of the computational soundness theorem also relies on two additional lemmas which were not needed in the previous chapters, due to the addition of blame. The first lemma states that the denotational semantics of a value is never empty. It is needed to ensure that a blame produced in the right hand side of an application is correctly propagated by the operator $\mathbb{B}^{(\cdot)}_{(\cdot)}$ if the left hand side is a value.

Lemma 12.27. *For every value* $\mathbf{v} \in \mathsf{Values}^G$ *and every* $\rho \in \mathsf{Envs}$, $[\![\mathbf{v}]\!]^G_\rho \neq \emptyset$.

Proof. See appendix page 290. □

The last lemma we need states that the semantics of a value can never contain a blame, which is crucial for the soundness of our semantics.

Lemma 12.28. *For every value* $\mathbf{v} \in \mathsf{Values}^G$ *and every* $\rho \in \mathsf{Envs}$, $[\![\mathbf{v}]\!]^G_\rho \cap \mathbb{B}\mathsf{lame} = \emptyset$.

Proof. See appendix page 290. □

Finally, we can state and prove the computational soundness of λ_{G}. As anticipated, the theorem features two cases, depending on whether the term at hand reduces to another term or to a blame. This theorem ensures that our denotational semantics is consistent with the operational semantics of λ_{G} for non-diverging terms.

Theorem 12.29 (Computational soundness for λ_{G})**.** *For every term* $\mathbf{E} \in \mathsf{Terms}^G$ *such that*

$\Gamma \vdash \mathbf{E} : \tau$ *and every environment* $\rho \in [\![\Gamma]\!]^G$,

$$\mathbf{E} \rightsquigarrow \mathbf{E}' \implies [\![\mathbf{E}]\!]^G_\rho = [\![\mathbf{E}']\!]^G_\rho$$
$$\mathbf{E} \rightsquigarrow \mathsf{blame}\ p \implies [\![\mathbf{E}]\!]^G_\rho = \{\mathsf{blame}\ p\}$$

Proof. See appendix page 290. □

Chapter 13.

Discussion

In this second part of the manuscript, we have described how to adapt and extend the interpretation of types as sets of values proposed by semantic subtyping to define a denotational semantics for several languages.

We started this work with two goals in mind. The first was to provide a true set-theoretic interpretation of gradual types and a denotational semantics for a gradually-typed language, to better understand the connection between set-theoretic types and gradual types. The second was to bridge the gap between the interpretation of types proposed by Frisch et al. [27] to define their subtyping relation and the interpretation of the values of a language. That is, our aim was to show that the elements of their interpretation domain $\mathscr{D}$ corresponded precisely to the values of a functional language.

However, as we have shown in Chapter 9, even for a very simple λ-calculus, the behaviour of functions cannot be properly represented using the interpretation domain $\mathscr{D}$. The problem comes from the fact that, due to cardinality reasons, functions are represented as *finite* binary relations in $\mathscr{D}$. Since it is possible for a function to accept an infinite number of inputs and to have an infinite number of outputs (especially in a higher-order setting), from a denotational perspective, the only way to properly interpret a function using $\mathscr{D}$ is to interpret it as a possibly infinite set of finite binary relations. However, this causes values to be now interpreted as *sets* of elements of $\mathscr{D}$, whereas the input of a function is still represented as a single element of $\mathscr{D}$, thus breaking the idea that functions map values to values.

This led us to proposing a slight modification to the interpretation domain. In Chapter 10, we have presented a new domain $\mathscr{D}^{\mathrm{F}}$ in which finite relations take *finite sets* of elements as inputs. By only slightly revising the interpretation of types and the semantics presented in Chapter 9 to use set-containment instead of membership when needed, we have obtained a sound and adequate denotational semantics for our simple λ-calculus with set-theoretic types. We have also shown in Theorem 10.10 that our new interpretation of types induces the same subtyping relation as the interpretation proposed by Frisch et al. [27]. This ensures that our work is still in line with our original goal of reconciling semantic subtyping with the interpretation of the values of a language.

Having defined a sound and adequate denotational semantics for a simple functional calculus, we then extended our language with function interfaces, typecases, and non-deterministic choices. According to Frisch et al. [27], these three constructs are necessary to ensure that the subtyping relation presented in Chapter 2 coincides with the subtyping relation that would be induced by directly interpreting types as sets of values. From a more practical point of view, interfaces and typecases along with intersection types provide a way to perform function overloading, which is a powerful and frequently used feature of many programming languages.

This extension yields a language that corresponds to the functional core of ℂDuce . In Chapter 11, we showed how to adapt the interpretation domain $\mathscr{D}^{\mathrm{F}}$ to account for interfaces, typecases,

and non-determinism. The latter was accounted for by adding *marks*, which are strings over the alphabet $\{l, r\}$, to elements of the domain. These marks denote the execution path that led to the production of a result. Interfaces and typecases were accounted for by adding dummy inputs denoted $\mho_d$ whose role is to ensure that, given any relation R belonging to a type t, and any negation type $\neg(s' \to t')$, it is possible to "complete" R to obtain a relation belonging to $t \wedge \neg(s' \to t')$, as long as this intersection is non-empty. This is necessary to ensure that typecases can always be reduced without ambiguity, that is, that the semantics of every value either belongs to a type or its negation. Nevertheless, this was not sufficient, and we had to slightly modify the syntax and type system of $\mathbb{C}$Duce as the original typing rule for λ-abstractions, which allows their type to be arbitrarily refined with negation types, is incompatible with our semantics. Instead, we required λ-abstractions to be explicitly annotated with their negation types, which allowed us to prove the soundness of our semantics.

In the last chapter of this part, Chapter 12, we then tackled our goal of providing a set-theoretic interpretation of gradual types and a denotational semantics for a cast language. We distinguished between the values that belong to every materialization of a type and the values that belong to some materializations of a type, which we reflected in the interpretation domain by adding tags (either ? or !) to the elements of the domain. The interpretation of types that ensued gave us plenty of powerful results: a semantic definition of gradual subtyping, a semantic definition of precision, and a theorem stating that every set-theoretic gradual type can be represented equivalently with a single, top-level occurrence of the dynamic type. Nevertheless, everything is not perfect, as we showed that if the subtyping relation on gradual types is made to be a conservative extension of semantic subtyping as defined in Chapter 2, then it is unsound as it allows subsumptions such as $? \to ? \leq ? \to \mathbb{0}$. Our solution was to slightly modify the interpretation domain by adding again a marker $\mho$ to the input of functions, whose role is to guarantee that, for distinct types t, the types $\mathbb{0} \to t$ are all distinguishable from each other, which in turn prevents unsound subsumptions on gradual types.

We then presented a cast language restricted to simple types and without pairs, to focus on the semantics of casts and applications. We argued that the interpretation of types we presented in the first section was too disconnected from the behaviour of the values of our cast language. For example, our interpretation of the type $? \to ?$ contained all functions, while, in a cast language, a function of type $? \to ?$ is specifically a function that maps boxed values (values cast to ?) to boxed values. This led us to defining a new interpretation of types, which we called the *concrete interpretation of gradual types*, which properly models the behaviour of the values of the cast language. Using this interpretation, we defined a denotational semantics of the cast language, which we proved to be type sound and computationally sound.

13.1. Related work

As we have discussed throughout this discussion, one of the main goals of this work was to bridge the gap between the interpretation of types proposed by Frisch et al. [27] and the interpretation of the values of a language. This led us to defining our denotational semantics in terms of sets of elements of the interpretation domain used to define the interpretation of types, so as to ensure types and terms are interpreted into the same domain.

To our knowledge, there is no existing work that attempts to define the denotational semantics of a language in terms of sets of elements of an interpretation domain. While Frisch et al. [27] define an interpretation of types as sets of values and prove that it coincides with their interpretation of types on the domain $\mathscr{D}$, they never directly propose an interpretation of expressions,

or even values, on $\mathscr{D}$.

We are also unaware of much existing work on the denotational semantics of gradual typing from a set-theoretic perspective. There are some similarities with the work of New et al. [52], in which they interpret the dynamic type ? as the sum of the types $? \times ?$, $? \to ?$, and a type corresponding to constants cast to ?. We can obtain a similar result from our set-theoretic interpretation of gradual types, where the sum type is simply interpreted as a union type: every element belonging to the interpretation of ? is either a function, a pair, or a constant tagged with ?. Our denotational interpretation of casts also bears some resemblance to the *direct cast translation* of New and Ahmed [51], in which casts are interpreted as functions in a metalanguage. For example, similarly to our semantics, they interpret a cast going from an arrow type $\tau_1 \to \tau_2$ to the dynamic type as a function that first casts its argument from $\tau_1 \to \tau_2$ to $? \to ?$, and then *injects* it into ?. Nevertheless, none of the aforementioned works propose a set-theoretic interpretation of gradual types or of a gradually-typed language.

13.2. Future work

In this section, we present some interesting directions for future work, and detail some aspects whose discussion we postponed until now.

13.2.1. About the meaning of Ω

Throughout this part of the manuscript, we consistently referred to Ω as being a type error, or symbolizing a stuck reduction. However, we never truly formalized this meaning via a theorem, and all the results we presented (particularly the type soundness theorems) would still hold if we simply ignored Ω in our semantics.

We believe that it is possible to prove a result which we call the Ω-adequacy, which states that the semantics of an expression contains Ω if and only if this expression reduces to a stuck term. Formally, we conjecture that we can prove the following result for the semantics of λ_{F}, provided we introduce a few changes to the interpretation domain $\mathscr{D}^{\mathrm{F}}$ and to the operational semantics of λ_{F}, which we detail in this part of the discussion.

> **Conjecture 13.1** (Ω-adequacy)**.** *For every closed term* $\mathbf{e} \in \mathsf{Terms}$, $\Omega \in [\![\mathbf{e}]\!]^F_\emptyset \iff \mathbf{e} \rightsquigarrow^* \mathbf{e}'$ *and* $\mathbf{e}'$ *is stuck.*

Notice that we only state this conjecture for closed terms. Generalizing it to arbitrary terms may be possible, although this would require to define head normal forms and would make the proofs more difficult, for a minimal gain.

To prove this property, we first need to introduce a *typed* β-reduction rule. Essentially, this means that an application is only reduced if the type of the argument is compatible with the type of the annotation of the λ-abstraction. Formally, this means replacing the standard β-reduction rule with the following:

$$[\mathrm{R}^{\mathrm{F}}_{\mathrm{app}}] \qquad (\lambda x{:}t.\, \mathbf{e})\, \mathbf{v} \rightsquigarrow \mathbf{e}\,[\mathbf{v}/x] \qquad \text{if } \vdash \mathbf{v} : t$$

This change is necessary since, without it, an ill-typed application can still reduce to a value, even though its semantics contains Ω. For example, the application $(\lambda x{:}\mathsf{Int}.\, x)\ \mathtt{true}$ reduces to true using a standard β-reduction, but its denotation according to Definition 10.5 is exactly $\{\Omega\}$. Using a typed β-reduction, this application would be stuck, thus verifying the Ω-adequacy.

This is not sufficient however, as there are still stuck ill-typed terms whose semantics does not contain Ω. As an example, consider the function $\mathtt{f} = \lambda x{:}\mathsf{Int}.\,(\lambda y{:}\mathsf{Int}.\,y)\;2$ which always return 2 for every integer argument, but is written so as to ensure that it can only be given type $\mathsf{Int} \to \mathsf{Int}$ and not $\mathsf{Int} \to 2$. From a denotational perspective, its semantics only contains relations that belong to the type $\mathsf{Int} \to 2$, even though it cannot be given this type statically. As such, if it is passed as argument to the function $\lambda \mathtt{f}{:}\mathsf{Int} \to \mathsf{Nat}.\,\mathtt{f}\;0)$, for example, the resulting application is stuck since $\mathtt{f}$ cannot be given type $\mathsf{Int} \to \mathsf{Nat}$. However, its semantics is exactly $\{2\}$, and does not contain Ω.

We can fix this by drawing inspiration from Chapter 11. The solution is to enforce a correspondence between the type of a value and the type of its denotation, so as to prove a stronger form of the type soundness property for values:

Conjecture 13.2 (Strong type soundness for values)**.** *For every (closed) value* $\mathbf{v} \in \mathsf{Values}$*,* $\vdash \mathbf{v} : t \iff [\![\mathbf{v}]\!]^F_\emptyset \subseteq [\![t]\!]^F$.

This can be done by first annotating λ-abstraction with their full type, and then introducing the elements $\mho_{(.)}$ to the interpretation domain so that two λ-abstractions with the same body and input type but with different output types can be distinguished. For example, the denotation of $\lambda^{\mathsf{Int}\to\mathsf{Int}}x.\,2$ would now contain the relation $\{(\mho_1, -1)\}$ while the denotation of $\lambda^{\mathsf{Int}\to\mathsf{Nat}}x.\,2$ would not.

Having proven this result, the last remaining step is to generalize the computational adequacy of λ_{F} to arbitrary closed terms (rather than limiting it to well-typed terms) whose semantics do not contain Ω. This is fairly straightforward as most of the proofs are still valid, the only major difference being that the strong type soundness for values must be invoked in the proof of Lemma 10.23.

Finally, the Ω-adequacy is obtained as a corollary of the generalized computational adequacy, and its proof highlights the true meaning of the operator $\Omega^{(.)}_{(.)}$. As an example, consider the case of a pair $(\mathbf{e}_1, \mathbf{e}_2)$ whose denotation contains Ω. By inversion of the definition of the semantics and of the operator $\Omega^{(.)}_{(.)}$, we distinguish two cases. The first case is when $\Omega \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_\rho$. By induction, we deduce that $\mathbf{e}_1$ reduces to a stuck term, and according to our reduction strategy (we reduce terms from left to right), the whole pair reduces to a stuck term. The second case is when $\Omega \in [\![\mathbf{e}_2]\!]^{\mathrm{F}}_\rho$, and where $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_\rho$ is non-empty but does not contain Ω. By the generalized computational adequacy property, this ensures that $\mathbf{e}_1$ reduces to a value $\mathbf{v}$. We also obtain that, by induction, $\mathbf{e}_2$ reduces to a stuck term. Therefore, according to our reduction strategy, we obtain that $(\mathbf{e}_1, \mathbf{e}_2) \rightsquigarrow^* (\mathbf{v}, \mathbf{e}_2)$, which in turn reduces to a stuck term.

13.2.2. About the denotational semantics of $\mathbb{C}$Duce

In Chapter 11, we presented a sound semantics for a language mimicking the functional core of $\mathbb{C}$Duce : a λ-calculus supporting typecases, function overloading, and non-determinism. There are two major directions for future work on this semantics.

About its adequacy

The first concerns the adequacy of our semantics, which we left as a conjecture. It should be possible to prove it for the deterministic restriction of our calculus by following the same strategy as in Chapter 10, and reusing the relation defined in Definition 10.21, adding a rule to remove pairs of the form $(\mho_d, \partial)$ from finite relations similarly to what we have done for pairs containing

Ω. The presence of typecases should only add a single case to the proof of the substitution lemma (Lemma 10.24), which should be provable by application of the induction hypothesis.

Proving the adequacy in the presence of non-deterministic expressions is, however, more complex, especially since marks now have to be taken into account when defining the relation. The several lemmas that led to the adequacy theorem also need to be restated. For example, Lemma 10.23 does not hold anymore: if $d\mathscr{R}\mathbf{e}$ and $\mathbf{e} \rightsquigarrow^* \mathbf{v}$, we do not have $d\mathscr{R}\mathbf{v}$, since the marks of a denotation change by reduction. At best, we could state that there exists a d' and a mark $\mathfrak{m}$ such that $d'\mathscr{R}\mathbf{v}$ and $[d']^{\mathfrak{m}} = d$. Whether this new statement is sufficient to prove the adequacy is left as future work.

About the inference of negative arrows

To obtain a type soundness property for our semantics, we slightly modified the type system of ℂDuce by imposing the restriction that λ-abstractions have to be fully annotated with their negation types, forbidding any form of type inference in the type system. In ℂDuce the type of a function can be arbitrarily refined using negative arrow types, so as to ensure that every value belongs to a type or its negation (this is particularly important for the soundness of the reduction of typecases).

Nevertheless, we argued that this change was not overly restrictive, by showing that to every ℂDuce term corresponds a term of our system that behaves identically (by bisimulation), and for which we can compute a denotational semantics. In essence, this term is obtained by first annotating the ℂDuce term by all the negative arrow types that are inferred by the type system, and then adding all the negative arrow types that are necessary to ensure typecases are resolved unambiguously. While the latter step can be done algorithmically, for the sake of simplicity we only presented the former step in a declarative way. Thus, this does not give us a compilation algorithm from ℂDuce to λ_{C}.

However, we believe that it should be possible to adapt the algorithmic type system of ℂDuce to obtain a way to algorithmically annotate λ-abstractions with the negations that are necessary to ensure a program is well-typed. Due to the complexity of ℂDuce's algorithmic type system, we leave this as future work.

13.2.3. About the denotational semantics for gradual typing

In Chapter 12, we presented a set-theoretic interpretation of gradual types featuring full-fledged set-theoretic types, including product types. However, for complexity reasons, in the second part of the chapter, we limited our denotational semantics to a cast calculus with simple types and without products. This gives two immediate ways to continue this work.

About pairs and product types

The first aspect is the addition of pairs and product types to the cast language of Section 12.1. However, this is not a feature that can simply be dropped-in, due to the behaviour of blames. Intuitively, following the idea that a gradually-typed language should fail as little as possible, an expression such as $\pi_1\,(3\langle \mathsf{Int} \Rightarrow_p ?\rangle, 2\langle \mathsf{Int} \Rightarrow_{p'} ?\rangle)\langle ?\times ? \Rightarrow_q \mathsf{Int}\times\mathsf{Bool}\rangle$ should not fail and reduce to 3, even though it is clear that the pair that is being projected is *not* of type $\mathsf{Int} \times \mathsf{Bool}$, since its second component is of type Int. However, since we project this pair to its first component, this error can safely be ignored. This is similar to how we handle cast functions: if we consider

a function `f` that maps, say, 0 to `true` and 1 to 1, then casting this function to $\mathsf{Int} \to \mathsf{Bool}$ only produces an error if it is later applied to 1, but not if it is applied to 0.

From an operational point of view, this is often formalized by seeing cast pairs as values. Here, the pair $(3\langle \mathsf{Int} \Rightarrow_p ? \rangle, 2\langle \mathsf{Int} \Rightarrow_{p'} ? \rangle)\langle ? \times ? \Rightarrow_q \mathsf{Int} \times \mathsf{Bool} \rangle$ is a value, and the cast is only evaluated when one tries to apply a projection to this value. From a denotational perspective, this is more complicated, since we do not have a notion of delayed cast in our interpretation domain. If the denotation of an expression contains the pair $(3^?, 2^?)^?$, and we cast this expression to $\mathsf{Int} \times \mathsf{Bool}$, we *have* to produce a denotation for the cast expression that belongs to $\mathsf{Int} \times \mathsf{Bool}$.

The unsatisfactory solution would be to simply fail in this case, but this would change the operational behaviour of casts.

A different solution would be to add blame as an element of $\mathscr{D}^{\mathrm{G}}$, instead of only restricting it to the output of relations. With such a modification, the above denotation could be cast to $(3^!, \mathsf{blame}\ q)^!$, and then its first projection would produce $3^!$ (ignoring the blame) while its second projection would produce blame q. However, this second solution causes a problem with the set-theoretic interpretation of types. For the type soundness theorem to hold, this means that blame needs to also be added to the interpretation of product types. That is, the interpretation of $\mathsf{Int} \times \mathsf{Bool}$ would now contain the pair $(3^!, \mathsf{blame}\ q)^!$. However, this also holds for the interpretation of $\mathsf{Int} \times \neg\mathsf{Bool}$. As such, the intersection $(\mathsf{Int} \times \mathsf{Bool}) \wedge (\mathsf{Int} \times \neg\mathsf{Bool})$ would not be empty anymore, which completely changes the subtyping relation on static types. In fact, no type would be truly empty anymore, since even $\mathbb{0}$ would contain all blame elements.

It may be possible to adapt the interpretation of subtyping to account for this. However, another possible solution would be to parameterize blame elements with elements of $\mathscr{D}^{\mathrm{G}}$. That is, a blame element would now be denoted $\mathsf{blame}_d\ p$. Using this, we could modify the interpretation of types so that $\mathsf{blame}_d\ p$ belongs to a type τ if and only if d belongs to τ. This would ensure that the interpretation of $\mathbb{0}$ is still empty, and that Bool and $\neg$Bool would not contain the same blame elements (thus making $(\mathsf{Int} \times \mathsf{Bool}) \wedge (\mathsf{Int} \times \neg\mathsf{Bool})$ empty). This seems to be a promising solution which we plan to explore in the future.

About set-theoretic types

Naturally, the second direction for future work on the set-theoretic semantics of cast languages concerns the addition of set-theoretic types. The semantics we presented in Definition 12.19 is, in itself, independent of the syntax of types. However, it relies on the concrete interpretation of types (Definition 12.16) and on the coercion of denotations (Definition 12.18) which both rely on the hypothesis that types do not contain set-theoretic connectives.

The first hurdle that comes from the presence of set-theoretic types is that a value can be both boxed (have type ?) and unboxed (have a precise static type). For example, it is possible to cast 3 to $\mathsf{Int} \wedge ?$, which gives it both type ? and Int. The same goes for functions, which can both have type ? and an arrow type (and thus, can be applied). This raises the question of how to distinguish, from a denotational perspective, a value that has been cast to ? from a value that has been cast to $? \wedge t$ where t is a static type. Necessarily, both must be denoted by elements tagged with ?.

This leads us towards an idea that we have already begun to explore in Chapter 6: that the type of a value can be propagated along its casts, so that a value cast to ? is effectively identical to the same value cast to $? \wedge t$ where t is the "constructor type" of the unboxed value (that is, the type of the constant if the unboxed value is a constant, and $\mathbb{0} \to \mathbb{1}$ if the unboxed value is a function). This also hints at the fact that there is no true distinction between an element tagged

with `?` and an element tagged with `!` in the semantics of a set-theoretic cast language, since the cast of a value to `?` can be performed without changing its semantics.

The second problem comes from the fact that the source and target types of a cast would not necessarily share the same syntactic structure anymore, since precision satisfies the standard distributivity and commutativity rules of set-theoretic connectives. This means that coercing a denotation as in Definition 12.18 cannot be performed syntactically anymore. A solution that comes to mind is to introduce the gradual type operators presented in Chapter 6, and base the coercion operation on these operators.

To summarize, despite its length, this second part just scratches the surface of the study of a set-theoretic semantics for cast languages. The bulk of the work looks to be still ahead, although we already identified and proposed viable solutions to many problems linked to this study.

Conclusion

Conclusion

This thesis started with the goal of defining and studying a language featuring both set-theoretic types and gradual typing. All our work builds on the semantic subtyping approach to set-theoretic types, which we adapted and extended in two different directions.

In the first part of this work, we presented a way to add gradual typing to a set-theoretic type system, based on the idea that the dynamic type behaves as a type variable. However, we realized that this could be generalized to any type system, and this gave us a simple and declarative way of embedding gradual typing in any existing type system. Thus, we also applied our approach to a simple type system with polymorphism but without subtyping.

Our presentation was not limited to the declarative aspects of gradual typing. We also presented two cast languages, one corresponding to our ML-like source language, and the other featuring full-fledged set-theoretic types. We proved the soundness of these languages, and showed that our novel approach to gradual typing unearths a new meaning for blame.

The second part of this thesis focused on the denotational semantics associated with semantic subtyping. We started by defining a denotational semantics for a simple calculus, in which terms and types are interpreted into the same domain. While this was not immediately possible with the interpretation domain we initially considered (which directly came from the semantic subtyping approach), we showed that a very simply modification of this domain made it possible, while preserving the subtyping relation. We proved that the resulting semantics are computationally sound and adequate, as well as sound with respect to the interpretation of types. We then extended our approach to provide a sound denotational semantics for the functional core of ℂDuce , which required once again a modification of the interpretation domain.

In the end of the second part, we applied the knowledge we acquired during this work to provide a set-theoretic interpretation of gradual types. This interpretation led to many powerful results about set-theoretic gradual types, but also pinpointed some interesting incompatibilities between the subtyping relation used previously and gradual typing, for which we gave a simple fix. We also presented a denotational semantics for a simply-typed cast language, which we proved to be sound. This semantics provides a good basis for future work on a cast language featuring set-theoretic types.

Finally, we went back to the cast language with set-theoretic types we studied in the first part of the manuscript to apply our newly obtained results. This allowed us not only to greatly simplify the semantics of the cast language, but also to simplify the proof of its soundness as well as the definition of the various operators introduced in its semantics.

Future work

We have already presented several directions for future work in the concluding discussions of both parts of this manuscript (Chapter 7 and Chapter 13). We recapitulate the ideas that seem the most important here.

A source language based on semantic relations. In Chapter 6, we introduced semantic versions of gradual subtyping and precision, which feature many interesting and powerful properties. We applied these relations to present a semantics for a cast language with set-theoretic types, but we did not provide an associated source language and compilation algorithm. Since the subtyping relations used in Chapter 5 and 6 on static types are different, the algorithms presented in the former cannot be reused directly and would need to be adapted to use the new semantic relations.

A sound and complete inference algorithm for gradual set-theoretic types. The type inference algorithm presented in Chapter 5 is sound but not complete. While it should be possible to modify the tallying algorithm to obtain its completeness, it may be better to simply focus on defining a new algorithm based on the new semantic relations. By leveraging the strong semantic properties of these relations (particularly Theorem 6.10), it should be possible to obtain a sound and complete algorithm that only requires little to no modification of the tallying algorithm.

Function interfaces and gradual typing. While the language presented in Chapter 6 features full-fledged set-theoretic types, it does not support many of the features that make set-theoretic types interesting in practice. Adding function interfaces and typecases as in Chapter 11 to this language would greatly improve its expressiveness.

A denotational interpretation of type errors. Throughout Part II of this manuscript, we continuously referred to the element Ω as being an error symbolizing a stuck reduction. However, we never truly formalized this meaning. In Chapter 13, we presented a possible course of action to tackle this problem which would be interesting to follow.

A denotational semantics for set-theoretic gradual typing. In Chapter 12, we limited our denotational semantics to a cast calculus with simple types only, although our interpretation of gradual types supports full-fledged set-theoretic types. Extending our denotational semantics to support set-theoretic types would be a major step forward and may provide us with new insights into the behaviour of gradual set-theoretic types.

Appendices

Appendix A.

Additional proofs and definitions

A.1. A declarative approach to gradual typing

A.1.1. Gradual typing for Hindley-Milner systems

This subsection of the appendix contains the full formal definitions of the three steps composing the type inference algorithm described in Chapter 4.

$$
\begin{array}{rll}
\langle\!\langle x\colon t\rangle\!\rangle &= \exists\alpha.\ (x \mathrel{\dot\preccurlyeq} \alpha) \wedge (\alpha \mathrel{\dot\leq} t) & \alpha \sharp t \\
\langle\!\langle c\colon t\rangle\!\rangle &= (b_c \mathrel{\dot\leq} t) & \\
\langle\!\langle (\lambda x.\, e)\colon t\rangle\!\rangle &= \exists\alpha_1,\alpha_2.\ (\mathsf{def}\ x\colon \alpha_1\ \mathsf{in}\ \langle\!\langle e\colon \alpha_2\rangle\!\rangle) \wedge (\alpha_1 \mathrel{\dot\preccurlyeq} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot\leq} t) & \alpha_1, \alpha_2 \sharp t, e \\
\langle\!\langle (\lambda x\colon \tau.\, e)\colon t\rangle\!\rangle &= \exists\alpha_1,\alpha_2.\ (\mathsf{def}\ x\colon \tau\ \mathsf{in}\ \langle\!\langle e\colon \alpha_2\rangle\!\rangle) \wedge (\tau \mathrel{\dot\preccurlyeq} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot\leq} t) & \alpha_1, \alpha_2 \sharp t, \tau, e \\
\langle\!\langle e_1\ e_2\colon t\rangle\!\rangle &= \exists\alpha.\ \langle\!\langle e_1\colon \alpha \to t\rangle\!\rangle \wedge \langle\!\langle e_2\colon \alpha\rangle\!\rangle & \alpha \sharp t, e_1, e_2 \\
\langle\!\langle (e_1, e_2)\colon t\rangle\!\rangle &= \exists\alpha_1,\alpha_2.\ \langle\!\langle e_1\colon \alpha_1\rangle\!\rangle \wedge \langle\!\langle e_2\colon \alpha_2\rangle\!\rangle \wedge (\alpha_1 \times \alpha_2 \mathrel{\dot\leq} t) & \alpha_1, \alpha_2 \sharp t, e_1, e_2 \\
\langle\!\langle \pi_i\ e\colon t\rangle\!\rangle &= \exists\alpha_1,\alpha_2.\ \langle\!\langle e\colon \alpha_1 \times \alpha_2\rangle\!\rangle \wedge (\alpha_i \mathrel{\dot\leq} t) & \alpha_1, \alpha_2 \sharp t, e \\
\langle\!\langle \mathsf{let}\ \vec\alpha\ x = e_1\ \mathsf{in}\ e_2\colon t\rangle\!\rangle &= \mathsf{let}\ x\colon \forall\vec\alpha; \alpha[\langle\!\langle e_1\colon \alpha\rangle\!\rangle]^{\mathsf{vars}(e_1)\setminus\vec\alpha}.\ \alpha\ \mathsf{in}\ \langle\!\langle e_2\colon t\rangle\!\rangle & \alpha \sharp \vec\alpha, e_1
\end{array}
$$

FIGURE A.1. Constraint generation

$$
\frac{}{\Gamma;\Delta \vdash (t_1 \mathrel{\dot\leq} t_2) \rightsquigarrow \{t_1 \mathrel{\dot\leq} t_2\} \mid \emptyset}
\qquad
\frac{}{\Gamma;\Delta \vdash (\tau \mathrel{\dot\preccurlyeq} \alpha) \rightsquigarrow \{\tau \mathrel{\dot\preccurlyeq} \alpha\} \mid \emptyset}
$$

$$
\frac{\Gamma(x) = \forall\vec\alpha.\,\tau \quad \vec\beta \sharp \Gamma}{\Gamma;\Delta \vdash (x \mathrel{\dot\preccurlyeq} \alpha) \rightsquigarrow \{\tau\{\vec\alpha := \vec\beta\} \mathrel{\dot\preccurlyeq} \alpha\} \mid \vec\beta}
\qquad
\frac{(\Gamma, x\colon \tau);\Delta \vdash C \rightsquigarrow D \mid \overline\alpha}{\Gamma;\Delta \vdash \mathsf{def}\ x\colon \tau\ \mathsf{in}\ C \rightsquigarrow D \mid \overline\alpha}
$$

$$
\frac{\Gamma;\Delta \vdash C \rightsquigarrow D \mid \overline\alpha}{\Gamma;\Delta \vdash (\exists\vec\alpha.\, C) \rightsquigarrow D \mid \overline\alpha \cup \vec\alpha}\ \vec\alpha \sharp \Gamma, \overline\alpha
\qquad
\frac{\Gamma;\Delta \vdash C_1 \rightsquigarrow D_1 \mid \overline\alpha_1 \qquad \Gamma;\Delta \vdash C_2 \rightsquigarrow D_2 \mid \overline\alpha_2}{\Gamma;\Delta \vdash C_1 \wedge C_2 \rightsquigarrow D_1 \cup D_2 \mid \overline\alpha_1 \cup \overline\alpha_2}\ \overline\alpha_1 \sharp \overline\alpha_2
$$

$$
\frac{\begin{array}{c}\Gamma;\Delta \cup \vec\alpha \vdash C_1 \rightsquigarrow D_1 \mid \overline\alpha_1 \\ (\Gamma, x\colon \forall\vec\alpha,\vec\beta.\,\alpha\theta_1);\Delta \vdash C_2 \rightsquigarrow D_2 \mid \overline\alpha_2\end{array}}{\Gamma;\Delta \vdash \mathsf{let}\ x\colon \forall\vec\alpha;\alpha[C_1]^{\vec\alpha'}.\,\alpha\ \mathsf{in}\ C_2 \rightsquigarrow D_2 \cup \mathsf{equiv}(\theta_1, D_1) \mid \overline\alpha}
\begin{array}{l}
\theta_1 \in \mathsf{solve}_{\Delta\cup\vec\alpha}(D_1) \\
\vec\alpha \sharp \Gamma\theta_1 \\
\vec\beta = \mathsf{vars}(\alpha\theta_1) \setminus (\mathsf{vars}(\Gamma\theta_1) \cup \vec\alpha \cup \vec\alpha') \\
\sharp\left\{\{\alpha\}, \vec\alpha, \overline\alpha_1, \overline\alpha_2, (\mathsf{vars}(\theta_1) \setminus \mathsf{vars}(D_1))\right\} \\
\alpha, \vec\alpha \sharp \Gamma, \Delta \\
\overline\alpha = \{\alpha\} \cup \vec\alpha \cup \overline\alpha_1 \cup \overline\alpha_2 \cup (\mathsf{vars}(\theta_1) \setminus \mathsf{vars}(D_1))
\end{array}
$$

$$
\begin{array}{rl}
\text{where} \qquad \mathsf{equiv}(\theta, D) \stackrel{\text{def}}{=} & \left\{\, (\alpha \mathrel{\dot\preccurlyeq} \alpha) \;\middle|\; \alpha \in \mathsf{vars}_{\dot\preccurlyeq}(D) \cup \mathsf{vars}(D)\theta \,\right\} \\
\cup & \bigcup_{\alpha\in\mathsf{dom}(\theta), \alpha\theta \text{ static}} \{(\alpha \mathrel{\dot\leq} \alpha\theta), (\alpha\theta \mathrel{\dot\leq} \alpha)\}
\end{array}
$$

FIGURE A.2. Constraint simplification

Figure A.1 defines a function $\langle\!\langle (\cdot) \colon (\cdot) \rangle\!\rangle$ such that, for every expression e and every static type t, $\langle\!\langle e \colon t \rangle\!\rangle$ is a structured constraint that expresses the conditions that must hold for e to have type $t\theta$ for some substitution θ.

Figure A.2 presents the constraint simplification rules in a form in which we track explicitly the variables we introduce and state precise freshness conditions. In a derivation $\Gamma; \Delta \vdash C \rightsquigarrow D \mid \overline{\alpha}$, the set $\overline{\alpha}$ is the set of fresh variables introduced by simplification.

$$
\begin{aligned}
(\!|x|\!)^{\mathscr{D}}_{\theta} &= x\ [\vec{\beta}\theta]\langle \tau\{\vec{\alpha} \coloneqq \vec{\beta}\}\theta \overset{\ell}{\Rightarrow} \alpha\theta \rangle \\
&\quad \text{with } \ell \text{ fresh} \\
&\quad \text{where } \mathscr{D} = \frac{}{\Gamma; \Delta \vdash \langle\!\langle x \colon t \rangle\!\rangle \rightsquigarrow \{(\tau\{\vec{\alpha} \coloneqq \vec{\beta}\} \mathrel{\dot{\preccurlyeq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}} \\
(\!|c|\!)^{\mathscr{D}}_{\theta} &= c \\
(\!|\lambda x.\, e|\!)^{\mathscr{D}}_{\theta} &= \lambda^{(\alpha_1 \to \alpha_2)\theta} x.\ (\!|e|\!)^{\mathscr{D}'}_{\theta} \\
&\quad \text{where } \mathscr{D} = \frac{\mathscr{D}' :: (\Gamma, x \colon \alpha_1); \Delta \vdash \langle\!\langle e \colon \alpha_2 \rangle\!\rangle \rightsquigarrow D'}{\Gamma; \Delta \vdash \langle\!\langle (\lambda x.\, e) \colon t \rangle\!\rangle \rightsquigarrow D' \cup \{(\alpha_1 \mathrel{\dot{\preccurlyeq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}} \\
(\!|\lambda x \colon \tau.\, e|\!)^{\mathscr{D}}_{\theta} &= (\lambda^{(\tau \to \alpha_2)\theta} x.\ (\!|e|\!)^{\mathscr{D}'}_{\theta})\langle (\tau \to \alpha_2)\theta \overset{\ell}{\Rightarrow} (\alpha_1 \to \alpha_2)\theta \rangle \\
&\quad \text{with } \ell \text{ fresh} \\
&\quad \text{where } \mathscr{D} = \frac{\mathscr{D}' :: (\Gamma, x \colon \tau); \Delta \vdash \langle\!\langle e \colon \alpha_2 \rangle\!\rangle \rightsquigarrow D'}{\Gamma; \Delta \vdash \langle\!\langle (\lambda x \colon \tau.\, e) \colon t \rangle\!\rangle \rightsquigarrow D' \cup \{(\tau \mathrel{\dot{\preccurlyeq}} \alpha_1), (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\}} \\
(\!|e_1\ e_2|\!)^{\mathscr{D}}_{\theta} &= (\!|e_1|\!)^{\mathscr{D}_1}_{\theta}\ (\!|e_2|\!)^{\mathscr{D}_2}_{\theta} \\
&\quad \text{where } \mathscr{D} = \frac{\mathscr{D}_1 :: \Gamma; \Delta \vdash \langle\!\langle e_1 \colon \alpha \to t \rangle\!\rangle \rightsquigarrow D_1 \qquad \mathscr{D}_2 :: \Gamma; \Delta \vdash \langle\!\langle e_2 \colon \alpha \rangle\!\rangle \rightsquigarrow D_2}{\Gamma; \Delta \vdash \langle\!\langle e_1\ e_2 \colon t \rangle\!\rangle \rightsquigarrow D_1 \cup D_2} \\
(\!|(e_1, e_2)|\!)^{\mathscr{D}}_{\theta} &= ((\!|e_1|\!)^{\mathscr{D}_1}_{\theta}, (\!|e_2|\!)^{\mathscr{D}_2}_{\theta}) \\
&\quad \text{where } \mathscr{D} = \frac{\mathscr{D}_1 :: \Gamma; \Delta \vdash \langle\!\langle e_1 \colon \alpha_1 \rangle\!\rangle \rightsquigarrow D_1 \qquad \mathscr{D}_2 :: \Gamma; \Delta \vdash \langle\!\langle e_2 \colon \alpha_2 \rangle\!\rangle \rightsquigarrow D_2}{\Gamma; \Delta \vdash \langle\!\langle (e_1, e_2) \colon t \rangle\!\rangle \rightsquigarrow D_1 \cup D_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t\}} \\
(\!|\pi_i\ e|\!)^{\mathscr{D}}_{\theta} &= \pi_i\ (\!|e|\!)^{\mathscr{D}'}_{\theta} \\
&\quad \text{where } \mathscr{D} = \frac{\mathscr{D}' :: \Gamma; \Delta \vdash \langle\!\langle e \colon \alpha_1 \times \alpha_2 \rangle\!\rangle \rightsquigarrow D'}{\Gamma; \Delta \vdash \langle\!\langle \pi_i\ e \colon t \rangle\!\rangle \rightsquigarrow D' \cup \{\alpha_i \mathrel{\dot{\leq}} t\}} \\
(\!|\mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2|\!)^{\mathscr{D}}_{\theta} &= \mathsf{let}\ x = \Lambda\vec{\alpha}_1, \vec{\beta}_1.\ (\!|e_1|\!)^{\mathscr{D}_1}_{\theta_1}\rho\theta\ \mathsf{in}\ (\!|e_2|\!)^{\mathscr{D}_2}_{\theta} \\
&\quad \text{where } \mathscr{D} = \frac{\mathscr{D}_1 :: \Gamma; \Delta \cup \vec{\alpha} \vdash C_1 \rightsquigarrow D_1 \qquad \mathscr{D}_2 :: (\Gamma, x \colon \forall \vec{\alpha}, \vec{\beta}.\, \alpha\theta_1); \Delta \vdash C_2 \rightsquigarrow D_2}{\Gamma; \Delta \vdash \langle\!\langle \mathsf{let}\ \vec{\alpha}\ x = e_1\ \mathsf{in}\ e_2 \colon t \rangle\!\rangle \rightsquigarrow D_2 \cup \mathsf{equiv}(\theta_1, D_1)} \\
&\quad \text{and} \qquad \theta_1 \in \mathsf{solve}_{\Delta \cup \vec{\alpha}}(D_1) \qquad \vec{\alpha}_1, \vec{\beta}_1 \text{ fresh} \qquad \rho = \{\vec{\alpha} \coloneqq \vec{\alpha}_1\} \cup \{\vec{\beta} \coloneqq \vec{\beta}_1\}
\end{aligned}
$$

FIGURE A.3. Algorithmic compilation

Figure A.3 defines the compilation algorithm that, given an expression e, a derivation $\mathscr{D}$ of $\Gamma; \Delta \vdash \langle\!\langle e \colon t \rangle\!\rangle \rightsquigarrow D$, and a substitution θ such that $\theta \Vdash_{\mathsf{vars}(e)} D$, produces a cast language expression $(\!|e|\!)^{\mathscr{D}}_{\theta}$. It is defined by induction on e. For each case, we deconstruct the derivation $\mathscr{D}$ to obtain the sub-derivations used to compile the sub-expressions of e; we write the derivation in a compressed form where we collapse applications of the rules for definition, existential, and conjunctive constraints.

A.1.2. Gradual typing with set-theoretic types

Subtyping

Lemma A.1. *For every type frame T such that* $\mathsf{vars}(T) = \{A_i \mid i \in I\}$*, there exists a type frame T' such that the four sets*

$$\mathsf{vars}_{+\mathsf{cov}}(T') \subseteq \{A_i^{+\wedge} \mid i \in I\} \qquad \mathsf{vars}_{+\mathsf{con}}(T') \subseteq \{A_i^{+\vee} \mid i \in I\}$$

$$\mathsf{vars}_{-\mathsf{cov}}(T') \subseteq \{A_i^{-\wedge} \mid i \in I\} \qquad \mathsf{vars}_{-\mathsf{con}}(T') \subseteq \{A_i^{-\vee} \mid i \in I\}$$

are pairwise disjoint and such that

$$T = T'\,[A_i/A_i^{+\wedge}]_{i\in I}\,[A_i/A_i^{+\vee}]_{i\in I}\,[A_i/A_i^{-\wedge}]_{i\in I}\,[A_i/A_i^{-\vee}]_{i\in I}$$

Proof. Clearly, T' is definable as a tree: it is the tree that coincides with T except on variables, and that, where T has a variable A_i, has one of $A_i^{+\wedge}$, $A_i^{+\vee}$, $A_i^{-\wedge}$, or $A_i^{-\vee}$ depending on the position of that occurrence of A_i. The tree T' is also clearly contractive and the sets of variables in different positions are disjoint.

For T' to be a type frame, it must also be regular. Since T is regular, it can be described by a finite system of equations

$$\begin{cases} x_1 = \overline{T}_1 \\ \quad\vdots \\ x_n = \overline{T}_n \end{cases}$$

such that every $\overline{T}_i$ is an inductively generated term of the grammar

$$\overline{T} ::= x \mid X \mid \alpha \mid b \mid \overline{T} \times \overline{T} \mid \overline{T} \to \overline{T} \mid \overline{T} \vee \overline{T} \mid \neg\overline{T} \mid \mathbb{0}$$

(x serves as a recursion variable) and that (reading the equations as a tree) $T = x_1$.

Then, T' can be defined as $x_1^{+\wedge}$ where

$$\begin{cases} x_1^{+\wedge} = f^{+\wedge}(\overline{T}_1) \\ x_1^{+\vee} = f^{+\vee}(\overline{T}_1) \\ x_1^{-\wedge} = f^{-\wedge}(\overline{T}_1) \\ x_1^{-\vee} = f^{-\vee}(\overline{T}_1) \\ \quad\vdots \\ x_n^{-\vee} = f^{-\vee}(\overline{T}_n) \end{cases}$$

and where (defining $\overline{+} = -$, $\overline{=} = +$, $\overline{\wedge} = \vee$, and $\overline{\vee} = \wedge$), $f^{pv}(\overline{T})$ is defined inductively as:

$$f^{pv}(x) = x^{pv} \qquad f^{pv}(X) = X^{pv} \qquad f^{pv}(\alpha) = \alpha^{pv}$$

$$f^{pv}(b) = b \qquad f^{pv}(\overline{T_1} \times \overline{T_2}) = f^{pv}(\overline{T_1}) \times f^{pv}(\overline{T_2}) \qquad f^{pv}(\overline{T_1} \to \overline{T_2}) = f^{p\bar{v}}(\overline{T_1}) \to f^{pv}(\overline{T_2})$$

$$f^{pv}(\overline{T_1} \vee \overline{T_2}) = f^{pv}(\overline{T_1}) \vee f^{pv}(\overline{T_2}) \qquad f^{pv}(\neg\overline{T'}) = \neg f^{\overline{p}\overline{v}}(T') \qquad f^{pv}(\mathbb{0}) = \mathbb{0}$$

At most $4n$ equations are needed to define T' (they could be less, since some x_i^{pv} could be unreachable from $x_1^{+\wedge}$). Therefore, T' is regular. □

Lemma A.2. *For every type frame* $T \not\leq \mathbb{0}$*, if* $\{X, Y\} \sharp \mathsf{vars}^X_-(T)$ *or* $\{X, Y\} \sharp \mathsf{vars}^X_+(T)$*, then* $T\,[X/Y] \not\leq \mathbb{0}$*.*

Proof. Let X, Y be two frame variables. We first give some auxiliary definitions.

Let σ range over the two symbols $\boxplus$ and $\boxminus$. We define $\overline{\sigma}$ as follows: $\overline{\boxplus} \stackrel{\text{def}}{=} \boxminus$ and $\overline{\boxminus} \stackrel{\text{def}}{=} \boxplus$.

Given a type frame T', we write $T' \vDash \boxplus$ if $\{X, Y\} \sharp \mathsf{vars}^X_-(T)$ and $T' \vDash \boxminus$ if $\{X, Y\} \sharp \mathsf{vars}^X_+(T)$.

Note that, for all T', T_1, and T_2, we have:

$$\begin{aligned}
(\neg T' \vDash \sigma) &\implies (T' \vDash \overline{\sigma}) \\
(T_1 \vee T_2 \vDash \sigma) &\implies (T_1 \vDash \sigma) \wedge (T_2 \vDash \sigma) \\
(T_1 \times T_2 \vDash \sigma) &\implies (T_1 \vDash \sigma) \wedge (T_2 \vDash \sigma) \\
(T_1 \to T_2 \vDash \sigma) &\implies (T_1 \vDash \sigma) \wedge (T_2 \vDash \sigma)
\end{aligned}$$

We define a function F^σ on domain element tags (finite sets of variables) as:

$$F^{\boxplus}(L) = \begin{cases} L \cup \{X, Y\} & \text{if } X \in L \text{ or } Y \in L \\ L & \text{otherwise} \end{cases} \qquad F^{\boxminus}(L) = \begin{cases} L \setminus \{X, Y\} & \text{if } X \notin L \text{ or } Y \notin L \\ L & \text{otherwise} \end{cases}$$

We also define F on domain elements as follows:

$$\begin{aligned}
F^\sigma(c^L) &= c^{F^\sigma(L)} \\
F^\sigma((d_1, d_2)^L) &= (F^\sigma(d_1), F^\sigma(d_2))^{F^\sigma(L)} \\
F^\sigma(\{(d_1, d'_1), \ldots, (d_n, d'_n)\}^L) &= \{(F^{\overline{\sigma}}(d_1), F^\sigma(d'_1)), \ldots, (F^{\overline{\sigma}}(d_n), F^\sigma(d'_n))\}^{F^\sigma(L)} \\
F^\sigma(\Omega) &= \Omega
\end{aligned}$$

We must show:

$$\left.\begin{array}{l} T \not\leq \mathbb{0} \\ \text{either } \{X, Y\} \sharp \mathsf{vars}^X_-(T) \text{ or } \{X, Y\} \sharp \mathsf{vars}^X_+(T) \end{array}\right\} \implies T\,[X/Y] \not\leq \mathbb{0}$$

This can be restated as:

$$\left.\begin{array}{l} \exists d \in \mathscr{D}.\ (d : T) \\ \exists \sigma.\ T \vDash \sigma \end{array}\right\} \implies \exists d' \in \mathscr{D}.\ (d' : T\,[X/Y])$$

We prove the following, stronger claim:

$$\forall d, T, \sigma. \quad T \vDash \sigma \implies \begin{cases} (d : T) \implies (F^\sigma(d) : T\,[X/Y]) \\ \neg(d : T) \implies \neg(F^{\overline{\sigma}}(d) : T\,[X/Y]) \end{cases}$$

by induction on the pair (d, T), ordered lexicographically. For a given d, T, and σ, we assume $T \vDash \sigma$ and proceed by case analysis on T and d.

Let $\theta = [X/Y]$.

$\underline{T = \alpha.}$

Since $\alpha\theta = \alpha$, we must show

$$(d : \alpha) \implies (F^{\sigma}(d) : \alpha) \qquad \neg(d : \alpha) \implies \neg(F^{\overline{\sigma}}(d) : \alpha)\,.$$

If $(d : \alpha)$, then $\alpha \in \mathrm{tags}(d)$ and also $\alpha \in \mathrm{tags}(F^{\sigma}(d))$. Likewise, if $d \notin [\![\alpha]\!]$, then $\alpha \notin \mathrm{tags}(d)$ and also $\alpha \notin \mathrm{tags}(F^{\overline{\sigma}}(d))$.

$\underline{T = Z, \text{ with } Z \neq X \text{ and } Z \neq Y.}$

Like the previous case.

$\underline{T = X.}$

Since $X \in \mathrm{vars}_{+}^{X}(X)$, we have $\sigma = \boxplus$.

We must show

$$(d : X) \implies (F^{\boxplus}(d) : X) \qquad \neg(d : X) \implies \neg(F^{\boxminus}(d) : X)\,.$$

If $(d : X)$, then $X \in \mathrm{tags}(d)$ and $X \in \mathrm{tags}(F^{\boxplus}(d))$. If $\neg(d : X)$, then $X \notin \mathrm{tags}(d)$ and $X \notin \mathrm{tags}(F^{\boxminus}(d))$.

$\underline{T = Y.}$

Since $Y \in \mathrm{vars}_{+}^{X}(Y)$, we have $\sigma = \boxplus$.

We must show

$$(d : Y) \implies (F^{\boxplus}(d) : X) \qquad \neg(d : Y) \implies \neg(F^{\boxminus}(d) : X)\,.$$

If $(d : Y)$, then $Y \in \mathrm{tags}(d)$ and $X \in \mathrm{tags}(F^{\boxplus}(d))$. If $\neg(d : Y)$, then $Y \notin \mathrm{tags}(d)$ and then $X \notin \mathrm{tags}(F^{\boxminus}(d))$.

$\underline{T = b.}$

Since $b\theta = b$, we must show

$$(d : b) \implies (F^{\sigma}(d) : b) \qquad \neg(d : b) \implies (F^{\overline{\sigma}}(d) : b)\,.$$

If $(d : b)$, then $d = c^{L}$ with $c \in \mathbb{B}b$. Then, $F^{\sigma}(d) = c^{F^{\sigma}(L)}$ and $(F^{\sigma}(d) : b)$.

If $\neg(d : b)$ and d is of the form c^{L}, then $c \notin \mathbb{B}b$: then, $F^{\overline{\sigma}}(d) \notin [\![b]\!]$. If d is not of the form c^{L}, then $F^{\overline{\sigma}}(d)$ is not either and we have $F^{\overline{\sigma}}(d) \notin [\![b]\!]$.

$\underline{T = T_1 \times T_2.}$

Since $T \vDash \sigma$, we have $T_1 \vDash \sigma$ and $T_2 \vDash \sigma$.

We must show

$$\begin{aligned}(d : T_1 \times T_2) &\implies (F^{\sigma}(d) : T_1\theta \times T_2\theta)\\ \neg(d : T_1 \times T_2) &\implies \neg(F^{\overline{\sigma}}(d) : T_1\theta \times T_2\theta)\,.\end{aligned}$$

If $(d : T_1 \times T_2)$, then d is of the form $(d_1, d_2)^{L}$ and, for both i, $(d_i : T_1)$. We have $F^{\sigma}(d) = (F^{\sigma}(d_1), F^{\sigma}(d_2))^{F^{\sigma}(L)}$. By IH, $(d_1 : T_1)$ implies $(F^{\sigma}(d_1) : T_1\theta)$; likewise for d_2. Therefore, $(F^{\sigma}(d) : T_1\theta \times T_2\theta)$.

If $\neg(d : T_1 \times T_2)$ and $d = (d_1, d_2)^L$, then either $\neg(d_1 : T_1)$ or $\neg(d_2 : T_2)$. Then, by IH, either $\neg(F^{\overline{\sigma}}(d_1) : T_1\theta)$ or $\neg(F^{\overline{\sigma}}(d_2) : T_2\theta)$. Therefore, $\neg(F^{\overline{\sigma}}(d) : T_1\theta \times T_2\theta)$. If d is of another form, then the result is immediate.

$\underline{T = T_1 \to T_2.}$

Since $T \vDash \sigma$, we have $T_1 \vDash \sigma$ and $T_2 \vDash \sigma$.

We must show

$$(d : T_1 \to T_2) \implies (F^{\sigma}(d) : T_1\theta \to T_2\theta)$$
$$\neg(d : T_1 \to T_2) \implies \neg(F^{\overline{\sigma}}(d) : T_1\theta \to T_2\theta)\ .$$

If $(d : T_1 \to T_2)$, then d is of the form $\{(d_j, d'_j) \mid j \in J\}^L$ and, for all $j \in J$, we have:

$$(d_j : T_1) \implies (d'_j : T_2)\ .$$

We have $F^{\sigma}(d) = \{(F^{\overline{\sigma}}(d_j), F^{\sigma}(d'_j)) \mid j \in J\}^{F^{\sigma}(L)}$.

For every j, by the induction hypothesis applied to T_1 and d_j, and to T_2 and d'_j, we get

$$(d_j : T_1) \implies (F^{\sigma}(d_j) : T_1\theta) \qquad \neg(d_j : T_1) \implies \neg(F^{\overline{\sigma}}(d_j) : T_1\theta)$$
$$(d'_j : T_2) \implies (F^{\sigma}(d'_j) : T_2\theta) \qquad \neg(d'_j : T_2) \implies \neg(F^{\overline{\sigma}}(d'_j) : T_2\theta)\ .$$

We must show, for all $j \in J$:

$$(F^{\overline{\sigma}}(d_j) : T_1\theta) \implies (F^{\sigma}(d'_j) : T_2\theta)$$

which we prove using the induction hypothesis (in particular, using the contrapositive of the second implication derived by induction).

If $\neg(d : T_1 \to T_2)$ and d is of the form $\{(d_j, d'_j) \mid j \in J\}^L$, then there exists a $j_0 \in J$ such that

$$(d_{j_0} : T_1) \qquad \neg(d'_{j_0} \in T_2)\ .$$

We have $F^{\overline{\sigma}}(d) = \{(F^{\sigma}(d_j), F^{\overline{\sigma}}(d'_j)) \mid j \in J\}^{F^{\overline{\sigma}}(L)}$. By IH, we show

$$(F^{\sigma}(d_{j_0}) : T_1\theta) \qquad \neg(F^{\overline{\sigma}}(d_{j_0}) : T_2\theta)\ .$$

If d is of another form, we have the result directly. then we get the result directly.

$\underline{T = T_1 \vee T_2.}$

Since $T \vDash \sigma$, we have $T_1 \vDash \sigma$ and $T_2 \vDash \sigma$.

By the induction hypothesis applied to d and T_i, we get

$$(d : T_i) \implies (F^{\sigma}(d) : T_i\theta) \qquad \neg(d : T_i) \implies \neg(F^{\overline{\sigma}}(d) : T_i\theta)\ .$$

We must show

$$(d : T_1 \vee T_2) \implies (F^{\sigma}(d) : T_1\theta \vee T_2\theta) \qquad \neg(d : T_1 \vee T_2) \implies (F^{\overline{\sigma}}(d) : T_1\theta \vee T_2\theta)\ .$$

To show the first implication, assume $(d : T_1 \vee T_2)$: then either $(d : T_1)$ or $(d : T_2)$; then either $(F^\sigma(d) : T_1\theta)$ or $(F^\sigma(d) : T_2\theta)$; then $(F^\sigma(d) : T_1\theta \vee T_2\theta)$. To show the second, assume $\neg(d : T_1 \vee T_2)$: then $\neg(d : T_1)$ and $\neg(d : T_2)$; then $\neg(F^{\overline{\sigma}}(d) : T_1)$ and $\neg(F^{\overline{\sigma}}(d) : T_2)$; then $\neg(F^{\overline{\sigma}}(d) : T_1 \vee T_2)$.

$\underline{T = \neg T'.}$

Since $T \vDash \sigma$, $T' \vDash \overline{\sigma}$.

By applying the induction hypothesis to d and T', we get

$$(d : T') \implies (F^{\overline{\sigma}}(d) : T'\theta) \qquad \neg(d : T') \implies \neg(F^\sigma(d) : T'\theta)\,.$$

We must show

$$(d : \neg T') \implies (F^\sigma(d) : \neg(T'\theta)) \qquad \neg(d : \neg T') \implies \neg(F^{\overline{\sigma}}(d) : \neg(T'\theta))\,.$$

For the first implication, assume $(d : \neg T')$: then $\neg(d : T')$, $\neg(F^\sigma(d) : T'\theta)$, and $(F^\sigma(d) : \neg(T'\theta))$. For the second, assume $\neg(d : \neg T')$: then $\neg\neg(d : T')$, that is, $(d : T')$; hence $(F^{\overline{\sigma}}(d) : T'\theta)$, and $\neg(F^{\overline{\sigma}}(d) : \neg(T'\theta))$.

$\underline{T = \mathbb{0}.}$

Both implications are trivial. □

Corollary A.3. *For every type frames T_1, T_2 such that $T_1 \leq T_2$, for every variable X such that $X \notin \mathsf{vars}^X_+(T_1) \cap \mathsf{vars}^X_+(T_2)$ and $X \notin \mathsf{vars}^X_-(T_1) \cap \mathsf{vars}^X_-(T_2)$, and for all Y such that $Y \sharp X, T_1, T_2$, it holds that $T_1\,[Y/X] \leq T_2$.*

Proof. If $X \notin \mathsf{vars}^X(T_1)$, the result is immediate because $T_1\,[Y/X] = T_1$. If $X \notin \mathsf{vars}^X(T_2)$, then we have $T_2 = T_2\,[Y/X]$ and the result can be derived by Proposition 5.2. We consider the case $X \in \mathsf{vars}^X(T_1) \cap \mathsf{vars}^X(T_2)$. In this case, we have $X \notin \mathsf{vars}^X_+(T_1) \cap \mathsf{vars}^X_-(T_1)$: otherwise, X could not occur in T_2. Therefore, X occurs only positively or only negatively in T_1.

Given T_1, T_2, X, and Y satisfying

$$X \notin \mathsf{vars}^X_+(T_1) \cap \mathsf{vars}^X_+(T_2) \qquad X \notin \mathsf{vars}^X_-(T_1) \cap \mathsf{vars}^X_-(T_2) \qquad Y \sharp T_1, T_2, X\,,$$

we must show $T_1 \leq T_2 \implies T_1\,[Y/X] \leq T_2$.

We show the contrapositive: $T_1\,[Y/X] \not\leq T_2 \implies T_1 \not\leq T_2$. Assume $T_1\,[Y/X] \not\leq T_2$.

We have $T_1 = T_1\,[Y/X]\,[X/Y]$ and $T_2 = T_2\,[X/Y]$. Let $T = T_1\,[Y/X] \setminus T_2$. We have $T \not\leq \mathbb{0}$ by definition of subtyping.

We show that either $\{X, Y\} \sharp \mathsf{vars}^X_-(T)$ or $\{X, Y\} \sharp \mathsf{vars}^X_+(T)$ holds. Note that

$$\mathsf{vars}^X_+(T) = \mathsf{vars}^X_+(T_1\,[Y/X]) \cup \mathsf{vars}^X_-(T_2) \qquad \mathsf{vars}^X_-(T) = \mathsf{vars}^X_-(T_1\,[Y/X]) \cup \mathsf{vars}^X_+(T_2)\,.$$

If $X \in \mathsf{vars}^X_+(T_1)$, then $X \notin \mathsf{vars}^X_-(T_1)$ and $X \notin \mathsf{vars}^X_+(T_2)$: therefore, $\{X, Y\} \sharp \mathsf{vars}^X_-(T)$. If $X \in \mathsf{vars}^X_-(T_1)$, then $X \notin \mathsf{vars}^X_+(T_1)$ and $X \notin \mathsf{vars}^X_-(T_2)$: therefore, $\{X, Y\} \sharp \mathsf{vars}^X_+(T)$.

By Lemma A.2, we have $T\,[X/Y] \not\leq \mathbb{0}$: that is, $(T_1\,[Y/X] \setminus T_2)\,[X/Y] \not\leq \mathbb{0}$; that is, $T_1\,[Y/X]\,[X/Y] \not\leq T_2\,[X/Y]$, which is $T_1 \not\leq T_2$. □

Lemma A.4. *For every type frame $T \not\leq \mathbb{0}$ and every variables $X, Y \in \mathscr{V}^X$, if $X \notin \mathsf{vars}^X_{\mathsf{even}}(T)$ and $Y \notin \mathsf{vars}^X_{\mathsf{odd}}(T)$ then $T\,[X/Y] \not\leq \mathbb{0}$.*

Proof. We first give some auxiliary definitions.

Let σ range over the two symbols $\triangle$ and $\triangledown$. We define $\overline{\sigma}$ as follows: $\overline{\triangle} \stackrel{\text{def}}{=} \triangledown$ and $\overline{\triangledown} \stackrel{\text{def}}{=} \triangle$.

Given a type frame T', we write $T' \vDash \triangle$ if $X \notin \mathsf{vars}^X_{\mathsf{odd}}(T')$ and $Y \notin \mathsf{vars}^X_{\mathsf{even}}(T')$; we write $T' \vDash \triangledown$ if $X \notin \mathsf{vars}^X_{\mathsf{even}}(T')$ and $Y \notin \mathsf{vars}^X_{\mathsf{odd}}(T')$.

Note that, for all T', T_1, and T_2, we have:

$$\begin{aligned}
(\neg T' \vDash \sigma) &\implies (T' \vDash \sigma) \\
(T_1 \vee T_2 \vDash \sigma) &\implies (T_1 \vDash \sigma) \wedge (T_2 \vDash \sigma) \\
(T_1 \times T_2 \vDash \sigma) &\implies (T_1 \vDash \sigma) \wedge (T_2 \vDash \sigma) \\
(T_1 \to T_2 \vDash \sigma) &\implies (T_1 \vDash \overline{\sigma}) \wedge (T_2 \vDash \sigma)
\end{aligned}$$

We define a function F^σ on domain element tags (finite sets of variables) as:

$$F^{\triangle}(L) = L \qquad F^{\triangledown}(L) = \begin{cases} L \cup \{X\} & \text{if } Y \in L \\ L \setminus \{X\} & \text{if } Y \notin L \end{cases}$$

We also define F on domain elements as follows:

$$\begin{aligned}
F^\sigma(c^L) &= c^{F^\sigma(L)} \\
F^\sigma((d_1, d_2)^L) &= (F^\sigma(d_1), F^\sigma(d_2))^{F^\sigma(L)} \\
F^\sigma(\{(d_1, d_1'), \ldots, (d_n, d_n')\}^L) &= \{(F^{\overline{\sigma}}(d_1), F^\sigma(d_1')), \ldots, (F^{\overline{\sigma}}(d_n), F^\sigma(d_n'))\}^{F^\sigma(L)} \\
F^\sigma(\Omega) &= \Omega
\end{aligned}$$

We must show:

$$\left.\begin{array}{l} T \not\leq \mathbb{0} \\ X \notin \mathsf{vars}^X_{\mathsf{even}}(T) \\ Y \notin \mathsf{vars}^X_{\mathsf{odd}}(T) \end{array}\right\} \implies T\,[X/Y] \not\leq \mathbb{0}$$

This can be restated as:

$$\left.\begin{array}{l} \exists d \in \mathscr{D}.\ (d : T) \\ T \vDash \triangledown \end{array}\right\} \implies \exists d' \in \mathscr{D}.\ (d' : T\,[X/Y])$$

We prove the following, stronger claim:

$$\forall d, T, \sigma. \quad T \vDash \sigma \implies \big((d : T) \iff (F^\sigma(d) : T\,[X/Y])\big)$$

by induction on the pair (d, T), ordered lexicographically. For a given d, T, and σ, we assume $T \vDash \sigma$ and proceed by case analysis on T and d.

Let $\theta = [X/Y]$.

<u>$T = \alpha$.</u>

Note that $\alpha\theta = \alpha$.

$$\begin{aligned}
(d : \alpha) &\iff \alpha \in \mathsf{tags}(d) \\
&\iff \alpha \in \mathsf{tags}(F^\sigma(d)) \quad \text{neither } F^{\triangle} \text{ nor } F^{\triangledown} \text{ affect variables other than } X \\
&\iff (F^\sigma(d) : \alpha)
\end{aligned}$$

$\underline{T = Z, \text{ with } Z \neq X \text{ and } Z \neq Y.}$

Like the previous case.

$\underline{T = X.}$

Note that we must have $T \vDash \triangle$ because $X \in \mathsf{vars}^X_{\mathsf{even}}(X)$ and $X \notin \mathsf{vars}^X_{\mathsf{odd}}(X)$.

Note that $X\theta = X$.

$$\begin{aligned}(d : X) &\iff X \in \mathsf{tags}(d)\\ &\iff X \in \mathsf{tags}(F^{\triangle}(d))\\ &\iff (F^{\triangle}(d) : X)\end{aligned}$$

$\underline{T = Y.}$

Note that we must have $T \vDash \triangledown$ because $Y \in \mathsf{vars}^X_{\mathsf{even}}(Y)$ and $Y \notin \mathsf{vars}^X_{\mathsf{odd}}(Y)$.

Note that $Y\theta = X$.

$$\begin{aligned}(d : Y) &\iff Y \in \mathsf{tags}(d)\\ &\iff X \in \mathsf{tags}(F^{\triangledown}(d))\\ &\iff (F^{\triangledown}(d) : X)\end{aligned}$$

$\underline{T = b.}$

Note that $b\theta = b$.

If $(d : b)$, then d must be of the form c^L with $c \in \mathbb{B}b$. Then, $F^\sigma(d) = c^{F^\sigma(L)}$ and $(F^\sigma(d) : b)$.

If $(F^\sigma(d) : b)$, then $F^\sigma(d)$ must be of the form c^L with $c \in \mathbb{B}b$. Then, $d = c^{L'}$ and $(d : b)$.

$\underline{T = T_1 \times T_2.}$

If $(d : T_1 \times T_2)$, then $d = (d_1, d_2)^L$, $(d_1 : T_1)$, and $(d_2 : T_2)$. We have $F^\sigma(d) = (F^\sigma(d_1), F^\sigma(d_2))^{F^\sigma(L)}$. By IH we have, for $i \in \{1, 2\}$, $(d_i : T_i) \iff (F^\sigma(d_i) : T_i\theta)$; hence, $(F^\sigma(d) : T_1\theta \times T_2\theta)$.

If $(F^\sigma(d) : T_1\theta \times T_2\theta)$, then $F^\sigma(d) = (d_1, d_2)^L$, $(d_1 : T_1\theta)$, and $(d_2 : T_2\theta)$. Then, we have $d = (d_1', d_2')^{L'}$, with $d_1 = F^\sigma(d_1')$ and $d_2 = F^\sigma(d_2')$. By IH we have, for $i \in \{1, 2\}$, $(d_i' : T_i) \iff (d_i : T_i\theta)$; hence, $(d : T_1 \times T_2)$.

$\underline{T = T_1 \to T_2.}$

Note that, since $T \vDash \sigma$, we have $T_1 \vDash \overline{\sigma}$ and $T_2 \vDash \sigma$.

If $(d : T_1 \to T_2)$, then $d = \{(d_j, d_j') \mid j \in J\}^L$ and

$$\forall j \in J.\ (d_j : T_1) \implies (d_j' : T_2)\ .$$

Then, $F^\sigma(d) = \{(F^{\overline{\sigma}}(d_j), F^\sigma(d_j')) \mid j \in J\}^{F^\sigma(L)}$. By IH, for every $j \in J$,

$$(d_j : T_1) \iff (F^{\overline{\sigma}}(d_j) : T_1\theta) \qquad (d_j' : T_2) \iff (F^\sigma(d_j') : T_2\theta)\ .$$

Therefore, we have

$$\forall j \in J.\ (F^{\overline{\sigma}}(d_j) : T_1\theta) \implies (F^{\sigma}(d'_j) : T_2\theta)$$

and hence $(F^{\sigma}(d) : T_1\theta \to T_2\theta)$.

If $(F^{\sigma}(d) : T_1\theta \to T_2\theta)$, then $F^{\sigma}(d) = \{(d_j, d'_j) \mid j \in J\}^L$ and

$$\forall j \in J.\ (d_j : T_1\theta) \implies (d'_j : T_2\theta)\ .$$

Then, $d = \{(\overline{d}_j, \overline{d}'_j) \mid j \in J\}^{L'}$, with, for every $j \in J$, $F^{\overline{\sigma}}(\overline{d}_j) = d_j$ and $F^{\sigma}(\overline{d}'_j) = d'_j$. By IH, for every $j \in J$,

$$(\overline{d}_j : T_1) \iff (d_j : T_1\theta) \qquad (\overline{d}'_j : T_2) \iff (d'_j : T_2\theta)\ .$$

Therefore, we have

$$\forall j \in J.\ (\overline{d}_j : T_1) \implies (\overline{d}'_j : T_2)$$

and hence $(d : T_1 \to T_2)$.

$\underline{T = T_1 \vee T_2.}$

$$\begin{aligned}
(d : T_1 \vee T_2) &\iff (d : T_1) \vee (d : T_2) \\
&\iff (F^{\sigma}(d) : T_1\theta) \vee (F^{\sigma}(d) : T_2\theta) && \text{by IH} \\
&\iff (F^{\sigma}(d) : T_1\theta \vee T_2\theta)
\end{aligned}$$

$\underline{T = \neg T'.}$

$$\begin{aligned}
(d : \neg T') &\iff \neg(d : T') \\
&\iff \neg(F^{\sigma}(d) : T'\theta) && \text{by IH} \\
&\iff (F^{\sigma}(d) : \neg(T'\theta))
\end{aligned}$$

$\underline{T = \mathbb{0}.}$

Trivial, since $(d : \mathbb{0})$ never holds for any d and since $\mathbb{0}\theta = \mathbb{0}$. □

Lemma A.5. *For every type frame T, if $T \leq \mathbb{0}$ then there exists a type frame T' and a substitution $\theta : \mathscr{V}^X \to \mathscr{V}^X$ such that:*

- $T' \leq \mathbb{0}$
- $T = T'\theta$
- $\mathsf{vars}^X_{\mathsf{even}}(T') \cap \mathsf{vars}^X_{\mathsf{odd}}(T') = \emptyset$

Proof. Assume that $\mathsf{vars}^X(T) = \{X_1, \ldots, X_n\}$.

By Corollary 5.9, we can find T' such that $\mathsf{vars}^X_{\mathsf{even}}(T') \subseteq \{X_1, \ldots, X_n\}$ is disjoint from $\mathsf{vars}^X_{\mathsf{odd}}(T') \subseteq \{X'_1, \ldots, X'_n\}$ and that $T = T'\,[X_i/X'_i]_{i=1}^n$.

We must prove $T' \leq \mathbb{0}$. We have $T \leq \mathbb{0}$, which is $T'\,[X_i/X'_i]_{i=1}^n \leq \mathbb{0}$. Therefore, we also have $T'\,[X_i/X'_i]_{i=1}^n\,[X'_i/X_i]_{i=1}^n \leq \mathbb{0}$ (by Proposition 5.2), which is $T'\,[X'_i/X_i]_{i=1}^n \leq \mathbb{0}$.

Let $\vec{X}$ be the vector $X_1 \ldots X_n$ and $\vec{X}'$ be the vector $X'_1 \ldots X'_n$. We have $\vec{X} \sharp \mathsf{vars}^X_{\mathsf{odd}}(T')$ and

$\vec{X}' \sharp \mathsf{vars}^X_{\mathsf{even}}(T')$. We also have $\vec{X} \sharp \vec{Y}$.

By Lemma A.4, we have

$$T' \not\leq \mathbb{0} \implies T'\,[\vec{X}'/\vec{X}] \not\leq \mathbb{0}$$

and, by contrapositive,

$$T'\,[\vec{X}'/\vec{X}] \leq \mathbb{0} \implies T' \leq \mathbb{0}$$

which yields $T' \leq \mathbb{0}$. □

Lemma A.6. *For every gradual types τ_1, τ_2 and every type frames $T_1 \in \star(\tau_1)$, $T_2 \in \star(\tau_2)$, if $\mathsf{vars}^X_{+\mathsf{cov}}(T_1, T_2)$, $\mathsf{vars}^X_{+\mathsf{con}}(T_1, T_2)$, $\mathsf{vars}^X_{-\mathsf{cov}}(T_1, T_2)$, and $\mathsf{vars}^X_{-\mathsf{con}}(T_1, T_2)$ are pairwise disjoint then $T_1 \leq T_2 \implies \tau_1^\bullet \leq \tau_2^\bullet$.*

Proof. We define

$$\begin{aligned}\theta = {} & [X^{+\wedge}/X]_{X \in \mathsf{vars}^X_{+\mathsf{cov}}(T_1,T_2)} \cup [X^{+\vee}/X]_{X \in \mathsf{vars}^X_{+\mathsf{con}}(T_1,T_2)} \\ & \cup [X^{-\wedge}/X]_{X \in \mathsf{vars}^X_{-\mathsf{cov}}(T_1,T_2)} \cup [X^{-\vee}/X]_{X \in \mathsf{vars}^X_{-\mathsf{con}}(T_1,T_2)}\end{aligned}$$

θ is well-defined because the four sets are disjoint. We have $T_1\theta = \tau_1^\bullet$ and $T_2\theta = \tau_2^\bullet$. We have $T_1\theta \leq T_2\theta$ by Proposition 5.2. □

Lemma A.7. *For every gradual types τ_1, τ_2, if $\tau_1 \leq \tau_2$ then $\tau_1^\bullet \leq \tau_2^\bullet$.*

Proof. By definition of $\tau_1 \leq \tau_2$, there exist $T_1 \in \star(\tau_1)$ and $T_2 \in \star(\tau_2)$ such that:

$$\mathsf{vars}^X_+(T_1) \sharp \mathsf{vars}^X_-(T_1) \qquad \mathsf{vars}^X_+(T_2) \sharp \mathsf{vars}^X_-(T_2) \qquad T_1 \leq T_2\,.$$

Let $\vec{X} = \big(\mathsf{vars}^X_+(T_1) \cap \mathsf{vars}^X_-(T_2)\big) \cup \big(\mathsf{vars}^X_-(T_1) \cap \mathsf{vars}^X_+(T_2)\big)$ and let $\vec{Y}$ be a vector of variables outside T_1 and T_2. Since T_1 and T_2 are polarized, we have

$$\forall X \in \vec{X}.\ \begin{cases} X \in \mathsf{vars}^X_+(T_1) \implies X \notin \mathsf{vars}^X_+(T_2) \\ X \in \mathsf{vars}^X_-(T_1) \implies X \notin \mathsf{vars}^X_-(T_2) \end{cases}$$

and we can apply Corollary A.3 to derive $T_1\,[\vec{Y}/\vec{X}] \leq T_2$. We have

$$\mathsf{vars}^X_+(T_1\,[\vec{Y}/\vec{X}], T_2) \sharp \mathsf{vars}^X_-(T_1\,[\vec{Y}/\vec{X}], T_2)\,.$$

We apply Corollary 5.15 to $T_1\,[\vec{Y}/\vec{X}]$ and T_2 to find T_1', T_2', $\vec{X}'$, and $\vec{Y}'$ such that:

$$T_1' \leq T_2' \qquad T_1\,[\vec{Y}/\vec{X}] = T_1'\,[\vec{X}'/\vec{Y}'] \text{ and } T_2 = T_2'\,[\vec{X}'/\vec{Y}'] \qquad \mathsf{vars}^X_{\mathsf{even}}(T_1', T_2') \sharp \mathsf{vars}^X_{\mathsf{odd}}(T_1', T_2').$$

We have

$$\begin{aligned}\tau_1 = T_1^\dagger = (T_1\,[\vec{Y}/\vec{X}])^\dagger = (T_1'\,[\vec{X}'/\vec{Y}'])^\dagger = (T_1')^\dagger \\ \tau_2 = T_2^\dagger = (T_2'\,[\vec{X}'/\vec{Y}'])^\dagger = (T_2')^\dagger\,.\end{aligned}$$

We also have

$$\mathsf{vars}^X_+(T_1', T_2') \sharp \mathsf{vars}^X_-(T_1', T_2') \qquad \mathsf{vars}^X_{\mathsf{even}}(T_1', T_2') \sharp \mathsf{vars}^X_{\mathsf{odd}}(T_1', T_2')$$

and therefore the following four sets are disjoint

$$\mathsf{vars}^X_{+\mathsf{cov}}(T_1', T_2') \qquad \mathsf{vars}^X_{+\mathsf{con}}(T_1', T_2') \qquad \mathsf{vars}^X_{-\mathsf{cov}}(T_1', T_2') \qquad \mathsf{vars}^X_{-\mathsf{con}}(T_1', T_2') \,.$$

Then, by Lemma A.6, we have $\tau_1^\bullet \leq \tau_2^\bullet$. □

Lemma A.8. *For every gradual types τ_1, τ_2, if there exists $T_1 \in \star^{\mathsf{var}}(\tau_1)$ and $T_2 \in \star^{\mathsf{var}}(\tau_2)$ such that $T_1 \leq T_2$, then $\tau_1^\bullet \leq \tau_2^\bullet$.*

Proof. We have

$$T_1^\dagger = \tau_1 \text{ and } T_2^\dagger = \tau_2 \qquad \mathsf{vars}^X_{\mathsf{cov}}(T_1) \sharp \mathsf{vars}^X_{\mathsf{con}}(T_1) \text{ and } \mathsf{vars}^X_{\mathsf{cov}}(T_2) \sharp \mathsf{vars}^X_{\mathsf{con}}(T_2) \qquad T_1 \leq T_2 \,.$$

We apply Corollary 5.15 to T_1 and T_2 to find T_1', T_2', $\vec{X}$, and $\vec{Y}$ such that:

$$T_1' \leq T_2' \qquad T_1 = T_1'\,[\vec{X}/\vec{Y}] \text{ and } T_2 = T_2'\,[\vec{X}/\vec{Y}] \qquad \mathsf{vars}^X_{\mathsf{even}}(T_1', T_2') \sharp \mathsf{vars}^X_{\mathsf{odd}}(T_1', T_2') \,.$$

Since we have

$$\mathsf{vars}^X_{\mathsf{cov}}(T_1') \sharp \mathsf{vars}^X_{\mathsf{con}}(T_1') \text{ and } \mathsf{vars}^X_{\mathsf{cov}}(T_2') \sharp \mathsf{vars}^X_{\mathsf{con}}(T_2') \qquad \mathsf{vars}^X_{\mathsf{even}}(T_1', T_2') \sharp \mathsf{vars}^X_{\mathsf{odd}}(T_1', T_2') \,,$$

we also have

$$\mathsf{vars}^X_{+}(T_1') \sharp \mathsf{vars}^X_{-}(T_1') \text{ and } \mathsf{vars}^X_{+}(T_2') \sharp \mathsf{vars}^X_{-}(T_2') \,.$$

Let $\vec{X}' = \left(\mathsf{vars}^X_{+}(T_1') \cap \mathsf{vars}^X_{-}(T_2')\right) \cup \left(\mathsf{vars}^X_{-}(T_1') \cap \mathsf{vars}^X_{+}(T_2')\right)$ and let $\vec{Y}'$ be a vector of variables outside T_1' and T_2'. We have

$$\forall X \in \vec{X}'. \quad \begin{cases} X \in \mathsf{vars}^X_{+}(T_1') \implies X \notin \mathsf{vars}^X_{+}(T_2') \\ X \in \mathsf{vars}^X_{-}(T_1') \implies X \notin \mathsf{vars}^X_{-}(T_2') \end{cases}$$

and we can apply Corollary A.3 to derive $T_1'\,[\vec{Y}'/\vec{X}'] \leq T_2'$.

We have

$$\tau_1 = T_1^\dagger = (T_1'\,[\vec{X}/\vec{Y}])^\dagger = (T_1')^\dagger = (T_1'\,[\vec{Y}'/\vec{X}'])^\dagger$$
$$\tau_2 = T_2^\dagger = (T_2'\,[\vec{X}/\vec{Y}])^\dagger = (T_2')^\dagger \,.$$

Let $T_1'' = T_1'\,[\vec{Y}'/\vec{X}']$.

We also have

$$\mathsf{vars}^X_{+}(T_1'', T_2') \sharp \mathsf{vars}^X_{-}(T_1'', T_2') \qquad \mathsf{vars}^X_{\mathsf{even}}(T_1'', T_2') \sharp \mathsf{vars}^X_{\mathsf{odd}}(T_1'', T_2')$$

and therefore the following four sets are disjoint

$$\mathsf{vars}^X_{+\mathsf{cov}}(T_1'', T_2') \qquad \mathsf{vars}^X_{+\mathsf{con}}(T_1'', T_2') \qquad \mathsf{vars}^X_{-\mathsf{cov}}(T_1'', T_2') \qquad \mathsf{vars}^X_{-\mathsf{con}}(T_1'', T_2') \,.$$

Then, by Lemma A.6, we have $\tau_1^\bullet \leq \tau_2^\bullet$. □

Proposition A.9. *For every type frame* $T \in \mathsf{TFrames}$*, and every type substitutions* $\theta_1, \theta_2 : \mathscr{V}^\alpha \cup \mathscr{V}^X \to \mathsf{GTypes}$ *satisfying the following two conditions:*

$$\forall A \in \mathsf{vars}_{\mathsf{cov}}(T), A\theta_1 \leq A\theta_2 \qquad \forall A \in \mathsf{vars}_{\mathsf{con}}(T), A\theta_2 \leq A\theta_1$$

then it holds that $T\theta_1 \leq T\theta_2$*.*

Proof. For every substitution θ, and every set $S \in \mathsf{Vars}$, we write $\theta|_S$ for the substitution that verifies $X\theta|_S = X\theta$ if $X \in S$ and $X\theta|_S = X$ otherwise.
We define

$$P(T, \theta_1, \theta_2) \overset{\text{def}}{\iff} \left(\theta_1|_{\mathsf{vars}_{\mathsf{cov}}(T)} \leq \theta_2|_{\mathsf{vars}_{\mathsf{cov}}(T)}\right) \text{ and } \left(\theta_2|_{\mathsf{vars}_{\mathsf{con}}(T)} \leq \theta_1|_{\mathsf{vars}_{\mathsf{con}}(T)}\right)$$

and note that the following hold

$$\begin{aligned}
P(A, \theta_1, \theta_2) &\implies A\theta_1 \leq A\theta_2 \\
P(T_1 \times T_2, \theta_1, \theta_2) &\implies P(T_1, \theta_1, \theta_2) \text{ and } P(T_2, \theta_1, \theta_2) \\
P(T_1 \to T_2, \theta_1, \theta_2) &\implies P(T_1, \theta_2, \theta_1) \text{ and } P(T_2, \theta_1, \theta_2) \\
P(T_1 \vee T_2, \theta_1, \theta_2) &\implies P(T_1, \theta_1, \theta_2) \text{ and } P(T_2, \theta_1, \theta_2) \\
P(\neg T', \theta_1, \theta_2) &\implies P(T', \theta_2, \theta_1)
\end{aligned}$$

We show the following result (which implies the statement)

$$\forall \theta_1, \theta_2, d, T. \quad \left.\begin{array}{l} P(T, \theta_1, \theta_2) \\ (d : T\theta_1) \end{array}\right\} \implies (d : T\theta_2)$$

by induction on (d, T).

$\underline{T = b \text{ or } T = \mathbb{0}.}$ Trivial, since $T\theta_1 = T = T\theta_2$.

$\underline{T = A.}$ We have $A\theta_1 \leq A\theta_2$ and $(d : A\theta_1)$, which implies $(d : A\theta_2)$.

$\underline{T = T_1 \times T_2.}$

We have $T\theta_1 = (T_1\theta_1) \times (T_2\theta_1)$ and $T\theta_2 = (T_1\theta_2) \times (T_2\theta_2)$.

Since $(d : T\theta_1)$, we have $d = (d_1, d_2)$ and $(d_i : T_i\theta_1)$.

Since $P(T_i, \theta_1, \theta_2)$ holds for both i, by IH we have $(d_i : T_i\theta_2)$. Then, $(d : T\theta_2)$.

$\underline{T = T_1 \to T_2.}$

We have $T\theta_1 = (T_1\theta_1) \to (T_2\theta_1)$ and $T\theta_2 = (T_1\theta_2) \to (T_2\theta_2)$.

Since $(d : T\theta_1)$, we have $d = \{(d_i, d_i') \mid i \in I\}$ and $\forall i \in I.\ (d_i : T_1\theta_1) \implies (d_i' : T_2\theta_1)$.

We have $P(T_1, \theta_2, \theta_1)$ and $P(T_2, \theta_1, \theta_2)$.

For every d_i such that $(d_i : T_1\theta_2)$, by IH we have $(d_i : T_1\theta_1)$, therefore $(d_i' : T_2\theta_1)$, and, by IH, $(d_i' : T_2\theta_2)$. Therefore, $\forall i \in I.\ (d_i : T_1\theta_2) \implies (d_i' : T_2\theta_2)$, and hence $(d : T\theta_2)$.

$\underline{T = T_1 \vee T_2.}$

We have either $(d : T_1\theta_1)$ or $(d : T_2\theta_1)$. Therefore, since $P(T_i, \theta_1, \theta_2)$ holds for both i, by IH we have either $(d : T_1\theta_2)$ or $(d : T_2\theta_2)$, and hence $(d : T\theta_2)$.

$\underline{T = \neg T'.}$

We have $\neg(d : T'\theta_1)$. Since $P(T', \theta_2, \theta_1)$, by IH $(d : T'\theta_2) \implies (d : T'\theta_1)$. Therefore, by contrapositive, we have $\neg(d : T'\theta_2)$, hence $(d : \neg T'\theta_2)$. □

Lemma A.10. *For every gradual types $\tau_1, \tau_2 \in$ GTypes, if $\tau_1 \preccurlyeq \tau_2$ then there exists $T \in \star^{\mathsf{var}}(\tau_1)$ and $\theta : \mathscr{V}^X \to$ GTypes such that $T\theta = \tau_2$.*

Proof. By definition of $\tau_1 \preccurlyeq \tau_2$, there exist a T_1 and a $\theta_1 : \mathscr{V}^X \to$ GTypes such that $T_1^\dagger = \tau_1$ and that $T_1\theta_1 = \tau_2$. Let $\mathsf{vars}^X(T_1) = \{X_1, \ldots, X_n\}$.

By Corollary 5.8, we can find a T such that $\mathsf{vars}^X_{\mathsf{cov}}(T) \subseteq \{X_1, \ldots, X_n\}$ is disjoint from $\mathsf{vars}^X_{\mathsf{con}}(T) \subseteq \{X'_1, \ldots, X'_n\}$ and such that $T_1 = T\,[X_i/X'_i]^n_{i=1}$. Clearly, $T^\dagger = T_1^\dagger = \tau_1$.

We take θ to be $[X_i\theta_1/X_i]^n_{i=1} \cup [X_i\theta_1/X'_i]^n_{i=1}$ restricted to $\mathsf{vars}^X(T)$. We have:

$$T\theta = T([X_i\theta_1/X_i]^n_{i=1} \cup [X_i\theta_1/X'_i]^n_{i=1}) = T\,[X_i/X'_i]^n_{i=1}\theta_1 = T_1\theta_1 = \tau_2\,. \qquad \square$$

Lemma A.11. *For every gradual types $\tau_1, \tau_2, \tau_3 \in$ GTypes, if $\tau_1 \leq \tau_2 \preccurlyeq \tau_3$ then there exists $\tau'_2 \in$ GTypes such that $\tau_1 \preccurlyeq \tau'_2 \leq \tau_3$.*

Proof. By Lemma A.10, since $\tau_2 \preccurlyeq \tau_3$, there exist T_2 and $\theta : \mathscr{V}^X(T_2) \to$ GTypes such that $T_2^\dagger = \tau_2$, that $T_2\theta = \tau_3$, and that $\mathsf{vars}^X_{\mathsf{cov}}(T_2) \cap \mathsf{vars}^X_{\mathsf{con}}(T_2) = \emptyset$. Assume that $\mathsf{vars}^X_{\mathsf{cov}} = \{X_1, \ldots, X_n\}$ and $\mathsf{vars}^X_{\mathsf{con}} = \{Y_1, \ldots, Y_m\}$.

Let $\overline{\theta} = [(X_i\theta)^{\mathbb{O}}/X_i]^n_{i=1} \cup [(Y_i\theta)^{\mathbb{O}}/Y_i]^n_{i=1}$. We have $(T_2\overline{\theta})^\dagger = T_2\theta = \tau_3$.

Let $\hat{\theta} = [\bigwedge^n_{j=1} X_j\overline{\theta}/X_i]^n_{i=1} \cup [\bigvee^m_{j=1} Y_j\overline{\theta}/Y_i]^m_{i=1}$.

Let $\check{\theta} = [\bigwedge^n_{j=1} X_j\overline{\theta}/X^1] \cup [\bigvee^m_{j=1} Y_j\overline{\theta}/X^0]$.

We have:

$$\forall i = 1, \ldots, n.\ X_i\hat{\theta} \leq X_i\overline{\theta} \qquad \forall i = 1, \ldots, m.\ Y_i\overline{\theta} \leq Y_i\theta$$

We take $\tau'_2 = (\tau_1^{\mathbb{O}}\check{\theta})^\dagger$. We must show:

$$\tau_1 \preccurlyeq (\tau_1^{\mathbb{O}}\check{\theta})^\dagger \qquad (\tau_1^{\mathbb{O}}\check{\theta})^\dagger \leq \tau_3$$

The former holds because $(\tau_1^{\mathbb{O}}\check{\theta})^\dagger = \tau_1^{\mathbb{O}}\,[\bigwedge^n_{j=1} X_j\theta/X^1]\,[\bigvee^m_{j=1} Y_j\theta/X^0]$ and $\tau_1^{\mathbb{O}} \in \star(\tau_1)$.

To show the latter, we show:

$$(\tau_1^{\mathbb{O}}\check{\theta})^\dagger \leq (\tau_2^{\mathbb{O}}\check{\theta})^\dagger \qquad \tau_2^{\mathbb{O}}\check{\theta} = T_2\hat{\theta} \qquad (T_2\hat{\theta})^\dagger \leq (T_2\overline{\theta})^\dagger$$

We show $(\tau_1^{\mathbb{O}}\check{\theta})^\dagger \leq (\tau_2^{\mathbb{O}}\check{\theta})^\dagger$. By Theorem 5.18, $\tau_1 \leq \tau_2$ implies $\tau_1^{\mathbb{O}} \leq \tau_2^{\mathbb{O}}$. By Proposition 5.20, $\tau_1^{\mathbb{O}}\check{\theta} \leq \tau_2^{\mathbb{O}}\check{\theta}$. Both $\tau_1^{\mathbb{O}}\check{\theta}$ and $\tau_2^{\mathbb{O}}\check{\theta}$ are covariantly polarized, therefore, $(\tau_1^{\mathbb{O}}\check{\theta})^{\dagger^{\mathbb{O}}} = \tau_1^{\mathbb{O}}\check{\theta}$ and $(\tau_2^{\mathbb{O}}\check{\theta})^{\dagger^{\mathbb{O}}} = \tau_2^{\mathbb{O}}\check{\theta}$. Hence, $(\tau_1^{\mathbb{O}}\check{\theta})^\dagger \leq (\tau_2^{\mathbb{O}}\check{\theta})^\dagger$.

To show $\tau_2^{\mathbb{O}}\check{\theta} = T_2\hat{\theta}$, just note that $\tau_2^{\mathbb{O}} = T_2([X^1/X_i]^n_{i=1} \cup [X^0/Y_i]^m_{i=1})$.

Now we show $(T_2\hat{\theta})^\dagger \leq (T_2\overline{\theta})^\dagger$. First, note that $\hat{\theta}|_{\mathsf{vars}_{\mathsf{cov}}(T_2)} \leq \overline{\theta}|_{\mathsf{vars}_{\mathsf{cov}}(T_2)}$ and $\overline{\theta}|_{\mathsf{vars}_{\mathsf{con}}(T_2)} \leq \hat{\theta}|_{\mathsf{vars}_{\mathsf{con}}(T_2)}$. Hence, by Proposition 5.20, we have $T_2\hat{\theta} \leq T_2\overline{\theta}$. Since both $T_2\hat{\theta}$ and $T_2\overline{\theta}$ are covariantly polarized, we have $T_2\hat{\theta} = ((T_2\hat{\theta})^\dagger)^{\mathbb{O}}$ and $T_2\overline{\theta} = ((T_2\overline{\theta})^\dagger)^{\mathbb{O}}$. This yields the result we need. □

Simplifying the semantics

To tackle the proof of Proposition 6.6, we follow the same principle as in Chapter 2. We extend the interpretation of Definition 2.8 to account for the new element $\mathfrak{V}$ as follows:

$$(\{(\iota_i, \partial_i) \mid i \in I\}^L :_\eta t_1 \to t_2)^q = \forall i \in I.\ (\iota_i = \mathfrak{V} \vee (\iota_i :_\eta t_1)^q) \implies (\partial_i :_\eta t_2)^q$$

We define the interpretation $[\![t]\!]^{q\prime}_\eta$ of a set-theoretic type t as:

$$[\![t]\!]^{q\prime}_\eta \stackrel{\text{def}}{=} \{d \in \mathscr{D}' \mid (d :_\eta t)^q\}$$

And we now write $\dot{\leq}^q$ for the induced subtyping relation on set-theoretic types as:

$$t_1 \dot{\leq}^q t_2 \iff^{\text{def}} \forall \eta : \mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D}').\ [\![t_1]\!]^{q\prime}_\eta \subseteq [\![t_2]\!]^{q\prime}_\eta$$

We also define the canonical assigment $\overline{\eta}$ on the new domain as $\overline{\eta}(\alpha) = \{d \in \mathscr{D}' \mid \alpha \in \mathsf{tags}(d)\}$.

Lemma A.12. *For every type $t \in \mathsf{Types}$, $[\![t]\!] = [\![t]\!]^q_{\overline{\eta}}$.*

Proof. The statement is proven by a straightforward induction on the pair (d, t). □

Lemma A.13. *Let $V \in \mathscr{P}_f(\mathscr{V}^\alpha)$, and $T = \{t \in \mathsf{Types} \mid \mathsf{vars}(t) \subseteq V\}$. For every $t \in T$, the following holds:*

$$[\![t]\!]^{q\prime}_{\overline{\eta}} = \emptyset \implies \forall \eta : \mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D}').\ [\![t]\!]^{q\prime}_\eta = \emptyset$$

Proof. We follow the exact same proof as for Lemma 2.10. We prove the stronger statement:

$$\forall t \in T.\ \forall d \in \mathscr{D}.\ (d :_\eta t)^q \iff (F^\eta_V(d) :_{\overline{\eta}} t)^q$$

where the function $F^\eta_V(.)$ is defined as follows:

$$F^\eta_V(d) = \begin{cases} c^{l^\eta_V(d)} & \text{if } d = c^L \\ (F^\eta_V(d_1), F^\eta_V(d_2))^{l^\eta_V(d)} & \text{if } d = (d_1, d_2)^L \\ \{(F^\eta_\Omega(\iota_i), F^\eta_\Omega(\partial_i)) \mid i \in I\}^{l^\eta_V(d)} & \text{if } d = \{(\iota_i, \partial_i) \mid i \in I\}^L \end{cases}$$

$$l^\eta_V(d) = \{\alpha \in V \mid d \in \eta(\alpha)\}$$

Where $F^\eta_V(\Omega) = \Omega$ and $F^\eta_V(\mathfrak{V}) = \mathfrak{V}$.

The proof is done by induction on the pair (d, t) ordered lexicographically, and by case analysis on t. The only case that changes from the proof of Lemma 2.10 is the following.

- $\underline{t = t_1 \to t_2}$. If d is not of the form $\{(\iota_i, \partial_i) \mid i \in I\}^L$, then the result is immediate. Otherwise, we have

$$(\{(\iota_i, \partial_i) \mid i \in I\}^L :_\eta t_1 \to t_2)^q \iff (\forall i \in I.\ (\iota_i = \mathfrak{V} \vee (\iota_i :_\eta t_1)^q) \implies (\partial_i :_\eta t_2)^q)$$
$$(F^\eta_V(\{(\iota_i, \partial_i) \mid i \in I\}^L) :_{\overline{\eta}} t_1 \to t_2)^q$$
$$\iff (\forall i \in I.\ (\iota_i = F^\eta_V(\mathfrak{V}) \vee (F^\eta_V(\iota_i) :_{\overline{\eta}} t_1)^q) \implies (F^\eta_V(\partial_i) :_{\overline{\eta}} t_2)^q)$$

and for $i \in I$, both $(d_i :_\eta t_1)^q \iff (F^\eta_V(d_i) :_{\overline{\eta}} t_1)^q$ and $(\partial_i :_\eta t_2)^q \iff$

$(F_V^\eta(\partial_i) \mathbin{:_{\overline{\eta}}} t_2)^q$ hold by IH, and $F_V^\eta(\mho) = \mho$. Hence the result.

□

Proposition A.14. *For all types $t_1, t_2 \in \mathsf{Types}$, $t_1 \dot{\leq} t_2 \iff t_1 \dot{\leq}^q t_2$.*

Proof. By definition, $t_1 \dot{\leq} t_2 \iff [\![t_1 \setminus t_2]\!]' = \emptyset$, and $t_1 \dot{\leq}^q t_2 \iff \forall \eta.\ [\![t_1 \setminus t_2]\!]_\eta^{q\prime} = \emptyset$. By Lemma A.12, the first statement becomes $t_1 \dot{\leq} t_2 \iff [\![t_1 \setminus t_2]\!]_{\overline{\eta}}^{q\prime} = \emptyset$. The implication $t_1 \dot{\leq}^q t_2 \implies t_1 \dot{\leq} t_2$ is then immediate. The converse follows from Lemma A.13, taking $V = \mathsf{vars}(t_1 \setminus t_2)$. □

Lemma A.15. *For every $t \in \mathsf{Types}$, every substitution $\theta : \mathscr{V}^\alpha \to \mathsf{Types}$, and every assignment $\eta : \mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D}')$, if η' is defined by $\eta'(\alpha) = [\![\alpha\theta]\!]_\eta^{q\prime}$, then $[\![t\theta]\!]_\eta^{q\prime} = [\![t]\!]_{\eta'}^{q\prime}$.*

Proof. The statement is shown by straightforward induction on the pair (d, t). □

Proposition A.16. *For every types $t_1, t_2 \in \mathsf{Types}$, if $t_1 \dot{\leq} t_2$ then $t_1\theta \dot{\leq} t_2\theta$ for every type substitution θ.*

Proof. By Proposition A.14, we have $t_1 \dot{\leq}^q t_2$. By the definition of $[\![.]\!]_\cdot^{q\prime}$, this proves $\forall \eta.\ [\![t_1 \setminus t_2]\!]_\eta^{q\prime} = \emptyset$.
Now consider an arbitrary $\theta : \mathscr{V}^\alpha \to \mathsf{Types}$ and an assignment $\eta : \mathscr{V}^\alpha \to \mathscr{P}(\mathscr{D}')$. Consider η' defined as $\eta'(\alpha) = [\![\alpha\theta]\!]_\eta^{q\prime}$. By Lemma A.15, since $[\![t_1 \setminus t_2]\!]_{\eta'}^{q\prime} = \emptyset$, we deduce that $[\![(t_1 \setminus t_2)\theta]\!]_\eta^{q\prime} = \emptyset$. This proves that $t_1\theta \dot{\leq}^q t_2\theta$, and the result follows from Proposition A.14. □

Proposition A.17. *For every type frame $T \in \mathsf{TFrames}$, and every type substitutions $\theta_1, \theta_2 : \mathscr{V}^\alpha \cup \mathscr{V}^X \to \mathsf{GTypes}$ satisfying the following two conditions:*

$$\forall A \in \mathsf{vars}_{\mathsf{cov}}(T), A\theta_1 \dot{\leq} A\theta_2 \qquad \forall A \in \mathsf{vars}_{\mathsf{con}}(T), A\theta_2 \dot{\leq} A\theta_1$$

then it holds that $T\theta_1 \dot{\leq} T\theta_2$.

Proof. We follow the same proof as for Proposition A.9.
For every substitution θ, and every set $S \in \mathsf{Vars}$, we write $\theta|_S$ for the substitution that verifies $X\theta|_S = X\theta$ if $X \in S$ and $X\theta|_S = X$ otherwise.
We define

$$P(T, \theta_1, \theta_2) \overset{\text{def}}{\iff} \left(\theta_1|_{\mathsf{vars}_{\mathsf{cov}}(T)} \dot{\leq} \theta_2|_{\mathsf{vars}_{\mathsf{cov}}(T)}\right) \text{ and } \left(\theta_2|_{\mathsf{vars}_{\mathsf{con}}(T)} \dot{\leq} \theta_1|_{\mathsf{vars}_{\mathsf{con}}(T)}\right)$$

and note that the following hold

$$\begin{aligned}
P(A,\theta_1,\theta_2) &\implies A\theta_1 \dot{\leq} A\theta_2\\
P(T_1\times T_2,\theta_1,\theta_2) &\implies P(T_1,\theta_1,\theta_2) \text{ and } P(T_2,\theta_1,\theta_2)\\
P(T_1\to T_2,\theta_1,\theta_2) &\implies P(T_1,\theta_2,\theta_1) \text{ and } P(T_2,\theta_1,\theta_2)\\
P(T_1\vee T_2,\theta_1,\theta_2) &\implies P(T_1,\theta_1,\theta_2) \text{ and } P(T_2,\theta_1,\theta_2)\\
P(\neg T',\theta_1,\theta_2) &\implies P(T',\theta_2,\theta_1)
\end{aligned}$$

We show the following result (which implies the statement)

$$\forall\theta_1,\theta_2,d,T.\quad \left.\begin{array}{c} P(T,\theta_1,\theta_2)\\ (d:T\theta_1)\end{array}\right\} \implies (d:T\theta_2)$$

by induction on (d,T). The only case that changes from the proof of Proposition A.9 is the following:

$\underline{T = T_1\to T_2.}$

We have $T\theta_1 = (T_1\theta_1)\to(T_2\theta_1)$ and $T\theta_2 = (T_1\theta_2)\to(T_2\theta_2)$.

Since $(d:T\theta_1)$, we have $d = \{(\iota_i,\partial_i)\mid i\in I\}$ and $\forall i\in I.\ (\iota_i = \mho \vee (\iota_i : T_1\theta_1)) \implies (\partial_i : T_2\theta_1)$.

We have $P(T_1,\theta_2,\theta_1)$ and $P(T_2,\theta_1,\theta_2)$.

For every ι_i such that $(\iota_i : T_1\theta_2)$, by IH we have $(\iota_i : T_1\theta_1)$, therefore $(\partial_i : T_2\theta_1)$, and, by IH, $(\partial_i : T_2\theta_2)$. Additionally, for every ι_i such that $\iota_i = \mho$, we have $(\partial_i : T_2\theta_1)$, and, by IH, $(\partial_i : T_2\theta_2)$. Therefore, $\forall i\in I.\ (\iota_i = \mho \vee (d_i : T_1\theta_2)) \implies (d_i' : T_2\theta_2)$, and hence $(d : T\theta_2)$. □

Lemma A.18. *For every gradual type $\tau\in$ GTypes, we have $\tau^{\Downarrow}\dot{\leq}\tau^{\Uparrow}$.*

Proof. Let $\theta_{\Downarrow} = [\mathbb{0}/X^1]\,[\mathbb{1}/X^0]$ and $\theta_{\Uparrow} = [\mathbb{1}/X^1]\,[\mathbb{0}/X^0]$. We have $\tau^{\Downarrow} = \tau^{\mathbb{O}}\theta_{\Downarrow}$ and $\tau^{\Uparrow} = \tau^{\mathbb{O}}\theta_{\Uparrow}$.

Let $T = \tau^{\mathbb{O}}$. We have $\mathrm{vars}^X_{\mathrm{cov}}(T) = \{X_1\}$ and $\mathrm{vars}^X_{\mathrm{con}}(T) = \{X^0\}$. Moreover, we have $X_1\theta_{\Downarrow} = \mathbb{0}\dot{\leq}\mathbb{1} = X_1\theta_{\Uparrow}$, and $X_0\theta_{\Uparrow} = \mathbb{0}\dot{\leq}\mathbb{1} = X_0\theta_{\Downarrow}$.

Therefore, by Proposition A.17 on T and the substitutions $\theta_{\Uparrow}$ and $\theta_{\Downarrow}$, we deduce $T\theta_{\Downarrow}\dot{\leq}T\theta_{\Uparrow}$, which gives $\tau^{\Downarrow}\dot{\leq}\tau^{\Uparrow}$. □

We introduce the following two lemmas from Frisch [25] (Lemma 4.6 and Lemma 4.8).

Lemma A.19. *Let $(X_i)_{i\in P}$, $(X_i)_{i\in N}$, $(Y_i)_{i\in P}$, $(Y_i)_{i\in N}$ be four finite families of sets. Then:*

$$\left(\bigcap_{i\in P} X_i\times Y_i\right)\setminus\left(\bigcup_{i\in N} X_i\times Y_i\right) = \bigcup_{N'\subseteq N}\left(\bigcap_{i\in P} X_i\setminus\bigcup_{i\in N'} X_i\right)\times\left(\bigcap_{i\in P} Y_i\setminus\bigcup_{i\in N\setminus N'} Y_i\right)$$

Lemma A.20. *Let $(X_i)_{i\in P}$ and $(X_i)_{i\in N}$ be two finite families of sets. Then:*

$$\bigcap_{i\in P}\mathscr{P}_f(X_i)\subseteq\bigcup_{i\in N}\mathscr{P}_f(X_i) \iff \exists i_0\in N.\ \bigcap_{i\in P} X_i\subseteq X_{i_0}$$

Lemma A.21. *For every types $t_1,t_2\in$ Types, $[\![t_1\to t_2]\!]' = \mathscr{P}_f((\overline{[\![t_1]\!]'}^{\mathscr{D}'_\Omega}\cup\{\mho\})\times\overline{[\![t_2]\!]'}^{\mathscr{D}'_\Omega})^{\mathscr{D}'\times\mathscr{D}'_\Omega}$.*

Proof. We have the following equalities:

$$\begin{aligned}\llbracket t_1 \to t_2 \rrbracket' = &\{R \in \mathscr{P}_f(\mathscr{I}' \times \mathscr{D}'_\Omega) \mid \forall (\iota, \partial) \in R,\ \neg((\iota = \mho \vee \iota \in \llbracket t_1 \rrbracket') \wedge \partial \notin \llbracket t_2 \rrbracket')\} \\ &- \{R \in \mathscr{P}_f(\mathscr{I}' \times \mathscr{D}'_\Omega) \mid R \cap ((\llbracket t_1 \rrbracket' \cup \{\mho\}) \times \overline{\llbracket t_2 \rrbracket'}^{\mathscr{D}'_\Omega}) = \emptyset\} \\ &- \{R \in \mathscr{P}_f(\mathscr{I}' \times \mathscr{D}'_\Omega) \mid R \subseteq \overline{(\llbracket t_1 \rrbracket' \cup \{\mho\}) \times \overline{\llbracket t_2 \rrbracket'}^{\mathscr{D}'_\Omega}}^{\mathscr{I}' \times \mathscr{D}'_\Omega}\}\end{aligned}$$

□

Lemma A.22. *Let P, N be two finite sets of arrow types where P is non-empty, we have:*

$$\bigwedge_{t_1 \to t_2 \in P} t_1 \to t_2 \dot{\leq} \bigvee_{t_1 \to t_2 \in N} t_1 \to t_2 \iff \exists \hat{t}_1 \to \hat{t}_2 \in N.$$

$$\left(\llbracket \hat{t}_1 \rrbracket' \subseteq \bigcup_{t_1 \to t_2 \in P} \llbracket t_1 \rrbracket'\right) \wedge \forall P' \subsetneq P.\ \left(\llbracket \hat{t}_1 \rrbracket' \cup \{\mho\} \subseteq \bigcup_{t_1 \to t_2 \in P'} (\llbracket t_1 \rrbracket' \cup \{\mho\})\right) \vee \left(\bigcap_{t_1 \to t_2 \in P \setminus P'} \llbracket t_2 \rrbracket' \subseteq \llbracket \hat{t}_2 \rrbracket'\right)$$

Proof. Corollary of lemmas A.21, A.20, and A.19. □

Note that according to the same lemma from Frisch et al. [27], the subtyping relation holds immediately if we can chose $\hat{t}_1$ to be empty. However, in our case, if $\llbracket \hat{t}_1 \rrbracket' = \emptyset$, then the conditions on the right yields in particular for $P' = \emptyset$ that we must have

$$\{\mho\} \subseteq \emptyset \vee \left(\bigcap_{t_1 \to t_2 \in P} \llbracket t_2 \rrbracket' \subseteq \llbracket \hat{t}_2 \rrbracket'\right)$$

which is equivalent to

$$\bigcap_{t_1 \to t_2 \in P} \llbracket t_2 \rrbracket' \subseteq \llbracket \hat{t}_2 \rrbracket'$$

since the condition on the left hand side of the disjunction cannot hold.

Proposition A.23. *For every gradual types $\tau, \tau', \sigma \in \mathsf{GTypes}$, if $\tau \gtrsim \mathbb{0} \to \mathbb{1}$ and $\sigma \gtrsim \widetilde{\mathrm{dom}}(\tau)$ then $\tau \gtrsim \sigma \to \tau \mathbin{\tilde{\circ}} \sigma$. Moreover, if $\tau \gtrsim \sigma \to \tau'$ then $\tau \mathbin{\tilde{\circ}} \sigma \gtrsim \tau'$.*

Proof. By definition of $\gtrsim$, we have $\tau^{\Downarrow} \dot{\leq} \mathbb{0} \to \mathbb{1}$, and $\sigma^{\Uparrow} \dot{\leq} (\widetilde{\mathrm{dom}}(\tau))^{\Uparrow} \simeq \mathrm{dom}(\tau^{\Downarrow})$, thus $\tau^{\Downarrow} \circ \sigma^{\Uparrow}$ is well-defined. A similar reasoning proves that $\tau^{\Uparrow} \circ \sigma^{\Downarrow}$ is also well-defined. Moreover, Proposition 6.13 proves that $\tau^{\Downarrow} \circ \sigma^{\Uparrow} \dot{\leq} \tau^{\Uparrow} \circ \sigma^{\Downarrow}$, hence $(\tau \mathbin{\tilde{\circ}} \sigma)^{\Uparrow} \simeq \tau^{\Uparrow} \circ \sigma^{\Downarrow}$ and $(\tau \mathbin{\tilde{\circ}} \sigma)^{\Downarrow} \simeq \tau^{\Downarrow} \circ \sigma^{\Uparrow}$ ①.
By Proposition 6.13, it holds that $\tau^{\Downarrow} \dot{\leq} \sigma^{\Uparrow} \to \tau^{\Downarrow} \circ \sigma^{\Uparrow}$. By ①, we deduce $\tau^{\Downarrow} \dot{\leq} \sigma^{\Uparrow} \to (\tau \mathbin{\tilde{\circ}} \sigma)^{\Downarrow}$, which proves $\tau^{\Downarrow} \dot{\leq} (\sigma \to \tau \mathbin{\tilde{\circ}} \sigma)^{\Downarrow}$. A similar reasoning with $\tau^{\Uparrow}$ proves $\tau \gtrsim \sigma \to \tau \mathbin{\tilde{\circ}} \sigma$.
Now if $\tau \gtrsim \sigma \to \tau'$, then by definition of $\gtrsim$ we have $\tau^{\Downarrow} \dot{\leq} (\sigma \to \tau')^{\Downarrow} \simeq \sigma^{\Uparrow} \to \tau'^{\Downarrow}$. By Proposition 6.18, we deduce $\tau^{\Downarrow} \circ \sigma^{\Uparrow} \dot{\leq} \tau'^{\Downarrow}$. By ①, we have $(\tau \mathbin{\tilde{\circ}} \sigma)^{\Downarrow} \dot{\leq} \tau'^{\Downarrow}$. A similar reasoning

with $\tau^{\Uparrow}$ proves $\tau \mathbin{\tilde{\circ}} \sigma \mathrel{\widetilde{\leq}} \tau'$. □

Proposition A.24. *For every gradual types $\tau, \tau_1, \tau_2 \in \mathsf{GTypes}$, if $\tau \mathrel{\widetilde{\leq}} \mathbb{1} \times \mathbb{1}$ then $\tau \mathrel{\widetilde{\leq}} \widetilde{\pi_1}(\tau) \times \widetilde{\pi_2}(\tau)$ and if $\tau \mathrel{\widetilde{\leq}} \tau_1 \times \tau_2$ then $\widetilde{\pi_1}(\tau) \mathrel{\widetilde{\leq}} \tau_1$ and $\widetilde{\pi_2}(\tau) \mathrel{\widetilde{\leq}} \tau_2$.*

Proof. By definition of $\mathrel{\widetilde{\leq}}$, we have $\tau^{\Downarrow} \mathrel{\dot{\leq}} \mathbb{1} \times \mathbb{1}$, thus $\pi_i(\tau^{\Downarrow})$ is well-defined for $i \in \{1, 2\}$. A similar reasoning proves that $\pi_i(\tau^{\Uparrow})$ is also well-defined. Moreover, Proposition 6.14 proves that $\pi_i(\tau^{\Downarrow}) \mathrel{\dot{\leq}} \pi_i(\tau^{\Uparrow})$, hence $(\widetilde{\pi_i}(\tau))^{\Uparrow} \simeq \pi_i(\tau^{\Uparrow})$ and $(\widetilde{\pi_i}(\tau))^{\Downarrow} \simeq \pi_i(\tau^{\Downarrow})$ for $i \in \{1, 2\}$ ①.
By Proposition 6.14, it holds that $\tau^{\Downarrow} \mathrel{\dot{\leq}} \pi_1(\tau^{\Downarrow}) \times \pi_2(\tau^{\Downarrow})$. By ①, we deduce $\tau^{\Downarrow} \mathrel{\dot{\leq}} (\widetilde{\pi_1}(\tau))^{\Downarrow} \times (\widetilde{\pi_2}(\tau))^{\Downarrow}$, which proves $\tau^{\Downarrow} \mathrel{\dot{\leq}} (\widetilde{\pi_1}(\tau) \times \widetilde{\pi_2}(\tau))^{\Downarrow}$. A similar reasoning with $\tau^{\Uparrow}$ proves $\tau \mathrel{\widetilde{\leq}} \widetilde{\pi_1}(\tau) \times \widetilde{\pi_2}(\tau)$.
Now if $\tau \mathrel{\widetilde{\leq}} \tau_1 \times \tau_2$, then by definition of $\mathrel{\widetilde{\leq}}$ we have $\tau^{\Downarrow} \mathrel{\dot{\leq}} (\tau_1 \times \tau_2)^{\Downarrow} \simeq {\tau_1}^{\Downarrow} \times {\tau_2}^{\Downarrow}$. By Proposition 6.19, we deduce $\pi_i(\tau^{\Downarrow}) \mathrel{\dot{\leq}} {\tau_i}^{\Downarrow}$ for $i \in \{1, 2\}$. By ①, we have $(\widetilde{\pi_i}(\tau))^{\Downarrow} \mathrel{\dot{\leq}} {\tau_i}^{\Downarrow}$. A similar reasoning with $\tau^{\Uparrow}$ proves $\widetilde{\pi_i}(\tau) \mathrel{\widetilde{\leq}} \tau_i$. □

Proposition A.25. *For every gradual types $\tau, \tau', \sigma \in \mathsf{GTypes}$ such that $\tau \mathrel{\widetilde{\leq}} \mathbb{0} \to \mathbb{1}$, $\tau' \mathrel{\widetilde{\leq}} \mathbb{0} \to \mathbb{1}$, $\sigma \mathrel{\widetilde{\leq}} \widetilde{\mathsf{dom}}(\tau)$, and $\sigma \mathrel{\widetilde{\leq}} \widetilde{\mathsf{dom}}(\tau')$, if $\tau \mathrel{\widetilde{\leqslant}} \tau'$ then $\tau \mathbin{\tilde{\circ}} \sigma \mathrel{\widetilde{\leqslant}} \tau' \mathbin{\tilde{\circ}} \sigma$.*

Proof. By Proposition 6.3, we have $\tau^{\Downarrow} \mathrel{\dot{\leq}} \tau'^{\Downarrow}$. By Proposition 6.13, we have $\tau^{\Downarrow} \circ \sigma^{\Uparrow} \mathrel{\dot{\leq}} \tau'^{\Downarrow} \circ \sigma^{\Uparrow}$. Thus, by Definition 6.15, $(\tau \mathbin{\tilde{\circ}} \sigma)^{\Downarrow} \mathrel{\dot{\leq}} (\tau' \mathbin{\tilde{\circ}} \sigma)^{\Downarrow}$. A similar reasoning proves $(\tau' \mathbin{\tilde{\circ}} \sigma)^{\Uparrow} \mathrel{\dot{\leq}} (\tau \mathbin{\tilde{\circ}} \sigma)^{\Uparrow}$, hence $\tau \mathbin{\tilde{\circ}} \sigma \mathrel{\widetilde{\leqslant}} \tau' \mathbin{\tilde{\circ}} \sigma$. □

Proposition A.26. *For every gradual types $\tau, \tau' \in \mathsf{GTypes}$ such that $\tau \mathrel{\widetilde{\leq}} \mathbb{1} \times \mathbb{1}$ and $\tau' \mathrel{\widetilde{\leq}} \mathbb{1} \times \mathbb{1}$, if $\tau \mathrel{\widetilde{\leqslant}} \tau'$ then for every $i \in \{1, 2\}$, $\widetilde{\pi_i}(\tau) \mathrel{\widetilde{\leqslant}} \widetilde{\pi_i}(\tau')$.*

Proof. Let $i \in \{1, 2\}$. By Proposition 6.3, we have $\tau^{\Downarrow} \mathrel{\dot{\leq}} \tau'^{\Downarrow}$. By Proposition 6.14, we have $\pi_i(\tau^{\Downarrow}) \mathrel{\dot{\leq}} \pi_i(\tau'^{\Downarrow})$. Thus, by Definition 6.15, $(\widetilde{\pi_i}(\tau))^{\Downarrow} \mathrel{\dot{\leq}} (\widetilde{\pi_i}(\tau'))^{\Downarrow}$. A similar reasoning proves $(\widetilde{\pi_i}(\tau'))^{\Uparrow} \mathrel{\dot{\leq}} (\widetilde{\pi_i}(\tau))^{\Uparrow}$, hence $\widetilde{\pi_i}(\tau) \mathrel{\widetilde{\leqslant}} \widetilde{\pi_i}(\tau')$. □

Lemma A.27. *For every value $V \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ and every type environment Γ, if $\Gamma \vdash V : \tau$ then $\Gamma \vdash V : \mathsf{type}(V)$ and $\mathsf{type}(V) \mathrel{\widetilde{\leq}} \tau$.*

Proof. By case analysis on V, and induction over the derivation $\Gamma \vdash V : \tau$. If the last rule used in the derivation is $[\mathrm{T}_{\mathsf{Sub}}^{\langle \mathsf{ST} \rangle}]$, then the result follows immediately by induction. Otherwise, we distinguish the following cases:

- $\underline{V = c.}$ The only rule that can be applied is $[\mathrm{T}_{\mathsf{Cst}}^{\langle \mathsf{ST} \rangle}]$, and $\tau = b_c \wedge ? = \mathsf{type}(c)$.
- $\underline{V = \lambda^{\tau_1 \to \tau_2} x.\, E.}$ By $[\mathrm{T}_{\mathsf{Abstr}}^{\langle \mathsf{ST} \rangle}]$, $\tau = \tau_1 \to \tau_2 = \mathsf{type}(V)$.
- $\underline{V = (V_1, V_2).}$ By $[\mathrm{T}_{\mathsf{Pair}}^{\langle \mathsf{ST} \rangle}]$, $\tau = \tau_1 \times \tau_2$ where $\forall i \in \{1, 2\}$, $\Gamma \vdash V_i : \tau_i$. By induction hypothesis, $\Gamma \vdash V_i : \mathsf{type}(V_i)$ and $\mathsf{type}(V_i) \mathrel{\widetilde{\leq}} \tau_i$. Therefore, $\mathsf{type}(V) = \mathsf{type}(V_1) \times \mathsf{type}(V_2) \mathrel{\widetilde{\leq}} \tau$ and by $[\mathrm{T}_{\mathsf{Pair}}^{\langle \mathsf{ST} \rangle}]$, $\Gamma \vdash V : \mathsf{type}(V)$.
- $\underline{V = V'\langle \tau_1 \Rightarrow_p \tau_2 \rangle}$. Suppose that p is positive. The negative case is proven similarly.

By $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cast+}}]$, we have $\tau = \tau_2$, and $\mathsf{type}(V) = \tau_2$, hence the result.

- $\underline{V = \Lambda\vec{\alpha}.\, E.}$ Forbidden by the hypothesis $\Gamma \vdash V : \tau$, which cannot contain a type scheme.

□

Lemma A.28 (Progress). *For every term $E \in \mathsf{Terms}^{\langle \mathsf{ST} \rangle}$, if $\emptyset \vdash E : \forall\vec{\alpha}.\tau$ then one of the following holds:*

- *there exists $E' \in \mathsf{Terms}^{\langle \mathsf{ST} \rangle}$ such that $E \rightsquigarrow E'$;*
- *there exists $\ell \in \mathscr{L}$ such that $E \rightsquigarrow \mathsf{blame}\ \ell$;*
- $E \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$.

Proof. By induction on the derivation $\emptyset \vdash E : \forall\vec{\alpha}.\tau$ and case analysis over the last rule used.

- $\underline{[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cst}}]}$. Immediate, E is a value c.
- $\underline{[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Var}}]}$. Impossible since E is not well-typed in an empty environment.
- $\underline{[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Proj}}]}$. We have $E = \pi_i\, E'$ where $\emptyset \vdash E' : \tau_1 \times \tau_2$. By induction hypothesis, we distinguish three cases:
  - $E' \rightsquigarrow E''$, in which case, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}]$ and $\mathscr{E} = \pi_i\, [\,]$, we have $E \rightsquigarrow \pi_i\, E''$;
  - $E' \rightsquigarrow \mathsf{blame}\ \ell$, in which case, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CtxBlame}}]$ and $\mathscr{E} = \pi_i\, [\,]$, we have $E \rightsquigarrow \mathsf{blame}\ \ell$;
  - $E' \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ in which case, by inversion of the typing rules, either $E' = (V_1, V_2)$ and $E \rightsquigarrow V_i$ by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Proj}}]$, or $E' = V'\langle \tau_1' \Rightarrow_p \tau_2' \rangle$ where $\tau_2' \cong \tau_1 \times \tau_2$ and E reduces by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CProj}}]$.
- $\underline{[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Pair}}]}$. We have $E = (E_1, E_2)$ where for every $i \in \{1, 2\}$, $\emptyset \vdash E_i : \tau_i$. By induction hypothesis, we distinguish the following cases:
  - $E_2 \rightsquigarrow E_2'$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}]$ and $\mathscr{E} = (E_1, [\,])$ we have $E \rightsquigarrow (E_1, E_2')$;
  - $E_2 \rightsquigarrow \mathsf{blame}\ \ell$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CtxBlame}}]$ and $\mathscr{E} = (E_1, [\,])$ we have $E \rightsquigarrow \mathsf{blame}\ \ell$;
  - $E_2 \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ and $E_1 \rightsquigarrow E_1'$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}]$ and $\mathscr{E} = ([\,], E_2)$ we have $E \rightsquigarrow (E_1', E_2)$;
  - $E_2 \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ and $E_1 \rightsquigarrow \mathsf{blame}\ \ell$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}]$ and $\mathscr{E} = ([\,], E_2)$ we have $E \rightsquigarrow \mathsf{blame}\ \ell$;
  - $E_2 \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ and $E_1 \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$, then $E \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$.
- $\underline{[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{App}}]}$. We have $E = E_1\, E_2$ where $\emptyset \vdash E_1 : \tau_2 \to \tau_1$ and $\emptyset \vdash E_2 : \tau_2$. By induction hypothesis, we distinguish the following cases:
  - $E_2 \rightsquigarrow E_2'$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}]$ and $\mathscr{E} = E_1\, [\,]$ we have $E \rightsquigarrow E_1\, E_2'$;
  - $E_2 \rightsquigarrow \mathsf{blame}\ \ell$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CtxBlame}}]$ and $\mathscr{E} = E_1\, [\,]$ we have $E \rightsquigarrow \mathsf{blame}\ \ell$;
  - $E_2 \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ and $E_1 \rightsquigarrow E_1'$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}]$ and $\mathscr{E} = [\,]\, E_2$ we have $E \rightsquigarrow E_1'\, E_2$;

  - $E_2 \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ and $E_1 \rightsquigarrow \mathsf{blame}\ \ell$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}]$ and $\mathscr{E} = []\, E_2$ we have $E \rightsquigarrow \mathsf{blame}\ \ell$;
  - $E_2 \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ and $E_1 \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$, then by inversion of the typing rules either $E_1 = \lambda^{\tau_2' \to \tau_1'} x.\, E'$, and by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{App}}]$ we have $E \rightsquigarrow E'\,[E_2/x]$; or $E_1 = V_1' \langle \tau_1' \Rightarrow_p \tau_2' \rangle$ where $\tau_2' \gtrless \tau_2 \to \tau_1$ and E reduces by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CApp}}]$.

- $\underline{[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Abstr}}]}$. Immediate since E is necessarily a value.

- $\underline{[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Let}}]}$. We have $E = \mathsf{let}\, x = E_1\, \mathsf{in}\, E_2$ where $\emptyset \vdash E_1 : \forall \vec{\alpha}_1.\tau_1$ and $x : \forall \vec{\alpha}_1.\tau_1 \vdash E_2 : \tau_2$. By induction hypothesis, we distinguish the following cases:
  - $E_1 \rightsquigarrow E_1'$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}]$ and $\mathscr{E} = \mathsf{let}\, x = []\, \mathsf{in}\, E_2$ we have $E \rightsquigarrow \mathsf{let}\, x = E_1'\, \mathsf{in}\, E_2$;
  - $E_1 \rightsquigarrow \mathsf{blame}\ \ell$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CtxBlame}}]$ and $\mathscr{E} = \mathsf{let}\, x = []\, \mathsf{in}\, E_2$ we have $E \rightsquigarrow \mathsf{blame}\ \ell$;
  - $E_1 \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Let}}]$ we have $E \rightsquigarrow E_2\,[E_1/x]$.

- $\underline{[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{TAbstr}}]}$. Immediate since E is a value.

- $\underline{[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{TApp}}]}$. We have $E = E'[\vec{t}]$ where $\emptyset \vdash E' : \forall \vec{\alpha}'.\tau'$. By induction hypothesis, we distinguish three cases:
  - $E' \rightsquigarrow E''$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}]$ and $\mathscr{E} = [][\vec{t}]$ we have $E \rightsquigarrow E''[\vec{t}]$;
  - $E' \rightsquigarrow \mathsf{blame}\ \ell$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CtxBlame}}]$ and $\mathscr{E} = [][\vec{t}]$ we have $E \rightsquigarrow \mathsf{blame}\ \ell$,
  - $E' \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$, then by inversion of the typing rules, we have $E' = \Lambda \vec{\alpha}'.\, E''$ and by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{TApp}}]$ we have $E \rightsquigarrow E''\,[\vec{t}/\vec{\alpha}']$.

- $\underline{[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cast+}}]}$. We have $E = E'\langle \tau_1 \Rightarrow_\ell \tau_2 \rangle$ where $\emptyset \vdash E' : \tau_1$. By induction hypothesis, we distinguish the following cases:
  - $E' \rightsquigarrow E''$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}]$ and $\mathscr{E} = []\langle \tau_1 \Rightarrow_\ell \tau_2 \rangle$ we have $E \rightsquigarrow E''\langle \tau_1 \Rightarrow_\ell \tau_2 \rangle$;
  - $E' \rightsquigarrow \mathsf{blame}\ \ell'$, by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CtxBlame}}]$ and $\mathscr{E} = []\langle \tau_1 \Rightarrow_\ell \tau_2 \rangle$ we have $E \rightsquigarrow \mathsf{blame}\ \ell'$;
  - $E' \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ where $\tau_1 \vee \tau_2 \not\gtrless \mathsf{cons}(E')$, then $E \rightsquigarrow E'\langle \tau_1 \wedge \mathsf{cons}(E') \Rightarrow_\ell \tau_2 \wedge \mathsf{cons}(E') \rangle$ by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cons}}]$;
  - $E' \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ where $\tau_2 \lesssim \mathbb{0}$, then $E \rightsquigarrow \mathsf{blame}\ \ell$ by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Blame}}]$;
  - $E' \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ where $\tau_2 \not\lesssim \mathbb{0}$ and $\tau_1 \vee \tau_2 \lesssim \mathsf{cons}(E')$, if $\mathsf{cons}(E') = \mathbb{1} \times \mathbb{1}$ or $\mathsf{cons}(E') = \mathbb{0} \to \mathbb{1}$ then E is a value. Otherwise, $\mathsf{cons}(E') \in \mathscr{B}$ and by Lemma 6.31, we have $E' = c$ and $\mathsf{cons}(E') = b_c$. Since $\tau_2 \lesssim b_c$ and $\tau_2 \not\lesssim \mathbb{0}$, we deduce that $\tau_2 \wedge b_c \not\lesssim \mathbb{0}$. By Lemma 6.30, we deduce that $b_c \wedge ? \lesssim \tau_2$, thus $E \rightsquigarrow c$ by $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Simpl}}]$.

- $\underline{[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cast-}}]}$. We have $E = E'\langle \tau_1 \Rightarrow_\ell \tau_2 \rangle$ where $\emptyset \vdash E' : \tau_1$ and $\tau_2 \lnsim \tau_1$. We follow the same reasoning as in the previous case and distinguish the same cases, except when $E' \in \mathsf{Values}^{\langle \mathsf{ST} \rangle}$ where $\tau_2 \lesssim \mathbb{0}$. By Lemma 6.28, we deduce that $\tau_1 \not\lesssim \mathbb{0}$. Thus, by Lemma 6.29, necessarily $\tau_2 \not\lesssim \mathbb{0}$ and this case cannot occur.

- $\underline{[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]}$. Immediate by induction.

□

Lemma A.29. *If* $\Gamma, x : \forall\vec{\alpha}'.\tau' \vdash E : \forall\vec{\alpha}.\tau$*, then for every expression* E' *such that* $\Gamma \vdash E' : \forall\vec{\alpha}'.\tau'$*, we have* $\Gamma \vdash E\,[E'/x] : \forall\vec{\alpha}.\tau$*.*

Proof. By induction on E generalized over Γ.

- $\underline{E = c}$. Immediate since x does not appear in E.
- $\underline{E = y}$. If $x \neq y$ then the result is immediate. Otherwise, by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Var}}]$, we have $\Gamma, x : \forall\vec{\alpha}'.\tau' \vdash x : \forall\vec{\alpha}'.\tau'$ and by inversion of the typing rules, $\forall\vec{\alpha}'.\tau' \cong \forall\vec{\alpha}.\tau$. Since $E\,[E'/x] = E'$, the result follows.
- $\underline{E = \lambda^{\tau_1\to\tau_2} y.\, E''}$. By inversion of the typing rules, $\tau_1 \to \tau_2 \cong \forall\vec{\alpha}.\tau$, and $\Gamma, x : \forall\vec{\alpha}'.\tau', y : \tau_1 \vdash E'' : \tau_2$. By induction hypothesis, and $\Gamma, y : \tau_1 \vdash E''\,[E'/x] : \tau_2$, and the result follows by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Abstr}}]$ and $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{E = E_1\,E_2}$. By inversion of the typing rules, $\Gamma, x : \forall\vec{\alpha}'.\tau' \vdash E_1 : \tau_2 \to \tau_1$ and $\Gamma, x : \forall\vec{\alpha}'.\tau' \vdash E_2 : \tau_2$, where $\tau_1 \cong \forall\vec{\alpha}.\tau$. By IH, we have $\Gamma \vdash E_1\,[E'/x] : \tau_2 \to \tau_1$ and $\Gamma \vdash E_2\,[E'/x] : \tau_2$ and the result follows from $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{App}}]$ and $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{E = \pi_i\,E''}$. By inversion of the typing rules, $\Gamma, x : \forall\vec{\alpha}'.\tau' \vdash E'' : \tau_1 \times \tau_2$ where $\tau_i \cong \forall\vec{\alpha}.\tau$. By IH, we have $\Gamma \vdash E''\,[E'/x] : \tau_1 \times \tau_2$, and the result follows from $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Proj}}]$ and $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{E = (E_1, E_2)}$. By inversion of the typing rules, $\Gamma, x : \forall\vec{\alpha}'.\tau' \vdash E_1 : \tau_1$ and $\Gamma, x : \forall\vec{\alpha}'.\tau' \vdash E_2 : \tau_2$, where $\tau_1 \times \tau_2 \cong \forall\vec{\alpha}.\tau$. By IH, we have $\Gamma \vdash E_1\,[E'/x] : \tau_1$ and $\Gamma \vdash E_2\,[E'/x] : \tau_2$ and the result follows from $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Pair}}]$ and $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{E = \mathsf{let}\, y = E_1\, \mathsf{in}\, E_2}$. By inversion of the typing rules, $\Gamma, x : \forall\vec{\alpha}'.\tau' \vdash E_1 : \forall\vec{\beta}.\tau_1$ and $\Gamma, x : \forall\vec{\alpha}'.\tau', y : \forall\vec{\beta}.\tau_1 \vdash E_2 : \tau_2$, where $\tau_2 \cong \forall\vec{\alpha}.\tau$. By IH, we have $\Gamma \vdash E_1\,[E'/x] : \forall\vec{\beta}.\tau_1$ and $\Gamma, y : \forall\vec{\beta}.\tau_1 \vdash E_2\,[E'/x] : \tau_2$ and the result follows from $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Let}}]$ and $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{E = E''[\vec{t}]}$. By inversion of the typing rules, $\Gamma, x : \forall\vec{\alpha}'.\tau' \vdash E'' : \forall\vec{\beta}.\tau''$ where $\tau''\,[\vec{t}/\vec{\beta}] \cong \forall\vec{\alpha}.\tau$. By IH, we have $\Gamma \vdash E''\,[E'/x] : \forall\vec{\beta}.\tau''$, and the result follows from $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{TApp}}]$ and $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{E = \Lambda\vec{\beta}.\, E''}$. By inversion of the typing rules, $\Gamma, x : \forall\vec{\alpha}'.\tau' \vdash E'' : \tau''$ where $\forall\vec{\beta}.\tau'' \cong \forall\vec{\alpha}.\tau$. By IH, we have $\Gamma \vdash E''\,[E'/x] : \tau''$, and the result follows from $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{TAbstr}}]$ and $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{E = E''\langle \tau_1 \Rightarrow_p \tau_2\rangle}$. By inversion of the typing rules, $\Gamma, x : \forall\vec{\alpha}'.\tau' \vdash E'' : \tau_1$ and $\tau_2 \cong \forall\vec{\alpha}.\tau$. By IH, we have $\Gamma \vdash E''\,[E'/x] : \tau_1$, and the result follows from $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Cast+}}]$ (resp. $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Cast-}}]$) if $\tau_1 \lessapprox \tau_2$ (resp. $\tau_2 \lessapprox \tau_1$) and $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.

□

Lemma A.30. *If* $\Gamma \vdash \mathscr{E}\,[E] : \forall\vec{\alpha}.\tau$*, then* $\Gamma \vdash E : \forall\vec{\alpha}'.\tau'$ *and for every expression* E' *such that* $\Gamma \vdash E' : \forall\vec{\alpha}'.\tau'$*, we have* $\Gamma \vdash \mathscr{E}\,[E'] : \forall\vec{\alpha}.\tau$*.*

Proof. By induction on $\mathscr{E}$.

- $\underline{\mathscr{E} = [\,].}$ Immediate since $\mathscr{E}\,[E] = E$.
- $\underline{\mathscr{E} = E''\,\mathscr{E}'.}$ By inversion of the typing rules, $\Gamma \vdash E'' : \tau_1 \to \tau_2$, and $\Gamma \vdash \mathscr{E}'\,[E] : \tau_1$, and $\tau_2 \lesssim \forall\vec{\alpha}.\tau$. By induction hypothesis, $\Gamma \vdash E : \forall\vec{\alpha}'.\tau'$ and $\Gamma \vdash \mathscr{E}'\,[E'] : \tau_1$. Thus, by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{App}}]$ we have $\Gamma \vdash \mathscr{E}\,[E'] : \tau_2$ and the result follows by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{\mathscr{E} = \mathscr{E}'\,V.}$ By inversion of the typing rules, $\Gamma \vdash \mathscr{E}'\,[E] : \tau_1 \to \tau_2$, and $\Gamma \vdash V : \tau_1$, and $\tau_2 \lesssim \forall\vec{\alpha}.\tau$. By induction hypothesis, $\Gamma \vdash E : \forall\vec{\alpha}'.\tau'$ and $\Gamma \vdash \mathscr{E}'\,[E'] : \tau_1 \to \tau_2$. Thus, by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{App}}]$ we have $\Gamma \vdash \mathscr{E}\,[E'] : \tau_2$ and the result follows by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{\mathscr{E} = \mathscr{E}'[\vec{t}].}$ By inversion of the typing rules, $\Gamma \vdash \mathscr{E}'\,[E] : \forall\vec{\beta}.\tau''$, and $\tau''\,[\vec{t}/\vec{\beta}] \lesssim \forall\vec{\alpha}.\tau$. By induction hypothesis, $\Gamma \vdash E : \forall\vec{\alpha}'.\tau'$ and $\Gamma \vdash \mathscr{E}'\,[E'] : \forall\vec{\beta}.\tau''$. Thus, by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{TApp}}]$ we have $\Gamma \vdash \mathscr{E}\,[E'] : \tau''\,[\vec{t}/\vec{\beta}]$ and the result follows by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{\mathscr{E} = (E'', \mathscr{E}').}$ By inversion of the typing rules, $\Gamma \vdash E'' : \tau_1$, and $\Gamma \vdash \mathscr{E}'\,[E] : \tau_2$, and $\tau_1 \times \tau_2 \lesssim \forall\vec{\alpha}.\tau$. By induction hypothesis, $\Gamma \vdash E : \forall\vec{\alpha}'.\tau'$ and $\Gamma \vdash \mathscr{E}'\,[E'] : \tau_2$. Thus, by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Pair}}]$ we have $\Gamma \vdash \mathscr{E}\,[E'] : \tau_1 \times \tau_2$ and the result follows by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{\mathscr{E} = (\mathscr{E}', V).}$ By inversion of the typing rules, $\Gamma \vdash \mathscr{E}'\,[E] : \tau_1$, and $\Gamma \vdash V : \tau_2$, and $\tau_1 \times \tau_2 \lesssim \forall\vec{\alpha}.\tau$. By induction hypothesis, $\Gamma \vdash E : \forall\vec{\alpha}'.\tau'$ and $\Gamma \vdash \mathscr{E}'\,[E'] : \tau_1$. Thus, by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Proj}}]$ we have $\Gamma \vdash \mathscr{E}\,[E'] : \tau_1 \times \tau_2$ and the result follows by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{\mathscr{E} = \pi_i\,\mathscr{E}'.}$ By inversion of the typing rules, $\Gamma \vdash \mathscr{E}'\,[E] : \tau_1 \times \tau_2$, and $\tau_i \lesssim \forall\vec{\alpha}.\tau$. By induction hypothesis, $\Gamma \vdash E : \forall\vec{\alpha}'.\tau'$ and $\Gamma \vdash \mathscr{E}'\,[E'] : \tau_1 \times \tau_2$. Thus, by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Proj}}]$ we have $\Gamma \vdash \mathscr{E}\,[E'] : \tau_i$ and the result follows by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{\mathscr{E} = \mathsf{let}\,x = \mathscr{E}'\,\mathsf{in}\,E''.}$ By inversion of the typing rules, $\Gamma \vdash \mathscr{E}'\,[E] : \forall\vec{\beta}.\tau_1$, and $\Gamma, x : \forall\vec{\beta}.\tau_1 \vdash E'' : \tau_2$, and $\tau_2 \lesssim \forall\vec{\alpha}.\tau$. By induction hypothesis, $\Gamma \vdash E : \forall\vec{\alpha}'.\tau'$ and $\Gamma \vdash \mathscr{E}'\,[E'] : \forall\vec{\beta}.\tau_1$. Thus, by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Let}}]$ we have $\Gamma \vdash \mathscr{E}\,[E'] : \tau_2$ and the result follows by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.
- $\underline{\mathscr{E} = \mathscr{E}'\langle \tau_1 \Rightarrow_p \tau_2\rangle.}$ By inversion of the typing rules, $\Gamma \vdash \mathscr{E}'\,[E] : \tau_1$, and $\tau_2 \lesssim \forall\vec{\alpha}.\tau$. By induction hypothesis, $\Gamma \vdash E : \forall\vec{\alpha}'.\tau'$ and $\Gamma \vdash \mathscr{E}'\,[E'] : \tau_1$. Thus, by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Cast}+}]$ or $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Cast}-}]$ depending on the polarity of p, we have $\Gamma \vdash \mathscr{E}\,[E'] : \tau_2$ and the result follows by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Sub}}]$.

□

Lemma A.31. *If* $\Gamma \vdash E : \forall\vec{\alpha}.\tau$*, then for every static type substitution* $\theta : \mathscr{V}^{\alpha} \to \mathsf{GTypes}$ *such that* $\mathsf{dom}(\theta) \cap \vec{\alpha} = \emptyset$, $\Gamma\theta \vdash E\theta : \forall\vec{\alpha}.\tau\theta$.

Proof. By induction on the derivation $\Gamma \vdash E : \tau$ and case analysis over the last rule used.

- $\underline{[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Cst}}]}$. Immediate since $b_c\theta = b_c$ and $c\theta = c$.
- $\underline{[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Var}}]}$. We have $E = x$, $\Gamma \vdash x : \forall\vec{\alpha}.\tau$, and $\Gamma(x) = \forall\vec{\alpha}.\tau$. This gives $(\Gamma\theta)(x) = \forall\vec{\alpha}.\tau\theta$. And since $x\theta = x$, the result follows by $[\mathrm{T}^{\langle \mathsf{ST}\rangle}_{\mathsf{Var}}]$.

- $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cast}+}]$, $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cast}-}]$, $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]$. Follow immediately from Proposition 6.7.
- The other cases are immediate by application of the induction hypothesis.

□

Lemma A.32 (Subject reduction)**.** *For every term $E, E' \in \mathsf{Terms}^{\langle \mathsf{ST} \rangle}$, if $\emptyset \vdash E : \forall \vec{\alpha}.\tau$ and $E \rightsquigarrow E'$ then $\emptyset \vdash E' : \forall \vec{\alpha}.\tau$.*

Proof. By induction on E and case analysis on the reduction rule used for $E \rightsquigarrow E'$.

- $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cons}}]$. $V\langle \tau_1 \Rightarrow_p \tau_2 \rangle \rightsquigarrow V\langle \tau_1 \wedge \mathsf{cons}(V) \Rightarrow_p \tau_2 \wedge \mathsf{cons}(V) \rangle$. Suppose that p is positive. The negative case is proven similarly. By inversion of the typing rules, $\emptyset \vdash V : \tau_1$, $\tau_2 \cong \forall \vec{\alpha}.\tau$, and $\tau_1 \lessapprox \tau_2$. By Lemma 6.27, we have $\emptyset \vdash V : \tau_1 \wedge \mathsf{cons}(V)$. Moreover, since $\tau_1 \lessapprox \tau_2$, we have $\tau_1 \wedge \mathsf{cons}(V) \lessapprox \tau_2 \wedge \mathsf{cons}(V)$. Finally, by $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cast}+}]$, it holds that $\emptyset \vdash V\langle \tau_1 \wedge \mathsf{cons}(V) \Rightarrow_p \tau_2 \wedge \mathsf{cons}(V) \rangle : \tau_2 \wedge \mathsf{cons}(V)$ and the result follows by subtyping and $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]$.
- $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{SimpI}}]$. $c\langle \tau_1 \Rightarrow_p \tau_2 \rangle \rightsquigarrow c$ and $b_c \wedge ? \cong \tau_2$. By inversion of the typing rules, $\tau_2 \cong \forall \vec{\alpha}.\tau$. By transitivity, this yields $b_c \wedge ? \cong \forall \vec{\alpha}.\tau$. By $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cst}}]$ and $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]$ we deduce $\emptyset \vdash c : \forall \vec{\alpha}.\tau$.
- $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{App}}]$. $(\lambda^{\tau_1 \to \tau_2} x.\, E)\, V \rightsquigarrow E\,[V/x]$. By inversion of the typing rules, $\tau_1 \to \tau_2 \cong \tau_1' \to \tau_2'$ where $\emptyset \vdash V : \tau_1'$, $\tau_2' \cong \forall \vec{\alpha}.\tau$, and $x : \tau_1 \vdash E : \tau_2$. By Proposition 6.17, we have $\tau_1' \cong \tau_1$, hence $\emptyset \vdash V : \tau_1$ by $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]$. We also have by Definition 6.15 that $\tau_1 \to \tau_2 \mathbin{\tilde{\circ}} \tau_1 = \tau_1$. Moreover, since $\tau_1 \to \tau_2 \cong \tau_1 \to \tau_2'$, by Proposition 6.18, we have $\tau_2 \cong \tau_2'$. Hence $\tau_2 \cong \forall \vec{\alpha}.\tau$ by transitivity of subtyping. By Lemma 6.33, we deduce that $\emptyset \vdash E\,[V/x] : \tau_2$ and the result follows by $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]$.
- $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Proj}}]$. $\pi_i\,(V_1, V_2) \rightsquigarrow V_i$. By inversion of the typing rules, $\emptyset \vdash (V_1, V_2) : \tau_1 \times \tau_2$ where $\tau_i \cong \forall \vec{\alpha}.\tau$. By inversion of $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Pair}}]$, we have $\emptyset \vdash V_i : \tau_i$ and the result follows by $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]$.
- $[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CApp}}]$. $(V\langle \tau_1 \Rightarrow_p \tau_2 \rangle)\, V' \rightsquigarrow (V\,(V'\langle \widetilde{\mathsf{dom}}(\tau_2) \wedge \sigma \Rightarrow_{\overline{p}} \widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma \rangle))\langle \tau_1 \mathbin{\tilde{\circ}} (\sigma \wedge \widetilde{\mathsf{dom}}(\tau_1)) \Rightarrow_p \tau_2 \mathbin{\tilde{\circ}} (\sigma \wedge \widetilde{\mathsf{dom}}(\tau_1)) \rangle$ where $\sigma = \mathsf{type}(V')$. Suppose that p is positive. The negative case is proven similarly. By inversion of the typing rules, we have:

  $$① \quad \emptyset \vdash V : \tau_1 \qquad ② \quad \tau_1 \lessapprox \tau_2 \qquad ③ \quad \emptyset \vdash V' : \sigma'$$

  $$④ \quad \tau_2 \cong \sigma' \to \tau_2' \qquad ⑤ \quad \tau_2' \cong \forall \vec{\alpha}.\tau$$

  By Lemma 6.26 and ③, we deduce that $\sigma \cong \sigma'$. By Proposition 6.17 and ④, we deduce $\sigma' \cong \widetilde{\mathsf{dom}}(\tau_2)$, hence $\sigma \cong \widetilde{\mathsf{dom}}(\tau_2)$. Thus, by Lemma 6.26, we have $\emptyset \vdash V' : \widetilde{\mathsf{dom}}(\tau_2) \wedge \sigma$ Ⓐ.

  Proposition 6.20 and ② ensure $\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma \lessapprox \widetilde{\mathsf{dom}}(\tau_2) \wedge \sigma$. Together with Ⓐ, this proves $\emptyset \vdash V'\langle \widetilde{\mathsf{dom}}(\tau_2) \wedge \sigma \Rightarrow_{\overline{p}} \widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma \rangle : \widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma$ Ⓑ.

  By Proposition 6.18, we have $\tau_1 \cong (\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma) \to \tau_1 \mathbin{\tilde{\circ}} (\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma)$. By $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]$ and ①, along with $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{App}}]$ and Ⓑ, we deduce that $\emptyset \vdash V\,(V'\langle \widetilde{\mathsf{dom}}(\tau_2) \wedge \sigma \Rightarrow_{\overline{p}} \widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma \rangle) : \tau_1 \mathbin{\tilde{\circ}} (\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma)$ Ⓒ.

  By Proposition 6.21, we deduce that $\tau_1 \mathbin{\tilde{\circ}} (\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma) \lessapprox \tau_2 \mathbin{\tilde{\circ}} (\widetilde{\mathsf{dom}}(\tau_1) \wedge \sigma)$. Along

with Ⓒ, this entails $\emptyset : E' : \tau_2 \mathbin{\tilde{\circ}} (\widetilde{\mathrm{dom}}(\tau_1) \wedge \sigma)$. Finally, ④ and Proposition 6.18 prove $\tau_2 \mathbin{\tilde{\circ}} (\widetilde{\mathrm{dom}}(\tau_1) \wedge \sigma) \mathrel{\tilde{\leq}} \tau_2'$ and the result follows from ⑤ and $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]$.

- $\underline{[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{CProj}}]}$. $\pi_i\,(V\langle \tau_1 \Rightarrow_p \tau_2 \rangle) \rightsquigarrow (\pi_i\,V)\langle \widetilde{\pi_i}(\tau_1) \Rightarrow_p \widetilde{\pi_i}(\tau_2) \rangle$. Suppose that p is positive. The negative case is proven similarly. By inversion of the typing rules, we have:

$$① \quad \emptyset \vdash V : \tau_1 \qquad ② \quad \tau_1 \mathrel{\tilde{\leq}} \tau_2 \qquad ③ \quad \tau_2 \mathrel{\tilde{\leq}} \tau_1' \times \tau_2' \qquad ④ \quad \tau_i' \mathrel{\tilde{\leq}} \forall \vec{\alpha}.\tau$$

  By Proposition 6.19, we have $\tau_1 \mathrel{\tilde{\leq}} \widetilde{\pi_1}(\tau_1) \times \widetilde{\pi_2}(\tau_1)$. By $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]$ and ①, this entails $\emptyset \vdash V : \widetilde{\pi_1}(\tau_1) \times \widetilde{\pi_2}(\tau_1)$. By $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Proj}}]$, we deduce $\emptyset \vdash \pi_i\,V : \widetilde{\pi_i}(\tau_1)$ Ⓐ.
  By Proposition 6.22 and ②, we have $\widetilde{\pi_i}(\tau_1) \mathrel{\tilde{\leq}} \widetilde{\pi_i}(\tau_2)$ Ⓑ. The result follows by an application of $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Cast}+}]$ with Ⓐ and Ⓑ, and an application of $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]$ with ④.

- $\underline{[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{TApp}}]}$. $(\Lambda \vec{\beta}.\,E)[\vec{t}] \rightsquigarrow E\,[\vec{t}/\vec{\beta}]$. By inversion of the typing rules, $\emptyset \vdash \Lambda \vec{\beta}.\,E : \forall \vec{\beta}.\tau'$ where $\emptyset \vdash E : \tau'$ and $\tau'\,[\vec{t}/\vec{\beta}] \mathrel{\tilde{\leq}} \forall \vec{\alpha}.\tau$. By Lemma 6.35, we have $\emptyset \vdash E\,[\vec{t}/\vec{\beta}] : \tau'\,[\vec{t}/\vec{\beta}]$. The result follows by subtyping.

- $\underline{[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Let}}]}$. $\mathsf{let}\, x = V\, \mathsf{in}\, E \rightsquigarrow E\,[V/x]$. By inversion of the typing rules, $\emptyset \vdash V : \forall \vec{\alpha}_1.\tau_1$ and $x : \forall \vec{\alpha}_1.\tau_1 \vdash E : \tau_2$ where $\tau_2 \mathrel{\tilde{\leq}} \forall \vec{\alpha}.\tau$. By Lemma 6.33, we deduce that $\emptyset \vdash E\,[V/x] : \tau_2$, hence the result by $[\mathrm{T}^{\langle \mathsf{ST} \rangle}_{\mathsf{Sub}}]$.

- $\underline{[\mathrm{R}^{\langle \mathsf{ST} \rangle}_{\mathsf{Ctx}}]}$. $\mathscr{E}\,[E] \rightsquigarrow \mathscr{E}\,[E']$ where $E \rightsquigarrow E'$. Immediate consequence of the induction hypothesis and Lemma 6.34.

□

A.2. Denotational semantics

A.2.1. Second approach to the semantics of the functional core calculus

Theorem A.33 (Conservativity of the semantics)**.** *For every term* $\mathbf{e} \in \mathsf{Terms}$*, every environment* $\rho \in \mathsf{Envs}$*,*

$$d \in [\![\mathbf{e}]\!]_\rho \implies I(d) \in [\![\mathbf{e}]\!]^F_{I(\rho)}$$

Proof. The proof is done by structural induction on $\mathbf{e} \in \mathsf{Terms}$, where the induction hypothesis is generalized over ρ.

- $\underline{\mathbf{e} = c.}$ Immediate since $[\![c]\!]_\rho = \{c\}$, $[\![c]\!]^{\mathrm{F}}_{I(\rho)} = \{c\}$, and $I(c) = c$.

- $\underline{\mathbf{e} = x.}$ Immediate by definition of $I(\rho)$: $[\![x]\!]_\rho = \{\rho(x)\}$ and $[\![x]\!]^{\mathrm{F}}_{I(\rho)} = I(\rho)(x) = \{I(\rho(x))\}$.

- $\underline{\mathbf{e} = \lambda x{:}t.\,\mathbf{e}'.}$ Let $R \in [\![\lambda x{:}t.\,\mathbf{e}']\!]_\rho$ and let $(S, \partial) \in I(R)$. By Definition 10.6, there exists $(d_0, \partial_0) \in R$ such that $S = \{I(d_0)\}$ and $\partial = I(\partial_0)$. We distinguish two cases:

  1. $d_0 \in [\![t]\!]$. By Lemma 10.8, $S \subseteq [\![t]\!]^{\mathrm{F}}$. And by Definition 9.5, $\partial_0 \in [\![\mathbf{e}']\!]_{\rho, x \mapsto d_0}$. Thus, by induction hypothesis, $\partial = I(\partial_0) \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{I(\rho), x \mapsto I(d_0)}$.
  2. $d_0 \notin [\![t]\!]$. By Lemma 10.8, $S \subseteq [\![\neg t]\!]^{\mathrm{F}}$. And by Definition 9.5, $\partial_0 = \Omega$. Thus,

$\partial = \Omega$, and this proves that $I(R) \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{I(\rho)}$.

- $\underline{\mathbf{e} = \mathbf{e}_1\,\mathbf{e}_2.}$ Let $\partial \in [\![\mathbf{e}_1\,\mathbf{e}_2]\!]_\rho$. We distinguish four cases:
  1. There exists $R \in [\![\mathbf{e}_1]\!]_\rho$ and $d \in [\![\mathbf{e}_2]\!]_\rho$ such that $(d, \partial) \in R$. By induction hypothesis, $I(R) \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{I(\rho)}$ and $I(d) \in [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{I(\rho)}$. By Definition 10.6, $(I(d), I(\partial)) \in I(R)$. Thus $I(\partial) \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{I(\rho)}$.
  2. $\partial = \Omega$ where $\Omega \in [\![\mathbf{e}_1]\!]_\rho$. By induction hypothesis, $\Omega \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{I(\rho)}$, thus $\Omega \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{I(\rho)}$.
  3. $\partial = \Omega$ where $\Omega \in [\![\mathbf{e}_2]\!]_\rho$ and $[\![\mathbf{e}_1]\!]_\rho \neq \emptyset$. By induction hypothesis, $\Omega \in [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{I(\rho)}$ and $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_{I(\rho)} \neq \emptyset$. Thus $\Omega \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{I(\rho)}$.
  4. $\partial = \Omega$ where there exists $d \in [\![\mathbf{e}_1]\!]_\rho$ such that $d \notin \mathscr{P}_f(\mathscr{D} \times \mathscr{D}_\Omega)$ and $[\![\mathbf{e}_2]\!]_\rho \neq \emptyset$. By induction hypothesis, $I(d) \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{I(\rho)}$. By Definition 10.6, $I(d) \notin \mathscr{P}_f(\mathscr{F} \times \mathscr{D}^{\mathrm{F}}_\Omega)$. And by induction hypothesis, $[\![\mathbf{e}_2]\!]^{\mathrm{F}}_{I(\rho)} \neq \emptyset$. Thus $\Omega \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{I(\rho)}$.
- $\underline{\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2).}$ Let $\partial \in [\![(\mathbf{e}_1, \mathbf{e}_2)]\!]_\rho$. We distinguish three cases:
  1. There exists $d_1 \in [\![\mathbf{e}_1]\!]_\rho$ and $d_2 \in [\![\mathbf{e}_2]\!]_\rho$ such that $\partial = (d_1, d_2)$. By induction hypothesis, $I(d_1) \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{I(\rho)}$ and $I(d_2) \in [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{I(\rho)}$. By Definition 10.6, $I(\partial) = (I(d_1), I(d_2))$. Thus $I(\partial) \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{I(\rho)}$.
  2. $\partial = \Omega$ where $\Omega \in [\![\mathbf{e}_1]\!]_\rho$. By induction hypothesis, $\Omega \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{I(\rho)}$, thus $\Omega \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{I(\rho)}$.
  3. $\partial = \Omega$ where $\Omega \in [\![\mathbf{e}_2]\!]_\rho$ and $[\![\mathbf{e}_1]\!]_\rho \neq \emptyset$. By induction hypothesis, $\Omega \in [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{I(\rho)}$ and $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_{I(\rho)} \neq \emptyset$. Thus $\Omega \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{I(\rho)}$.
- $\underline{\mathbf{e} = \pi_i\,\mathbf{e}'.}$ Let $\partial \in [\![\pi_i\,\mathbf{e}']\!]_\rho$. We distinguish three cases:
  1. There exists $(d_1, d_2) \in [\![\mathbf{e}']\!]_\rho$ such that $\partial = d_i$. By induction hypothesis, $I((d_1, d_2)) \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{I(\rho)}$. By Definition 10.6, $I((d_1, d_2)) = (I(d_1), I(d_2))$. Thus $I(\partial) = I(d_i) \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{I(\rho)}$.
  2. $\partial = \Omega$ where $\Omega \in [\![\mathbf{e}']\!]_\rho$. By induction hypothesis, $\Omega \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{I(\rho)}$, thus $\Omega \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{I(\rho)}$.
  3. $\partial = \Omega$ where there exists $d \in [\![\mathbf{e}']\!]_\rho$ such that $d \notin \mathscr{D} \times \mathscr{D}$. By induction hypothesis, $I(d) \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{I(\rho)}$. By Definition 10.6, $I(d) \notin \mathscr{D}^{\mathrm{F}} \times \mathscr{D}^{\mathrm{F}}$. Thus $\Omega \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{I(\rho)}$.

□

Theorem A.34 (Type soundness for λ_{F})**.** *For every type environment* $\Gamma \in \mathsf{TEnvs}$ *and every term* $\mathbf{e} \in \mathsf{Terms}$*, if* $\Gamma \vdash \mathbf{e} : t$ *then for every* $\rho \in [\![\Gamma]\!]^F$, $[\![\mathbf{e}]\!]^F_\rho \subseteq [\![t]\!]^F$.

Proof. The proof is done by structural induction on $\mathbf{e} \in \mathsf{Terms}$, supposing $\Gamma \vdash \mathbf{e} : t$. The induction hypothesis is generalized over Γ.

- $\underline{\mathbf{e} = c.}$ By inversion of the typing rules, $b_c \leq t$, therefore $c \in [\![t]\!]^{\mathrm{F}}$.
- $\underline{\mathbf{e} = x.}$ By inversion of the typing rules, $\Gamma(x) = t'$ where $t' \leq t$. By application of

Definition 10.13, $\rho(x) \subseteq [\![t']\!]^{\mathrm{F}} \subseteq [\![t]\!]^{\mathrm{F}}$.

- $\underline{\mathbf{e} = \lambda x{:}t_x.\, \mathbf{e}'.}$ By inversion of the typing rules, $\Gamma, x : t_x \vdash \mathbf{e}' : t_{\mathbf{e}}$ and $t_x \to t_{\mathbf{e}} \leq t$. Let $R \in [\![\lambda x{:}t_x.\, \mathbf{e}']\!]^{\mathrm{F}}_{\rho}$, and let $(S, \partial) \in R$ such that $S \cap [\![t_x]\!]^{\mathrm{F}} \neq \emptyset$. By definition of $[\![\lambda x{:}t_x.\, \mathbf{e}']\!]^{\mathrm{F}}_{\rho}$, either $S \subseteq [\![t_x]\!]^{\mathrm{F}}$ or $S \subseteq [\![\neg t_x]\!]^{\mathrm{F}}$. Since $S \cap [\![t_x]\!]^{\mathrm{F}} \neq \emptyset$, this ensures that $S \subseteq [\![t_x]\!]^{\mathrm{F}}$. Now, by definition, $\partial \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho, x \mapsto S}$. Since $(\rho, x \mapsto S) \in [\![\Gamma, x : t_x]\!]^{\mathrm{F}}$, we can apply the induction hypothesis to deduce that $[\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho, x \mapsto S} \subseteq [\![t_{\mathbf{e}}]\!]^{\mathrm{F}}$. Therefore, $\partial \in [\![t_{\mathbf{e}}]\!]^{\mathrm{F}}$, and this proves that $R \in [\![t_x \to t_{\mathbf{e}}]\!]^{\mathrm{F}}$, hence the result.

- $\underline{\mathbf{e} = \mathbf{e}_1\, \mathbf{e}_2.}$ By inversion of the typing rules, $\Gamma \vdash \mathbf{e}_1 : t_1 \to t_2$, $\Gamma \vdash \mathbf{e}_2 : t_1$, and $t_2 \leq t$. By induction hypothesis, $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho} \subseteq [\![t_1 \to t_2]\!]^{\mathrm{F}}$ and $[\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho} \subseteq [\![t_1]\!]^{\mathrm{F}}$. Therefore, $\Omega \notin [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho} \cup [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho}$. Moreover, $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho} \subseteq \mathscr{P}_f(\mathscr{F} \times \mathscr{D}^{\mathrm{F}}_{\Omega})$. Thus, we deduce that $\Omega^{\rho}_{\mathbf{e}_1\, \mathbf{e}_2} = \emptyset$. Now let $\partial \in [\![\mathbf{e}_1\, \mathbf{e}_2]\!]^{\mathrm{F}}_{\rho}$. By inversion, there exists $R \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho}$ and $S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho}$ such that $(S, \partial) \in R$. The induction hypothesis yields that $R \in [\![t_1 \to t_2]\!]^{\mathrm{F}}$ and $S \subseteq [\![t_1]\!]^{\mathrm{F}}$. Thus, by definition of $[\![t_1 \to t_2]\!]^{\mathrm{F}}$, this yields that $\partial \in [\![t_2]\!]^{\mathrm{F}}$, hence the result.

- $\underline{\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2).}$ By inversion of the typing rules, $\Gamma \vdash \mathbf{e}_1 : t_1$ and $\Gamma \vdash \mathbf{e}_2 : t_2$, and $t_1 \times t_2 \leq t$. Moreover, by induction hypothesis, $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho} \subseteq [\![t_1]\!]^{\mathrm{F}}$ and $[\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho} \subseteq [\![t_2]\!]^{\mathrm{F}}$. We deduce from this fact that $\Omega \notin [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho} \cup [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho}$, thus $\Omega^{\rho}_{(\mathbf{e}_1, \mathbf{e}_2)} = \emptyset$. This also yields that, by definition, $[\![(\mathbf{e}_1, \mathbf{e}_2)]\!]^{\mathrm{F}}_{\rho} = [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho} \times [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho} \subseteq [\![t_1]\!]^{\mathrm{F}} \times [\![t_2]\!]^{\mathrm{F}}$, hence the result.

- $\underline{\mathbf{e} = \pi_i\, \mathbf{e}'.}$ By inversion of the typing rules, $\Gamma \vdash \mathbf{e}' : t_1 \times t_2$ and $t_i \leq t$. Moreover, by induction hypothesis, $[\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho} \subseteq [\![t_1 \times t_2]\!]^{\mathrm{F}}$. Thus, this ensures that $[\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho} \subseteq \mathscr{D}^{\mathrm{F}} \times \mathscr{D}^{\mathrm{F}}$ and that $\Omega^{\rho}_{\pi_i\, \mathbf{e}'} = \emptyset$.
  Now let $d \in [\![\pi_i\, \mathbf{e}']\!]^{\mathrm{F}}_{\rho}$. By inversion, there exists $(d_1, d_2) \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho}$ such that $d = d_i$. The induction hypothesis yields that $d_i \in [\![t_i]\!]^{\mathrm{F}}$, and the result follows by subtyping.

□

Lemma A.35. *For every term* $\mathbf{e} \in \mathsf{Terms}$, $x \in \mathsf{Vars}$, $\rho \in \mathsf{Envs}^F$, *and* $S_1, S_2 \in \mathscr{P}(\mathscr{D}^F)$, *if* $S_1 \subseteq S_2$ *then* $[\![\mathbf{e}]\!]^F_{\rho, x \mapsto S_1} \subseteq [\![\mathbf{e}]\!]^F_{\rho, x \mapsto S_2}$

Proof. The proof is done by structural induction on $\mathbf{e} \in \mathsf{Terms}$, generalized over ρ.

- $\underline{\mathbf{e} = c.}$ Immediate since x does not appear in $\mathbf{e}$.

- $\underline{\mathbf{e} = y.}$ If $y \neq x$ then the result is immediate. Otherwise, $[\![x]\!]^{\mathrm{F}}_{\rho, x \mapsto S_1} = S_1 \subseteq S_2 = [\![x]\!]^{\mathrm{F}}_{\rho, x \mapsto S_2}$ which gives the result.

- $\underline{\mathbf{e} = \lambda y{:}t.\, \mathbf{e}'.}$ Let $R \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho, x \mapsto S_1}$. Let $(S, \partial) \in R$. If $S \subseteq [\![t]\!]^{\mathrm{F}}$ then $\partial \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho, x \mapsto S_1, y \mapsto S}$. By generalized induction hypothesis, $\partial \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho, x \mapsto S_2, y \mapsto S}$. Moreover, if $S \not\subseteq [\![t]\!]^{\mathrm{F}}$ then $\partial = \Omega$ independently of S_1 and S_2. Therefore, $R \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho, x \mapsto S_2}$.

- $\underline{\mathbf{e} = \mathbf{e}_1\, \mathbf{e}_2.}$ Let $\partial \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho, x \mapsto S_1}$. We distinguish four cases.
  1. There exists $R \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x \mapsto S_1}$ and $S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho, x \mapsto S_1}$ such that $(S, \partial) \in R$. By induction hypothesis, $R \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x \mapsto S_2}$ and $S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho, x \mapsto S_2}$, thus $\partial \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho, x \mapsto S_2}$.
  2. $\partial = \Omega$ and $\Omega \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x \mapsto S_1}$. By induction hypothesis, $\Omega \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x \mapsto S_2}$ and the result follows.

3. $\partial = \Omega$ and $\Omega \in [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho,x\mapsto S_1}$ and $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho,x\mapsto S_1} \neq \emptyset$. By induction hypothesis, $\Omega \in [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho,x\mapsto S_2}$ and $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho,x\mapsto S_2} \neq \emptyset$. Thus, the result follows.
4. $\partial = \Omega$ and there exists $d \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho,x\mapsto S_1}$ such that $d \notin \mathscr{P}_f(\mathscr{F} \times \mathscr{D}^{\mathrm{F}}_{\Omega})$ and $[\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho,x\mapsto S_1} \neq \emptyset$. By induction hypothesis, $d \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho,x\mapsto S_2}$ and $[\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho,x\mapsto S_2} \neq \emptyset$. This yields the result.

- $\underline{\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2).}$ Let $\partial \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho,x\mapsto S_1}$. We distinguish three cases.
  1. There exists $d_1 \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho,x\mapsto S_1}$ and $d_2 \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho,x\mapsto S_1}$ such that $\partial = (d_1, d_2)$. By induction hypothesis, $d_1 \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho,x\mapsto S_2}$ and $d_2 \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho,x\mapsto S_2}$, thus $\partial \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho,x\mapsto S_2}$.
  2. $\partial = \Omega$ and $\Omega \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho,x\mapsto S_1}$. By induction hypothesis, $\Omega \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho,x\mapsto S_2}$ and the result follows.
  3. $\partial = \Omega$ and $\Omega \in [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho,x\mapsto S_1}$ and $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho,x\mapsto S_1} \neq \emptyset$. By induction hypothesis, $\Omega \in [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho,x\mapsto S_2}$ and $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho,x\mapsto S_2} \neq \emptyset$. Thus, the result follows.
- $\underline{\mathbf{e} = \pi_i\, \mathbf{e}'.}$ Let $\partial \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho,x\mapsto S_1}$. We distinguish three cases.
  1. There exists $(d_1, d_2) \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho,x\mapsto S_1}$ such that $\partial = d_i$. By induction hypothesis, $(d_1, d_2) \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho,x\mapsto S_2}$ thus $\partial \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho,x\mapsto S_2}$.
  2. $\partial = \Omega$ and $\Omega \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho,x\mapsto S_1}$. By induction hypothesis, $\Omega \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho,x\mapsto S_2}$ and the result follows.
  3. $\partial = \Omega$ and there exists $d \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho,x\mapsto S_1}$ such that $d \notin \mathscr{D}^{\mathrm{F}} \times \mathscr{D}^{\mathrm{F}}$. By induction hypothesis, $d \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho,x\mapsto S_2}$, which yields the result.

□

Lemma A.36. *For every term* $\mathbf{e} \in \mathsf{Terms}$, $\mathbf{v} \in \mathsf{Values}$, $x \in \mathsf{Vars}$, $\rho \in \mathsf{Envs}^F$,

$$[\![\mathbf{e}\,[\mathbf{v}/x]]\!]^F_\rho = \bigcup_{S \in \mathscr{P}_f([\![\mathbf{v}]\!]^F_\rho)} [\![\mathbf{e}]\!]^F_{\rho,x\mapsto S}$$

Proof. The proof is done by structural induction on $\mathbf{e} \in \mathsf{Terms}$.

- $\underline{\mathbf{e} = c.}$ Immediate since x does not appear in c.
- $\underline{\mathbf{e} = y.}$ If $y \neq x$ the result is immediate. Otherwise, if $y = x$ we have $[\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{F}}_\rho = [\![\mathbf{v}]\!]^{\mathrm{F}}_\rho$ and we reason by double inclusion.
  - Consider $d \in [\![\mathbf{v}]\!]^{\mathrm{F}}_\rho$. Then we immediately have the result taking $S = \{d\}$: $[\![x]\!]^{\mathrm{F}}_{\rho,x\mapsto\{d\}} = \{d\}$.
  - Let $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{F}}_\rho)$ and consider $d \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho,x\mapsto S}$. Since $[\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho,x\mapsto S} = S$, we have $d \in S$. And since $S \subseteq [\![\mathbf{v}]\!]^{\mathrm{F}}_\rho$, $d \in [\![\mathbf{v}]\!]^{\mathrm{F}}_\rho$.
- $\underline{\mathbf{e} = \lambda y{:}t.\,\mathbf{e}'.}$ By definition, $[\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{F}}_\rho = [\![\lambda y{:}t.\,(\mathbf{e}'\,[\mathbf{v}/x])]\!]^{\mathrm{F}}_\rho$. We proceed by double inclusion.
  - Let $R \in [\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{F}}_\rho$, and let us write $R = \{(S_i, \partial_i) \mid i \in I\}$. Let $i \in I$. If $S_i \subseteq [\![t]\!]^{\mathrm{F}}$, then by definition this means that $\partial_i \in [\![\mathbf{e}'\,[\mathbf{v}/x]]\!]^{\mathrm{F}}_{\rho,y\mapsto S_i}$. By induction

hypothesis, there exists $S_i^{\mathrm{v}} \subseteq [\![\mathbf{v}]\!]_\rho^{\mathrm{F}}$ such that $\partial_i \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho, y\mapsto S_i, x\mapsto S_i^{\mathrm{v}}}$. Moreover, if $S_i \not\subseteq [\![t]\!]^{\mathrm{F}}$ then $\partial = \Omega$ by definition. So in this case, we just pose $S_i^{\mathrm{v}} = \emptyset$.
Now consider $S^{\mathrm{v}} = \bigcup_{i\in I} S_i^{\mathrm{v}}$. It is immediate that $S^{\mathrm{v}} \in \mathscr{P}_f([\![\mathbf{v}]\!]_\rho^{\mathrm{F}})$. Moreover, by Lemma A.35, we deduce that $\forall i \in I$, if $S_i \subseteq [\![t]\!]^{\mathrm{F}}$, then $\partial_i \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho, y\mapsto S_i, x\mapsto S^{\mathrm{v}}}$. Thus, this proves that $R \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho, x\mapsto S^{\mathrm{v}}}$.

- Let $S \in \mathscr{P}_f([\![\mathbf{v}]\!]_\rho^{\mathrm{F}})$ and $R \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho, x\mapsto S}$. Let $(S_i, \partial_i) \in R$. Note that if $S_i \not\subseteq [\![t]\!]^{\mathrm{F}}$ then $\partial = \Omega$ by definition, so this part is immediate. Now suppose that $S_i \subseteq [\![t]\!]^{\mathrm{F}}$. By definition, $\partial \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho, x\mapsto S, y\mapsto S_i}$. By induction hypothesis, this yields $\partial \in [\![\mathbf{e}'\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_{\rho, y\mapsto S_i}$ and thus $R \in [\![\mathbf{e}\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho$.

- $\underline{\mathbf{e} = \mathbf{e}_1\,\mathbf{e}_2}$. We prove the result by double inclusion. Let $\partial \in [\![\mathbf{e}\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho$. We distinguish several cases.

  1. There exists $R \in [\![\mathbf{e}_1\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho$ and $S \in \mathscr{P}_f([\![\mathbf{e}_2\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho)$ such that $(S, \partial) \in R$. By induction hypothesis, we deduce that there exists $S_1, S_2 \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{F}}_\rho)$ such that $R \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x\mapsto S_1}$ and $S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho, x\mapsto S_2}$. Taking $S = S_1 \cup S_2$ and applying Lemma A.35 yields that $R \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x\mapsto S}$ and $S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho, x\mapsto S}$, thus $\partial \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho, x\mapsto S}$.
  2. $\partial = \Omega$ where $\Omega \in [\![\mathbf{e}_1\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho$. By induction hypothesis, there exists $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{F}}_\rho)$ such that $\Omega \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x\mapsto S}$ and the result follows.
  3. $\partial = \Omega$ where $\Omega \in [\![\mathbf{e}_2\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho$ and $[\![\mathbf{e}_1\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho \neq \emptyset$. By induction hypothesis, there exists $S_1, S_2 \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{F}}_\rho)$ such that $\Omega \in [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho, x\mapsto S_1}$ and $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x\mapsto S_2} \neq \emptyset$. Taking $S = S_1 \cup S_2$ and applying Lemma A.35 yields that $\Omega \in [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho, x\mapsto S}$ and $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x\mapsto S} \neq \emptyset$, hence the result.
  4. $\partial = \Omega$ where there exists $d \in [\![\mathbf{e}_1\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho$ such that $d \notin \mathscr{P}_f(\mathscr{F} \times \mathscr{D}^{\mathrm{F}}_\Omega)$ and $[\![\mathbf{e}_2\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho \neq \emptyset$. By induction hypothesis, there exists $S_1, S_2 \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{F}}_\rho)$ such that $d \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x\mapsto S_1}$ and $[\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho, x\mapsto S_2} \neq \emptyset$. Taking $S = S_1 \cup S_2$ and applying Lemma A.35 yields that $d \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x\mapsto S}$ and $[\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho, x\mapsto S} \neq \emptyset$. Hence the result.

  The same reasoning proves the other inclusion.

- $\underline{\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2)}$. We prove the result by double inclusion. Let $\partial \in [\![\mathbf{e}\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho$. We distinguish three cases.

  1. $\partial = (d_1, d_2)$ where $\forall i \in \{1, 2\}$, $d_i \in [\![\mathbf{e}_i\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho$. By induction hypothesis, $\forall i \in \{1, 2\}$, there exists $S_i \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{F}}_\rho)$ such that $d_i \in [\![\mathbf{e}_i]\!]^{\mathrm{F}}_{\rho, x\mapsto S_i}$. Taking $S = S_1 \cup S_2$ and applying Lemma A.35 yields that $\forall i \in \{1, 2\}$, $d_i \in [\![\mathbf{e}_i]\!]^{\mathrm{F}}_{\rho, x\mapsto S}$. Thus, $\partial = (d_1, d_2) \in [\![\mathbf{e}]\!]^{\mathrm{F}}_{\rho, x\mapsto S}$.
  2. $\partial = \Omega$ where $\Omega \in [\![\mathbf{e}_1\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho$. By induction hypothesis, $\Omega \in [\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x\mapsto S}$ for some $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{F}}_\rho)$ and the result follows.
  3. $\partial = \Omega$ where $\Omega \in [\![\mathbf{e}_2\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho$ and $[\![\mathbf{e}_1\,[\mathbf{v}/x]\,]\!]^{\mathrm{F}}_\rho \neq \emptyset$. By induction hypothesis, there exists $S_1, S_2 \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{F}}_\rho)$ such that $\Omega \in [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho, x\mapsto S_1}$ and $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x\mapsto S_2} \neq \emptyset$. Taking $S = S_1 \cup S_2$ and applying Lemma A.35 yields that $\Omega \in [\![\mathbf{e}_2]\!]^{\mathrm{F}}_{\rho, x\mapsto S}$ and $[\![\mathbf{e}_1]\!]^{\mathrm{F}}_{\rho, x\mapsto S} \neq \emptyset$, hence the result.

  The second inclusion is proven using the same reasoning.

- $\underline{\mathbf{e} = \pi_i\, \mathbf{e}'.}$ We prove the result by double inclusion. Let $\partial \in [\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{F}}_{\rho}$. We distinguish three cases.
  1. $\partial = d_i$ where $(d_1, d_2) \in [\![\mathbf{e}'\,[\mathbf{v}/x]]\!]^{\mathrm{F}}_{\rho}$. By induction hypothesis, there exists $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{F}}_{\rho})$ such that $(d_1, d_2) \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho, x \mapsto S}$. Hence, $d_i \in [\![\pi_i\, \mathbf{e}']\!]^{\mathrm{F}}_{\rho, x \mapsto S}$.
  2. $\partial = \Omega$ where $\Omega \in [\![\mathbf{e}'\,[\mathbf{v}/x]]\!]^{\mathrm{F}}_{\rho}$. By induction hypothesis, there exists $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{F}}_{\rho})$ such that $\Omega \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho, x \mapsto S}$, and the result follows.
  3. $\partial = \Omega$ where there exists $d \in [\![\mathbf{e}'\,[\mathbf{v}/x]]\!]^{\mathrm{F}}_{\rho}$ such that $d \notin \mathscr{D}^{\mathrm{F}} \times \mathscr{D}^{\mathrm{F}}$. By induction hypothesis, there exists $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{F}}_{\rho})$ such that $d \in [\![\mathbf{e}']\!]^{\mathrm{F}}_{\rho, x \mapsto S}$, hence the result.

  The other inclusion is proven using the same reasoning.

□

A.2.2. A denotational semantics for CDuce

Lemma A.37. *For every type $t \in \mathsf{Types}$ and every mark $\mathfrak{m} \in \mathscr{M}$, if $d \in [\![t]\!]^{C}$ then $[d]^{\mathfrak{m}} \in [\![t]\!]^{C}$.*

Proof. Immediate by induction on (d, t) and Definition 11.3. □

Theorem A.38 (Type soundness for λ_{C})**.** *For every type environment $\Gamma \in \mathsf{TEnvs}$ and every term $\mathbf{e} \in \mathsf{Terms}^{C}$, if $\Gamma \vdash \mathbf{e} : t$ then for every $\rho \in [\![\Gamma]\!]^{C}$, $[\![\mathbf{e}]\!]^{C}_{\rho} \subseteq [\![t]\!]^{C}$.*

Proof. The proof is done by structural induction on $\mathbf{e} \in \mathsf{Terms}^{\mathrm{C}}$, supposing $\Gamma \vdash \mathbf{e} : t$. The induction hypothesis is generalized over Γ.

Note that if $\Gamma(x) = \mathbb{0}$ for some $x \in \mathsf{dom}(\Gamma)$, then $[\![\Gamma]\!]^{\mathrm{C}} = \emptyset$, and the result holds vacuously. Therefore, we consider that $\Gamma(x) \neq \mathbb{0}$ for every $x \in \mathsf{dom}(\Gamma)$, which ensures that the derivation of $\Gamma \vdash \mathbf{e} : t$ comes from a structural rule and an application of $[\mathrm{T}^{\mathrm{C}}_{\mathsf{Sub}}]$, and not $[\mathrm{T}^{\mathrm{C}}_{\mathsf{Efq}}]$.

- $\underline{\mathbf{e} = c.}$ By inversion of the typing rules, $b_c \leq t$, therefore $c^{\varepsilon} \in [\![t]\!]^{\mathrm{C}}$.
- $\underline{\mathbf{e} = x.}$ By inversion of the typing rules, $\Gamma(x) = t'$ where $t' \leq t$. By definition of $[\![\Gamma]\!]^{\mathrm{C}}$, we have $\rho(x) \subseteq [\![t']\!]^{\mathrm{C}} \subseteq [\![t]\!]^{\mathrm{C}}$.
- $\underline{\mathbf{e} = \lambda^{\bigwedge_{i \in I}(s_i \to t_i) \wedge \bigwedge_{n \in N} \neg(s_n \to t_n)} x.\, \mathbf{e}'.}$ By inversion of the typing rules, we have $\bigwedge_{i \in I}(s_i \to t_i) \wedge \bigwedge_{n \in N} \neg(s_n \to t_n) \leq t$, and for every $i \in I$, $\Gamma, x : s_i \vdash \mathbf{e}' : t_i$. Note that, by Definition 11.5, it immediately holds that $[\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho} \subseteq [\![\bigwedge_{n \in N} \neg(s_n \to t_n)]\!]^{\mathrm{C}}$. We just have to show that $[\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho} \subseteq [\![\bigwedge_{i \in I}(s_i \to t_i)]\!]^{\mathrm{C}}$, and the result will follow by definition of subtyping.
  Let $R^{\varepsilon} \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho}$, $i \in I$, and let $(\iota, \partial) \in R$. Suppose that $\iota \cap [\![s_i]\!]^{\mathrm{C}} \neq \emptyset$. By Definition 11.5, necessarily $S \subseteq [\![s_i]\!]^{\mathrm{C}}$, and $\partial \in [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho, x \mapsto \iota}$. Since $(\rho, x \mapsto \iota) \in [\![\Gamma, x : s_i]\!]^{\mathrm{C}}$, we can apply the induction hypothesis to deduce that $[\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho, x \mapsto \iota} \subseteq [\![t_i]\!]^{\mathrm{C}}$. Therefore, $\partial \in [\![t_i]\!]^{\mathrm{C}}$.
  Now, if $\iota = \mathfrak{O}_d$ where $d \in [\![s_i]\!]^{\mathrm{C}}$, by Definition 11.5, we have $\partial \in [\![t_i]\!]^{\mathrm{C}}$. Hence, $R^{\varepsilon} \in [\![s_i \to t_i]\!]^{\mathrm{C}}$ for every $i \in I$, and the result follows.
- $\underline{\mathbf{e} = \mathbf{e}_1\, \mathbf{e}_2.}$ By inversion of the typing rules, $\Gamma \vdash \mathbf{e}_1 : t_1 \to t_2$, $\Gamma \vdash \mathbf{e}_2 : t_1$, and $t_2 \leq t$.

By induction hypothesis, $[\![\mathbf{e}_1]\!]^{\mathrm{C}}_\rho \subseteq [\![t_1 \to t_2]\!]^{\mathrm{C}}$ and $[\![\mathbf{e}_2]\!]^{\mathrm{C}}_\rho \subseteq [\![t_1]\!]^{\mathrm{C}}$. Therefore, $\Omega \notin [\![\mathbf{e}_1]\!]^{\mathrm{C}}_\rho \cup [\![\mathbf{e}_2]\!]^{\mathrm{C}}_\rho$. Moreover, this ensures that for every $d \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_\rho$, $d = R^{\mathfrak{m}}$ where $R \in \mathscr{P}_f(\mathscr{I}^{\mathrm{C}} \times \mathscr{D}^{\mathrm{C}}_\Omega)$. Thus, we deduce that $\Omega^\rho_{\mathbf{e}_1\,\mathbf{e}_2} = \emptyset$.
Now let $\partial \in [\![\mathbf{e}_1\,\mathbf{e}_2]\!]^{\mathrm{C}}_\rho$. By inversion, there exists $\mathfrak{m}_1, \mathfrak{m}_2 \in \mathscr{M}$, $R^{\mathfrak{m}_1} \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_\rho$ and $S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{C}}_\rho\big|_{\mathfrak{m}_2}$ such that $(S, \partial') \in R$, and $\partial = [\partial']^{\mathfrak{m}_1 \cdot \mathfrak{m}_2}$. The induction hypothesis yields that $R^{\mathfrak{m}_1} \in [\![t_1 \to t_2]\!]^{\mathrm{C}}$ and $S \subseteq [\![t_1]\!]^{\mathrm{C}}$. Thus, by Definition 11.3, this yields that $\partial' \in [\![t_2]\!]^{\mathrm{C}}$, and it immediately follows that $\partial \in [\![t_2]\!]^{\mathrm{C}}$ by Lemma A.37.

- $\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2)$. By inversion of the typing rules, $\Gamma \vdash \mathbf{e}_1 : t_1$ and $\Gamma \vdash \mathbf{e}_2 : t_2$, and $t_1 \times t_2 \leq t$. Moreover, by induction hypothesis, $[\![\mathbf{e}_1]\!]^{\mathrm{C}}_\rho \subseteq [\![t_1]\!]^{\mathrm{C}}$ and $[\![\mathbf{e}_2]\!]^{\mathrm{C}}_\rho \subseteq [\![t_2]\!]^{\mathrm{C}}$. We deduce from this fact that $\Omega \notin [\![\mathbf{e}_1]\!]^{\mathrm{C}}_\rho \cup [\![\mathbf{e}_2]\!]^{\mathrm{C}}_\rho$, thus $\Omega^\rho_{(\mathbf{e}_1,\mathbf{e}_2)} = \emptyset$. This also yields immediately that $[\![(\mathbf{e}_1, \mathbf{e}_2)]\!]^{\mathrm{C}}_\rho \subseteq [\![t_1 \times t_2]\!]^{\mathrm{C}}$, hence the result.

- $\mathbf{e} = \pi_i\,\mathbf{e}'$. By inversion of the typing rules, $\Gamma \vdash \mathbf{e}' : t_1 \times t_2$ and $t_i \leq t$. Moreover, by induction hypothesis, $[\![\mathbf{e}']\!]^{\mathrm{C}}_\rho \subseteq [\![t_1 \times t_2]\!]^{\mathrm{C}}$. Thus, this ensures that for every $d \in [\![\mathbf{e}']\!]^{\mathrm{C}}_\rho$, $d = (d_1, d_2)^{\mathfrak{m}}$, thus $\Omega^\rho_{\pi_i\,\mathbf{e}'} = \emptyset$.
Now let $d \in [\![\pi_i\,\mathbf{e}']\!]^{\mathrm{C}}_\rho$. By inversion, there exists $(d_1, d_2)^{\mathfrak{m}} \in [\![\mathbf{e}']\!]^{\mathrm{C}}_\rho$ such that $d = [d_i]^{\mathfrak{m}}$. The induction hypothesis yields that $d_i \in [\![t_i]\!]^{\mathrm{C}}$, which yields $d \in [\![t]\!]^{\mathrm{C}}$ by Lemma A.37 and definition of subtyping.

- $\mathbf{e} = (x = \mathbf{e}' \in t_c)?\,\mathbf{e}_1 : \mathbf{e}_2$. By inversion of the typing rules, $\Gamma \vdash \mathbf{e}' : t'$, thus, by induction hypothesis, $[\![\mathbf{e}']\!]^{\mathrm{C}}_\rho \subseteq [\![t']\!]^{\mathrm{C}}$. This ensures that $\Omega \notin [\![\mathbf{e}']\!]^{\mathrm{C}}_\rho$, and thus $\Omega^\rho_{\mathbf{e}} = \emptyset$.
Now let $\partial \in [\![\mathbf{e}]\!]^{\mathrm{C}}_\rho$. We distinguish two cases.
  - $\exists \mathfrak{m} \in \mathscr{M}$ such that $S = [\![\mathbf{e}']\!]^{\mathrm{C}}_\rho\big|_{\mathfrak{m}} \subseteq [\![t_c]\!]^{\mathrm{C}}$ and $\exists S' \in \mathscr{P}_f(S), \partial' \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho, x \mapsto S'}$ such that $\partial = [\partial']^{\mathfrak{m}}$. Since $S' \subseteq [\![t_c]\!]^{\mathrm{C}} \cap [\![t']\!]^{\mathrm{C}}$, we have $(\rho, x \mapsto S') \in [\![\Gamma, x : t_c \wedge t']\!]^{\mathrm{C}}$. And by inversion of the typing rules, $\Gamma, x : t_c \wedge t' \vdash \mathbf{e}_1 : s$ where $s \leq t$. Thus, by IH, we have $\partial' \in [\![s]\!]^{\mathrm{C}} \subseteq [\![t]\!]^{\mathrm{C}}$. By Lemma A.37, we have $\partial \in [\![t]\!]^{\mathrm{C}}$.
  - $\exists \mathfrak{m} \in \mathscr{M}$ such that $[\![\mathbf{e}']\!]^{\mathrm{C}}_\rho\big|_{\mathfrak{m}} \not\subseteq [\![t_c]\!]^{\mathrm{C}}$ and $\exists S' \in \mathscr{P}_f(S), \partial' \in [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho, x \mapsto S'}$ where $S = [\![\mathbf{e}']\!]^{\mathrm{C}}_\rho\big|_{\mathfrak{m}} \cap [\![\neg t_c]\!]^{\mathrm{C}}$.
  Since $S' \subseteq [\![\neg t_c]\!]^{\mathrm{C}} \cap [\![t']\!]^{\mathrm{C}}$, we have $(\rho, x \mapsto S) \in [\![\Gamma, x : \neg t_c \wedge t']\!]^{\mathrm{C}}$. And by inversion of the typing rules, $\Gamma, x : \neg t_c \wedge t' \vdash \mathbf{e}_1 : s$ where $s \leq t$. Thus, by IH, we have $\partial' \in [\![s]\!]^{\mathrm{C}} \subseteq [\![t]\!]^{\mathrm{C}}$. By Lemma A.37, we have $\partial \in [\![t]\!]^{\mathrm{C}}$.

- $\mathbf{e} = \mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2)$. By inversion of the typing rules, $\Gamma \vdash \mathbf{e}_1 : s$ and $\Gamma \vdash \mathbf{e}_2 : s$ where $s \leq t$. By IH, this ensures $[\![\mathbf{e}_1]\!]^{\mathrm{C}}_\rho \subseteq [\![s]\!]^{\mathrm{C}}$ and similarly for $\mathbf{e}_2$. Thus, we have $\Omega \notin [\![\mathbf{e}_1]\!]^{\mathrm{C}}_\rho \cup [\![\mathbf{e}_2]\!]^{\mathrm{C}}_\rho$, which ensures that $\Omega \notin [\![\mathbf{e}]\!]^{\mathrm{C}}_\rho$.
Now let $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_\rho$. Either $\exists d' \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_\rho$ such that $d = [d']^l$, or $\exists d' \in [\![\mathbf{e}_2]\!]^{\mathrm{C}}_\rho$ and $d = [d']^r$. In both cases, $d' \in [\![s]\!]^{\mathrm{C}}$ by IH, and $d \in [\![s]\!]^{\mathrm{C}}$ follows by Lemma A.37. Thus $d \in [\![t]\!]^{\mathrm{C}}$ by subtyping.

□

Lemma A.39. *For every value* $\mathbf{v} \in \mathsf{Values}^C$ *and every* $d \in [\![\mathbf{v}]\!]^C_\rho$, $\mathsf{mark}(d) = \varepsilon$.

Proof. Immediate by Definition 11.5 and cases on $\mathbf{v}$. □

Lemma A.40. *For every term* $\mathbf{e} \in \mathsf{Terms}^C$, $x \in \mathsf{Vars}$, $\rho \in \mathsf{Envs}$, *and* $S_1, S_2 \in \mathscr{P}(\mathscr{D}^C)$ *such that* $\Omega \notin [\![\mathbf{e}]\!]^C_{\rho,x\mapsto S_1}$, *if* $S_1 \subseteq S_2$ *then* $[\![\mathbf{e}]\!]^C_{\rho,x\mapsto S_1} \subseteq [\![\mathbf{e}]\!]^C_{\rho,x\mapsto S_2}$

Proof. The proof is done by structural induction on $\mathbf{e} \in \mathsf{Terms}^{\mathrm{C}}$, generalized over ρ.

- $\underline{\mathbf{e} = c.}$ Immediate since x does not appear in $\mathbf{e}$.
- $\underline{\mathbf{e} = y.}$ If $y \neq x$ then the result is immediate. Otherwise, $[\![x]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1} = S_1 \subseteq S_2 = [\![x]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$ which gives the result.
- $\underline{\mathbf{e} = \lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)} y.\, \mathbf{e}'.}$ Let $R^\varepsilon \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}$. Let $(\iota, \partial) \in R$. There are three cases.
  1. There exists $i \in I$ such that $\iota \subseteq [\![s_i]\!]^{\mathrm{C}}$ then $\partial \in [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto S_1, y\mapsto \iota}$. By generalized induction hypothesis, $\partial \in [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto S_2, y\mapsto \iota}$.
  2. $\forall i \in I,\ \iota \subseteq [\![\neg s_i]\!]^{\mathrm{C}}$ then $\partial = \Omega$ independently of S_1 and S_2.
  3. $\iota = \mathfrak{V}_d$ for some $d \in \mathscr{D}^{\mathrm{C}}$. Then ∂ satisfies $\forall i \in I,\ d \in [\![s_i]\!]^{\mathrm{C}} \implies \partial \in [\![t_i]\!]^{\mathrm{C}}$ which is independent of S_1 and S_2.

  Therefore, $R^\varepsilon \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$.
- $\underline{\mathbf{e} = \mathbf{e}_1\, \mathbf{e}_2.}$ Let $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}$. By Definition 11.5, there exists $\mathfrak{m}_1, \mathfrak{m}_2 \in \mathscr{M}$, $R^{\mathfrak{m}_1} \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}$ and $S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}\big|_{\mathfrak{m}_2}$ such that $(S, d') \in R$ where $d = [d']^{\mathfrak{m}_1.\mathfrak{m}_2}$. By induction hypothesis, $R^{\mathfrak{m}_1} \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$ and $S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}\big|_{\mathfrak{m}_2}$ thus $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$ by Definition 11.5.
- $\underline{\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2).}$ Let $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}$. By Definition 11.5, there exists $d_1 \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}$ and $d_2 \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}$ such that $d = (d_1, d_2)^\varepsilon$. By induction hypothesis, $d_1 \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$ and $d_2 \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$, thus $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$.
- $\underline{\mathbf{e} = \pi_i\, \mathbf{e}'.}$ Let $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}$. By Definition 11.5, there exists $(d_1, d_2)^{\mathfrak{m}} \in [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}$ such that $d = [d_i]^{\mathfrak{m}}$. By induction hypothesis, $(d_1, d_2)^{\mathfrak{m}} \in [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$ thus $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$.
- $\underline{\mathbf{e} = \mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2).}$ Let $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}$. By Definition 11.5, there are two cases.
  1. There exists $d' \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}$ such that $d = [d']^l$. By IH, $d' \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$. Therefore, $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$ by Definition 11.5.
  2. There exists $d' \in [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}$ such that $d = [d']^r$. By IH, $d' \in [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$. Therefore, $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$ by Definition 11.5.
- $\underline{\mathbf{e} = (y = \mathbf{e}' \in t)?\, \mathbf{e}_1 : \mathbf{e}_2.}$ Let $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}$. We distinguish two cases.
  1. There exists $\mathfrak{m} \in \mathscr{M}$ such that $S \subseteq [\![t]\!]^{\mathrm{C}}$, $S' \in \mathscr{P}_f(S)$ and $d' \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1, y\mapsto S'}$ where $d = [d']^{\mathfrak{m}}$ and $S = [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}\big|_{\mathfrak{m}}$. By generalized IH, $S' \subseteq [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}\big|_{\mathfrak{m}}$. Thus, by applying the generalized IH on $S_1 \subseteq S_2$, we have $d' \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2, y\mapsto S'}$. We deduce that $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$.

2. There exists $\mathfrak{m} \in \mathscr{M}$ such that $S \neq \emptyset$, $S' \in \mathscr{P}_f(S)$ and $d' \in [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1,y\mapsto S'}$ where $d = [d']^{\mathfrak{m}}$ and $S = [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}\big|_{\mathfrak{m}} \cap [\![\neg t]\!]^{\mathrm{C}}$. By generalized IH, $S' \subseteq [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}\big|_{\mathfrak{m}} \cap [\![\neg t]\!]^{\mathrm{C}}$. Thus, we have $S \neq \emptyset$ and by applying the generalized IH on $S_1 \subseteq S_2$, we have $d' \in [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2,y\mapsto S'}$. We deduce that $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}$.

□

Lemma A.41. *For every term* $\mathbf{e} \in \mathsf{Terms}^C$, $\mathbf{v} \in \mathsf{Values}^C$, $x \in \mathsf{Vars}$, $\rho \in \mathsf{Envs}$ *such that* $\Omega \notin [\![\mathbf{e}\,[\mathbf{v}/x]]\!]^C_\rho$,

$$[\![\mathbf{e}\,[\mathbf{v}/x]]\!]^C_\rho = \bigcup_{S\in\mathscr{P}_f([\![\mathbf{v}]\!]^C_\rho)} [\![\mathbf{e}]\!]^C_{\rho,x\mapsto S}$$

Proof. The proof is done by structural induction on $\mathbf{e} \in \mathsf{Terms}^{\mathrm{C}}$.

- $\underline{\mathbf{e} = c.}$ Immediate since x does not appear in c.
- $\underline{\mathbf{e} = y.}$ If $y \neq x$ the result is immediate. Otherwise, if $y = x$ we have $[\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_\rho = [\![\mathbf{v}]\!]^{\mathrm{C}}_\rho$ and we reason by double inclusion.
  - Consider $d \in [\![\mathbf{v}]\!]^{\mathrm{C}}_\rho$. Then we immediately have the result taking $S = \{d\}$: $[\![x]\!]^{\mathrm{C}}_{\rho,x\mapsto\{d\}} = \{d\}$.
  - Let $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{C}}_\rho)$ and consider $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S}$. Since $[\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S} = S$, we have $d \in S$. And since $S \subseteq [\![\mathbf{v}]\!]^{\mathrm{C}}_\rho$, $d \in [\![\mathbf{v}]\!]^{\mathrm{C}}_\rho$.
- $\underline{\mathbf{e} = \lambda^{\wedge_{i\in I}(s_i\to t_i)\wedge\wedge_{n\in N}\neg(s_n\to t_n)} y.\,\mathbf{e}'.}$ By definition, $[\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_\rho = [\![\lambda^{\wedge_{i\in I}(s_i\to t_i)\wedge\wedge_{n\in N}\neg(s_n\to t_n)} y.\,(\mathbf{e}'\,[\mathbf{v}/x])]\!]^{\mathrm{C}}_\rho$. We proceed by double inclusion.
  - Let $R^\varepsilon \in [\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_\rho$, and let us write $R = \{(\iota_j, \partial_j) \mid j \in J\}$. Let $j \in J$. If $\iota_j \subseteq [\![s_i]\!]^{\mathrm{C}}$ for some $i \in I$, then by definition this means that $\partial_j \in [\![\mathbf{e}'\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_{\rho,y\mapsto\iota_j}$. By induction hypothesis, there exists $S^{\mathrm{v}}_j \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{C}}_\rho)$ such that $\partial_j \in [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,y\mapsto\iota_j,x\mapsto S^{\mathrm{v}}_j}$. Moreover, if $\iota_j \not\subseteq [\![s_i]\!]^{\mathrm{C}}$ for every $i \in I$, then $\partial = \Omega$ by definition. So in this case, we just pose $S^{\mathrm{v}}_j = \emptyset$.
    Now consider $S^{\mathrm{v}} = \bigcup_{j\in J} S^{\mathrm{v}}_j$. It is immediate that $S^{\mathrm{v}} \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{C}}_\rho)$. Moreover, by Lemma A.40, we deduce that $\forall j \in J, \forall i \in I$, if $\iota_j \subseteq [\![s_i]\!]^{\mathrm{C}}$, then $\partial_j \in [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,y\mapsto\iota_j,x\mapsto S^{\mathrm{v}}}$. Thus, this proves that $R^\varepsilon \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S^{\mathrm{v}}}$.
  - Let $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{C}}_\rho)$ and $R^\varepsilon \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S}$. Let $(\iota_j, \partial_j) \in R$. Note that if $\forall i \in I.\ \iota_j \not\subseteq [\![s_i]\!]^{\mathrm{C}}$ then $\partial_j = \Omega$ by definition, so this part is immediate. Now suppose that $\iota_j \subseteq [\![s_i]\!]^{\mathrm{C}}$ for some $i \in I$. By definition, $\partial_j \in [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto S,y\mapsto\iota_j}$. By induction hypothesis, this yields $\partial_j \in [\![\mathbf{e}'\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_{\rho,y\mapsto\iota_j}$ and thus $R^\varepsilon \in [\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_\rho$.
- $\underline{\mathbf{e} = \mathbf{e}_1\,\mathbf{e}_2.}$ We prove the result by double inclusion.
  - Let $d \in [\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_\rho$. By definition, there exists $\mathfrak{m}_1, \mathfrak{m}_2 \in \mathscr{M}$, $R^{\mathfrak{m}_1} \in [\![\mathbf{e}_1\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_\rho$ and $S \in \mathscr{P}_f([\![\mathbf{e}_2\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_\rho\big|_{\mathfrak{m}_2})$ such that $(S, d') \in R$ where $d = [d']^{\mathfrak{m}_1.\mathfrak{m}_2}$. By induction hypothesis, we deduce that there exists $S_1, S_2 \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{C}}_\rho)$ such that $R^{\mathfrak{m}_1} \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}$ and $S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho,x\mapsto S_2}\big|_{\mathfrak{m}_2}$. Taking $S = S_1 \cup S_2$ and ap-

plying Lemma A.40 yields that $R^{\mathfrak{m}_1} \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho,x\mapsto S}$ and $S \subseteq [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho,x\mapsto S}\big|_{\mathfrak{m}_2}$, thus $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S}$.

- The same reasoning proves the other inclusion.

- $\underline{\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2).}$ We prove the result by double inclusion.
  - Let $d \in [\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_{\rho}$. By definition, $d = (d_1, d_2)^{\epsilon}$ where $\forall i \in \{1,2\}$, $d_i \in [\![\mathbf{e}_i\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_{\rho}$. By induction hypothesis, $\forall i \in \{1,2\}$, there exists $S_i \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{C}}_{\rho})$ such that $d_i \in [\![\mathbf{e}_i]\!]^{\mathrm{C}}_{\rho,x\mapsto S_i}$. Taking $S = S_1 \cup S_2$ and applying Lemma A.40 yields that $\forall i \in \{1,2\}$, $d_i \in [\![\mathbf{e}_i]\!]^{\mathrm{C}}_{\rho,x\mapsto S}$. Thus, $d = (d_1, d_2)^{\epsilon} \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S}$.
  - The second inclusion is proven using the same reasoning.

- $\underline{\mathbf{e} = \pi_i\,\mathbf{e}'.}$ We prove the result by double inclusion.
  - Let $d \in [\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_{\rho}$. By definition, $d = [d_i]^{\mathfrak{m}}$ for some $(d_1, d_2)^{\mathfrak{m}} \in [\![\mathbf{e}'\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_{\rho}$. By induction hypothesis, there exists $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{C}}_{\rho})$ such that $(d_1, d_2)^{\mathfrak{m}} \in [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto S}$. Hence, $[d_i]^{\mathfrak{m}} \in [\![\pi_i\,\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto S}$.
  - The other inclusion is proven using the same reasoning.

- $\underline{\mathbf{e} = \mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2).}$ We prove the first inclusion, the second is proven using the same reasoning. Let $d \in [\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_{\rho}$ By Definition 11.5, there are two cases.
  1. There exists $d' \in [\![\mathbf{e}_1\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_{\rho}$ such that $d = [d']^{l}$. By IH, there exists $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{C}}_{\rho})$ such that $d' \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho,x\mapsto S}$. Therefore, $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S}$ by Definition 11.5.
  2. There exists $d' \in [\![\mathbf{e}_2\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_{\rho}$ such that $d = [d']^{r}$. By IH, there exists $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{C}}_{\rho})$ such that $d' \in [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho,x\mapsto S}$. Therefore, $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S}$ by Definition 11.5.

- $\underline{\mathbf{e} = (y = \mathbf{e}' \in t)?\,\mathbf{e}_1 : \mathbf{e}_2.}$ We prove the first inclusion, the second is proven using the same reasoning. Let $d \in [\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_{\rho}$. We distinguish two cases.
  1. There exists $\mathfrak{m} \in \mathscr{M}$ such that $S \subseteq [\![t]\!]^{\mathrm{C}}$, $S' \in \mathscr{P}_f(S)$ and $d' \in [\![\mathbf{e}_1\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_{\rho,y\mapsto S'}$ where $d = [d']^{\mathfrak{m}}$ and $S = [\![\mathbf{e}'\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_{\rho}\big|_{\mathfrak{m}}$. By IH, there exists $S_1 \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{C}}_{\rho})$ such that $S' \subseteq [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}\big|_{\mathfrak{m}}$. By IH, there exists $S_2 \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{C}}_{\rho})$ such that $d' \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho,y\mapsto S',x\mapsto S_2}$. By considering $\hat{S} = S_1 \cup S_2$ and applying Lemma A.40 we first deduce that $S' \subseteq [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto \hat{S}}\big|_{\mathfrak{m}}$ and then that $d' \in [\![\mathbf{e}_1]\!]^{\mathrm{C}}_{\rho,y\mapsto S'',x\mapsto \hat{S}}$ where $S'' = [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto \hat{S}}\big|_{\mathfrak{m}}$. Thus, $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto \hat{S}}$.
  2. There exists $\mathfrak{m} \in \mathscr{M}$ such that $S \neq \emptyset$, $S' \in \mathscr{P}_f(S)$ and $d' \in [\![\mathbf{e}_2]\!]^{\mathrm{C}}_{\rho,x\mapsto S_1,y\mapsto S'}$ where $d = [d']^{\mathfrak{m}}$ and $S = [\![\mathbf{e}']\!]^{\mathrm{C}}_{\rho,x\mapsto S_1}\big|_{\mathfrak{m}} \cap [\![\neg t]\!]^{\mathrm{C}}$. This case is proven exactly as the first one.

□

Lemma A.42. *For every relations $R, R' \in \mathscr{P}_f(\mathscr{I}^C \times \mathscr{D}^C_\Omega)$, every types $s, t \in$ Types, and every mark $\mathfrak{m} \in \mathscr{M}$, if $R \subseteq R'$ and $R'^{\mathfrak{m}} \in [\![s \to t]\!]^C$ then $R^{\mathfrak{m}} \in [\![s \to t]\!]^C$.*

Proof. Immediate by Definition 11.3. □

Lemma A.43. *For every λ-abstraction $\mathbf{v} = \lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)}x.\,\mathbf{e} \in \mathsf{Values}^C$ and every relation $R \in \mathscr{P}_f(\mathscr{I}^C \times \mathscr{D}^C_\Omega)$, the following holds:*

$$R^\varepsilon \in [\![\mathbf{v}]\!]^C_\rho \iff \begin{cases} \mho(R)^\varepsilon \in [\![\bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{n\in N}\neg(s_n \to t_n)]\!]^C \\ (R \setminus \mho(R))^\varepsilon \in [\![\lambda^{\bigwedge_{i\in I}(s_i\to t_i)}x.\,\mathbf{e}]\!]^C_\rho \end{cases}$$

where $\mho(R) = \{(\mho_d, \partial) \mid (\mho_d, \partial) \in R\}$

Proof. Suppose that $R^\varepsilon \in [\![\mathbf{v}]\!]^C_\rho$. By Theorem 11.6, we have $R^\varepsilon \in [\![\bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{n\in N}\neg(s_n \to t_n)]\!]^C$ which ensures by Lemma A.42 that $\mho(R)^\varepsilon \in [\![\bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{n\in N}\neg(s_n \to t_n)]\!]^C$. The second part is immediate by Definition 11.5.
For the converse, it suffices to remark that all the pairs of R verify the conditions of Definition 11.5 by hypothesis. We just have to show that $R^\varepsilon \in [\![\bigwedge_{n\in N}\neg(s_n \to t_n)]\!]^C$. By Theorem 11.6, we have that $(R \setminus \mho(R))^\varepsilon \in [\![\bigwedge_{i\in I}(s_i \to t_i)]\!]^C$ and by hypothesis $\mho(R)^\varepsilon \in [\![\bigwedge_{i\in I}(s_i \to t_i)]\!]^C$. By Definition 11.3, this ensures that $R^\varepsilon \in [\![\bigwedge_{i\in I}(s_i \to t_i)]\!]^C$. Since $\mho(R)^\varepsilon \in [\![\bigwedge_{n\in N}\neg(s_n \to t_n)]\!]^C$ and $\bigwedge_{n\in N}\neg(s_n \to t_n) \simeq \neg(\bigvee_{n\in N} s_n \to t_n)$, by contrapositive of Lemma A.42, we deduce that $\mho(R)^\varepsilon \notin [\![\bigvee_{n\in N} s_n \to t_n]\!]^C$ implies that $R^\varepsilon \notin [\![\bigvee_{n\in N} s_n \to t_n]\!]^C$, which ensures that $R^\varepsilon \in [\![\bigwedge_{n\in N}\neg(s_n \to t_n)]\!]^C$. Hence, $R^\varepsilon \in [\![\mathbf{v}]\!]^C_\rho$. □

Lemma A.44. *For every type $t \in \mathsf{Types}$ and every relation $R \in \mathscr{P}_f(\mathscr{I}^C \times \mathscr{D}^C_\Omega)$, if $R^{\mathfrak{m}} \in [\![t]\!]^C$ then $\hat{R}^{\mathfrak{m}} \in [\![t]\!]^C$ where $\hat{R} = \{(\mho_d, \partial) \mid (\iota, \partial) \in R \setminus \mho(R),\ d \in \iota\} \cup \mho(R)$.*

Proof. By induction on t, which we never apply the induction hypothesis under type constructors.

- $\underline{t = t_1 \to t_2}$. Let $(\mho_d, \partial) \in \hat{R}$. There are two cases. Either $(\mho_d, \partial) \in \mho(R)$, in which case the pair satisfies the condition of Definition 11.3 by hypothesis. Otherwise, there exists $\iota \in \mathscr{I}^C$ such that $d \in \iota$ and $(\iota, \partial) \in R \setminus \mho(R)$. Now suppose that $d \in [\![t_1]\!]^C$. This ensures that $\iota \cap [\![t_1]\!]^C \neq \emptyset$. By Definition 11.3, this proves that $\partial \in [\![t_2]\!]^C$, hence the result.

- $\underline{t = t_1 \vee t_2}$. By hypothesis, there exists $i \in I$ such that $R^{\mathfrak{m}} \in [\![t_i]\!]^C$. By IH, we have $\hat{R}^{\mathfrak{m}} \in [\![t_i]\!]^C$, which proves that $\hat{R}^{\mathfrak{m}} \in [\![t]\!]^C$.

- $\underline{t = \neg t'}$. We distinguish the following cases.
  - $t' = \neg(t_1 \to t_2)$ By hypothesis, we distinguish two cases. Either there exists $(\mho_d, \partial) \in R$ such that $d \in [\![t_1]\!]^C$ and $\partial \notin [\![t_2]\!]^C$. In that case, $(\mho_d, \partial) \in \hat{R}$ by definition, which ensures that $\hat{R}^{\mathfrak{m}} \in [\![\neg(t_1 \to t_2)]\!]^C$. Otherwise, there exists $(\iota, \partial) \in R$ such that $\iota \cap [\![t_1]\!]^C \neq \emptyset$ and $\partial \notin [\![t_2]\!]^C$. Thus, there exists $d \in \iota \cap [\![t_1]\!]^C$. By definition, $(\mho_d, \partial) \in \hat{R}$, which ensures that $\hat{R}^{\mathfrak{m}} \in [\![\neg(t_1 \to t_2)]\!]^C$.
  - $t' = \neg(t_1 \vee t_2)$ By hypothesis, $R^{\mathfrak{m}} \in [\![\neg t_1]\!]^C$ and $R^{\mathfrak{m}} \in [\![\neg t_2]\!]^C$. By IH, this yields that for every $i \in \{1, 2\}$, $\hat{R}^{\mathfrak{m}} \in [\![\neg t_i]\!]^C$, hence the result.
  - $t' = \neg t''$ Immediate by induction on t'' since $\neg\neg t'' \simeq t'' \simeq t$.
  - All other cases are immediate since $\mathbb{0} \to \mathbb{1} \leq t$ and thus any relation belongs to t.

- All other cases are vacuously true since $R^{\mathfrak{m}} \in [\![t]\!]^{\mathrm{C}}$ cannot hold.

□

Lemma A.45. *For every* $\mathbf{v} = \lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)}x.\,\mathbf{e} \in \mathsf{Values}^C$, *if* $R^{\varepsilon} \in [\![\lambda^{\bigwedge_{i\in I}(s_i\to t_i)}x.\,\mathbf{e}]\!]^C_\rho$ *then there exists* $R'^{\varepsilon} \in [\![\mathbf{v}]\!]^C_\rho$ *such that* $R \setminus \mho(R) \subseteq R'$.

Proof. We write $t = \bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{n\in N}\neg(s_n \to t_n)$. First, since $t \not\leq \mathbb{0}$ by definition of values and $t \leq \mathbb{0} \to \mathbb{1}$, there exists $R^{\varepsilon}_N \in [\![t]\!]^{\mathrm{C}}$. By Lemma A.44, we deduce that $\hat{R}^{\varepsilon}_N \in [\![t]\!]^{\mathrm{C}}$. Now by Lemma A.43, we deduce that $(R \setminus \mho(R))^{\varepsilon} \in [\![\lambda^{\bigwedge_{i\in I}(s_i\to t_i)}x.\,\mathbf{e}]\!]^{\mathrm{C}}_\rho$. If we write $R' = (R \setminus \mho(R)) \cup \hat{R}_N$, by Lemma A.43 again, since $\mho(R') = \hat{R}_N$, we deduce that $R'^{\varepsilon} \in [\![\lambda^{t}x.\,\mathbf{e}]\!]^{\mathrm{C}}_\rho$, hence the result. □

Theorem A.46 (Computational soundness for λ_{C})**.** *For every term* $\mathbf{e} \in \mathrm{Prg}_\Gamma(t)$ *and every environment* $\rho \in [\![\Gamma]\!]^C$, *if* $\mathbf{e} \rightsquigarrow \mathbf{e}'$ *then there exists* $\mathfrak{m} \in \mathscr{M}$ *such that* $[\![\mathbf{e}']\!]^{\mathrm{C}}_\rho = \{d \in \mathscr{D}^C \mid [d]^{\mathfrak{m}} \in [\![\mathbf{e}]\!]^C_\rho\}$.

Proof. The proof is done by structural induction on $\mathbf{e} \in \mathrm{Prg}_\Gamma(t)$ and cases over the reduction rule used for $\mathbf{e} \rightsquigarrow \mathbf{e}'$. Since $\Gamma \vdash \mathbf{e} : t$, by Theorem 11.6, $\Omega \notin [\![\mathbf{e}]\!]^{\mathrm{C}}_\rho$. Thus, all cases involving Ω can be ignored. Note that $\mathfrak{m} = \varepsilon$ in all cases except $[\mathrm{R}^{\mathrm{C}}_{\mathsf{Choice}_i}]$ and $[\mathrm{R}^{\mathrm{C}}_{\mathsf{Ctx}}]$.

- $[\mathrm{R}^{\mathrm{C}}_{\mathsf{App}}]$. $(\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)}x.\,\mathbf{e})\,\mathbf{v} \rightsquigarrow \mathbf{e}\,[\mathbf{v}/x]$. We proceed by double inclusion.
  - Let $d \in [\![(\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)}x.\,\mathbf{e})\,\mathbf{v}]\!]^{\mathrm{C}}_\rho$. By Definition 11.5, there exists $R^{\varepsilon} \in [\![\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)}x.\,\mathbf{e}]\!]^{\mathrm{C}}_\rho$ and $S \subseteq [\![\mathbf{v}]\!]^{\mathrm{C}}_\rho\big|_\varepsilon$ such that $(S, d') \in R$ where $d = [d']^{\varepsilon.\varepsilon} = d'$. By Definition 11.5, we have $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S}$ and by Lemma A.41, we deduce that $d \in [\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_\rho$.
  - Let $d \in [\![\mathbf{e}\,[\mathbf{v}/x]]\!]^{\mathrm{C}}_\rho$. By Lemma A.41, there exists $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{C}}_\rho)$ such that $d \in [\![\mathbf{e}]\!]^{\mathrm{C}}_{\rho,x\mapsto S}$. Since $\mathbf{e}$ is well-typed, we deduce that $\Gamma \vdash \mathbf{v} : \bigvee_{i\in I} s_i$. By hypothesis, $(\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)}x.\,\mathbf{e})\,\mathbf{v} \in \mathrm{Prg}_\Gamma(t)$, and by definition, $\mathsf{A}^{\to}(\bigvee_{i\in I} s_i) \subseteq \mathsf{A}^{\to}((\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)}x.\,\mathbf{e})\,\mathbf{v})$, which ensures that $\mathscr{W}_{\mathsf{A}^{\to}(\bigvee_{i\in I} s_i)}(\mathbf{v})$. By Proposition 11.8, this ensures that there exists $i \in I$ such that $\Gamma \vdash \mathbf{v} : s_i$. By Theorem 11.6, it holds that $S \subseteq [\![s_i]\!]^{\mathrm{C}}$. Therefore, the pair (S, d) satisfies the conditions of Definition 11.5, and we deduce that $\{(S, d)\}^{\varepsilon} \in [\![\lambda^{\bigwedge_{i\in I}(s_i\to t_i)}x.\,\mathbf{e}]\!]^{\mathrm{C}}_\rho$. By Lemma A.45, we then deduce that there exists $R'^{\varepsilon} \in [\![\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)}x.\,\mathbf{e}]\!]^{\mathrm{C}}_\rho$ such that $(S, d) \in R'$. Hence, by Definition 11.5, we obtain than $d \in [\![(\lambda^{\bigwedge_{i\in I}(s_i\to t_i)\wedge\bigwedge_{n\in N}\neg(s_n\to t_n)}x.\,\mathbf{e})\,\mathbf{v}]\!]^{\mathrm{C}}_\rho$.
- $[\mathrm{R}^{\mathrm{C}}_{\mathsf{Proj}_i}]$. $\pi_i\,(\mathbf{v}_1, \mathbf{v}_2) \rightsquigarrow \mathbf{v}_i$. By Theorem 11.6, $\Omega \notin [\![\mathbf{v}_1]\!]^{\mathrm{C}}_\rho \cup [\![\mathbf{v}_2]\!]^{\mathrm{C}}_\rho$. By Definition 11.5, $\forall d \in [\![(\mathbf{v}_1, \mathbf{v}_2)]\!]^{\mathrm{C}}_\rho$, $d = (d_1, d_2)^{\varepsilon}$. Therefore, $\Omega^{\rho}_{\pi_i\,(\mathbf{v}_1,\mathbf{v}_2)} = \emptyset$.
  We then deduce that by Definition 11.5, $[\![(\mathbf{v}_1, \mathbf{v}_2)]\!]^{\mathrm{C}}_\rho = \{(d_1, d_2)^{\varepsilon} \mid d_1 \in [\![\mathbf{v}_1]\!]^{\mathrm{C}}_\rho \wedge d_2 \in [\![\mathbf{v}_2]\!]^{\mathrm{C}}_\rho\}$, which immediately gives that $[\![\pi_i\,(\mathbf{v}_1, \mathbf{v}_2)]\!]^{\mathrm{C}}_\rho = [\![\mathbf{v}_i]\!]^{\mathrm{C}}_\rho$ since for every $d \in \mathscr{D}^{\mathrm{C}}$, $[d]^{\varepsilon} = d$.
- $[\mathrm{R}^{\mathrm{C}}_{\mathsf{CaseL}}]$. $(x = \mathbf{v} \in t')?\,\mathbf{e}_1 : \mathbf{e}_2 \rightsquigarrow \mathbf{e}_1\,[\mathbf{v}/x]$ where $\mathbf{v} \in t'$. By Lemma 11.13, we have

$\Gamma \vdash \mathbf{v} : \mathsf{type}(\mathbf{v})$ and by hypothesis, we have $\mathsf{type}(\mathbf{v}) \leq t'$ and $\Gamma, x : \mathsf{type}(\mathbf{v}) \vdash \mathbf{e}_1 : t$. Therefore, by Theorem 11.6, we deduce that $[\![\mathbf{v}]\!]^{\mathsf{C}}_\rho \subseteq \mathsf{type}(\mathbf{v}) \subseteq [\![t']\!]^{\mathsf{C}}$. We then proceed by double inclusion.

- Let $d \in [\![(x = \mathbf{v} \in t')?\, \mathbf{e}_1 : \mathbf{e}_2]\!]^{\mathsf{C}}_\rho$. By Definition 11.5, there exists $\mathfrak{m} \in \mathscr{M}$ and $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathsf{C}}_\rho|_{\mathfrak{m}})$ such that $d = [d']^{\mathfrak{m}}$ and $d' \in [\![\mathbf{e}_1]\!]^{\mathsf{C}}_{\rho, x \mapsto S}$. By Lemma A.41, we deduce that $d' \in [\![\mathbf{e}_1\,[\mathbf{v}/x]]\!]^{\mathsf{C}}_\rho$. And by Lemma A.39, we deduce that necessarily $\mathfrak{m} = \varepsilon$, and that $d = d'$, hence the result.
- The converse is proven in the exact same way.

- $\underline{[\mathrm{R}^{\mathsf{C}}_{\mathsf{CaseR}}]}$. $(x = \mathbf{v} \in t')?\, \mathbf{e}_1 : \mathbf{e}_2 \rightsquigarrow \mathbf{e}_2\,[\mathbf{v}/x]$ where $\mathbf{v} \notin t'$. By hypothesis, $(x = \mathbf{v} \in t')?\, \mathbf{e}_1 : \mathbf{e}_2 \in \mathsf{Prg}_\Gamma(t)$, and by Proposition 11.8, this ensures that $\mathbf{v} \in \neg t'$. By Theorem 11.6, we then deduce that $[\![\mathbf{v}]\!]^{\mathsf{C}}_\rho \subseteq \mathsf{type}(\mathbf{v}) \subseteq [\![\neg t']\!]^{\mathsf{C}}$. The rest of the proof is exactly the same as for $[\mathrm{R}^{\mathsf{C}}_{\mathsf{CaseL}}]$.

- $\underline{[\mathrm{R}^{\mathsf{C}}_{\mathsf{Choice}_i}]}$. $\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2) \rightsquigarrow \mathbf{e}_i$. Let $d \in \mathscr{D}^{\mathsf{C}}$ and suppose w.l.o.g. that $i = 1$. We have by Definition 11.5 that $[d]^l \in [\![\mathsf{choice}(\mathbf{e}_1, \mathbf{e}_2)]\!]^{\mathsf{C}}_\rho \iff d \in [\![\mathbf{e}_1]\!]^{\mathsf{C}}_\rho$, which is exactly the result. The same holds for $i = 2$ considering $[d]^r$.

- $\underline{[\mathrm{R}^{\mathsf{C}}_{\mathsf{ctx}}]}$. $\mathscr{E}\,[\mathbf{e}] \rightsquigarrow \mathscr{E}\,[\mathbf{e}']$. Straightforward by induction and cases over $\mathscr{E}$, considering Definition 11.5.

□

Lemma A.47. *For every term* $\mathbf{E} \in \mathsf{Terms}^{\mathsf{FCB}}$ *and every value* $\mathbf{V} \in \mathsf{Values}^{\mathsf{FCB}}$*, if* $\Gamma, x : t \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t'$ *and* $\Gamma \vdash \mathbf{V} \rightsquigarrow \mathbf{v} : t$*, then* $\Gamma \vdash \mathbf{E}\,[\mathbf{V}/x] \rightsquigarrow \mathbf{e}\,[\mathbf{v}/x] : t'$.

Proof. By induction on the derivation of $\Gamma, x : t \vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t'$, generalized over Γ.

- $\underline{[\mathrm{C}^{\mathsf{C}}_{\mathsf{Cst}}]}$. Immediate since $\mathbf{E} = \mathbf{e} = c$.

- $\underline{[\mathrm{C}^{\mathsf{C}}_{\mathsf{Var}}]}$. We have $\mathbf{E} = \mathbf{e} = x$ and $\mathbf{E}\,[\mathbf{V}/x] = \mathbf{V}$ and $\mathbf{e}\,[\mathbf{v}/x] = \mathbf{v}$. The result follows by hypothesis.

- $\underline{[\mathrm{C}^{\mathsf{C}}_{\mathsf{Abs}}]}$. We have $\mathbf{E} = \lambda^{\bigwedge_{i \in I}(s_i \to t_i)} y.\, \mathbf{E}'$ and $\mathbf{e} = \lambda^{\bigwedge_{i \in I}(s_i \to t_i) \wedge \bigwedge_{n \in N} \neg(s_n \to t_n)} y.\, \mathbf{e}'$ where, by hypothesis, for every $i \in I$, $\Gamma, x : t, y : s_i \vdash \mathbf{E}' \rightsquigarrow \mathbf{e}' : t_i$. By IH, we have $\Gamma, y : s_i \vdash \mathbf{E}'\,[\mathbf{V}/x] \rightsquigarrow \mathbf{e}'\,[\mathbf{v}/x] : t_i$. The result follows by application of $[\mathrm{C}^{\mathsf{C}}_{\mathsf{Abs}}]$.

- $\underline{[\mathrm{C}^{\mathsf{C}}_{\mathsf{Case}}]}$. We have $\mathbf{E} \equiv (y = \mathbf{E}' \in s)?\, \mathbf{E}_1 : \mathbf{E}_2$ and $\mathbf{e} \equiv (y = \mathbf{e}' \in s)?\, \mathbf{e}_1 : \mathbf{e}_2$. By hypothesis, $\Gamma, x : t \vdash \mathbf{E}' \rightsquigarrow \mathbf{e}' : s'$, and $\Gamma, x : t, y : s \wedge s' \vdash \mathbf{E}_1 \rightsquigarrow \mathbf{e}_1 : t'$, and $\Gamma, x : t, y : \neg s \wedge s' \vdash \mathbf{E}_2 \rightsquigarrow \mathbf{e}_2 : t'$. By IH, we deduce that $\Gamma \vdash \mathbf{E}'\,[\mathbf{V}/x] \rightsquigarrow \mathbf{e}'\,[\mathbf{v}/x] : s'$, and $\Gamma, y : s \wedge s' \vdash \mathbf{E}_1\,[\mathbf{V}/x] \rightsquigarrow \mathbf{e}_1\,[\mathbf{v}/x] : t'$, and $\Gamma, y : \neg s \wedge s' \vdash \mathbf{E}_2\,[\mathbf{V}/x] \rightsquigarrow \mathbf{e}_2\,[\mathbf{v}/x] : t'$. The result follows by application of $[\mathrm{C}^{\mathsf{C}}_{\mathsf{Case}}]$.

- $\underline{[\mathrm{C}^{\mathsf{C}}_{\mathsf{Sub}}]}$. $\underline{[\mathrm{C}^{\mathsf{C}}_{\mathsf{App}}]}$. $\underline{[\mathrm{C}^{\mathsf{C}}_{\mathsf{Pair}}]}$. $\underline{[\mathrm{C}^{\mathsf{C}}_{\mathsf{Proj}_i}]}$. $\underline{[\mathrm{C}^{\mathsf{C}}_{\mathsf{Choice}}]}$. $\underline{[\mathrm{C}^{\mathsf{C}}_{\mathsf{Efq}}]}$. All these cases are immediate by induction (or direct application of $[\mathrm{C}^{\mathsf{C}}_{\mathsf{Efq}}]$ for the last case).

□

Theorem A.48 (Soundness and completeness of compilation)**.** *For every term* $\mathbf{E} \in \mathsf{Terms}^{\mathsf{FCB}}$*, if* $\vdash \mathbf{E} : t$ *then there exists* $\mathbf{e} \in \mathsf{Prg}_{\emptyset}(t)$ *such that* $\vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t$ *and the following holds:*

1. $\mathbf{E} \rightsquigarrow \mathbf{E}' \implies \exists \mathbf{e}' \in \mathsf{Terms}^{C}.\ \mathbf{e} \rightsquigarrow \mathbf{e}'$ *and* $\vdash \mathbf{E}' \rightsquigarrow \mathbf{e}' : t$
2. $\mathbf{e} \rightsquigarrow \mathbf{e}' \implies \exists \mathbf{E}' \in \mathsf{Terms}^{\mathsf{FCB}}.\ \mathbf{E} \rightsquigarrow \mathbf{E}'$ *and* $\vdash \mathbf{E}' \rightsquigarrow \mathbf{e}' : t$

Proof. By Lemma 11.19, there exists $\mathbf{e}' \in \mathsf{Terms}^{\mathsf{C}}$ such that $\vdash \mathbf{E} \rightsquigarrow \mathbf{e}' : t$. Now by Lemma 11.25 and Corollary 11.24 taking $S = \mathsf{A}^{\rightarrow}(\mathbf{e}')$, if we note $\mathbf{e} = (\!|\mathbf{e}'|\!)_S$, we obtain that $\vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t$ and $\mathbf{e} \in \mathsf{Prg}_{\emptyset}(t)$. We now prove the two properties.

1. Suppose that $\mathbf{E} \rightsquigarrow \mathbf{E}'$ for some $\mathbf{E}'$. We reason by cases over the reduction rule, with the hypothesis that $\vdash \mathbf{E} \rightsquigarrow \mathbf{e} : t$.
   - $[\mathrm{R}^{\mathsf{FCB}}_{\mathsf{App}}]$. We have $(\lambda^{\bigwedge_{i\in I}(s_i \to t_i)} x.\, \hat{\mathbf{E}})\, \mathbf{V} \rightsquigarrow \hat{\mathbf{E}}\,[\mathbf{V}/x]$. By inversion of the system of Figure 11.2, we have $\vdash \mathbf{V} \rightsquigarrow \mathbf{v} : s$ and $\mathbf{e} = (\lambda^{\bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{n\in N} \neg(s_n \to t_n)} x.\, \hat{\mathbf{e}})\, \mathbf{v}$ where for every $i \in I$, , $x : s_i \vdash \hat{\mathbf{E}} \rightsquigarrow \hat{\mathbf{e}} : t_i$. Additionally, we have $\bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{n\in N} \neg(s_n \to t_n) \leq s \to t$. By Proposition 2.22, we have that $s \leq \bigvee_{i\in I} s_i$. By hypothesis, we have $\mathbf{e} \in \mathsf{Prg}_{\emptyset}(t)$, which means that, if we write $S = \mathsf{A}^{\rightarrow}(\bigvee_{i\in I} s_i)$, $\mathscr{W}_S(\mathbf{v})$ holds. By Proposition 11.8, this yields that there exists $i \in I$ such that $\vdash \mathbf{v} : s_i$. By Lemma A.47, we obtain that $\vdash \hat{\mathbf{E}}\,[\mathbf{V}/x] \rightsquigarrow \hat{\mathbf{e}}\,[\mathbf{v}/x] : t_i$. Proposition 2.23 yields that $t_i \leq t$, and the result follows since $\mathbf{e} \rightsquigarrow \hat{\mathbf{e}}\,[\mathbf{v}/x]$.
   - $[\mathrm{R}^{\mathsf{FCB}}_{\mathsf{Proj}_i}]$. Immediate by inversion of rules $[\mathrm{C}^{\mathsf{C}}_{\mathsf{Proj}_i}]$ and $[\mathrm{C}^{\mathsf{C}}_{\mathsf{Pair}}]$.
   - $[\mathrm{R}^{\mathsf{FCB}}_{\mathsf{Choice}_i}]$. Immediate by inversion of rule $[\mathrm{C}^{\mathsf{C}}_{\mathsf{Choice}}]$.
   - $[\mathrm{R}^{\mathsf{FCB}}_{\mathsf{CaseL}}]$. We have $(x = \mathbf{V} \in s)?\, \mathbf{E}_1 : \mathbf{E}_2 \rightsquigarrow \mathbf{E}_1\,[\mathbf{V}/x]$ and $\vdash \mathbf{V} : s$. By inversion of the system of Figure 11.2, we have $\mathbf{e} = (x = \mathbf{v} \in s)?\, \mathbf{e}_1 : \mathbf{e}_2$ where $\vdash \mathbf{V} \rightsquigarrow \mathbf{v} : s'$, and $x : s \wedge s' \vdash \mathbf{E}_1 \rightsquigarrow \mathbf{e}_1 : t$ and $x : \neg s \wedge s' \vdash \mathbf{E}_2 \rightsquigarrow \mathbf{e}_2 : t$.
     By Lemma 11.21, we have that $s \wedge s' \not\simeq \mathbb{0}$. By hypothesis, we have $\mathbf{e} \in \mathsf{Prg}_{\emptyset}(t)$, which means that, if we write $S = \mathsf{A}^{\rightarrow}(s)$, $\mathscr{W}_S(\mathbf{v})$ holds. By Proposition 11.8, this yields that either $\mathbf{v} \in s$ or $\mathbf{v} \in \neg s$. However, since $\vdash \mathbf{v} : s'$ and $s \wedge s' \not\simeq \mathbb{0}$, we have necessarily $\mathbf{v} \in s$ by Lemma 11.13. Thus, we deduce that $\mathbf{e} \rightsquigarrow \mathbf{e}_1\,[\mathbf{v}/x]$ and the result follows by Lemma A.47.
   - $[\mathrm{R}^{\mathsf{FCB}}_{\mathsf{CaseR}}]$. We have $(x = \mathbf{V} \in s)?\, \mathbf{E}_1 : \mathbf{E}_2 \rightsquigarrow \mathbf{E}_2\,[\mathbf{V}/x]$ and $\not\vdash \mathbf{V} : s$. By Lemma 11.16, we have $\vdash \mathbf{V} : \neg s$, and from there we can follow the same reasoning as in the previous case.
   - $[\mathrm{R}^{\mathsf{FCB}}_{\mathsf{Ctx}}]$. Immediate by case analysis over the context $\mathscr{E}$ and application of the induction hypothesis.
2. The second part is proven using the same reasoning, except there is no need to use the hypothesis that the value is well-annotated in the cases $[\mathrm{R}^{\mathsf{C}}_{\mathsf{App}}]$, $[\mathrm{R}^{\mathsf{C}}_{\mathsf{CaseL}}]$, and $[\mathrm{R}^{\mathsf{C}}_{\mathsf{CaseR}}]$.

□

A.2.3. Denotational semantics of a cast calculus

Proposition A.49. *For every types $\tau, \tau' \in \mathrm{GTypes}$, $\langle\!\langle \tau \rangle\!\rangle \subseteq \langle\!\langle \tau' \rangle\!\rangle \iff \tau \leq \tau'$.*

Proof. For every $d \in \mathscr{D}^{\mathrm{G}}$, $\tau, \tau' \in \mathrm{GTypes}$, we prove the following two results by induction on the pair (d, τ) lexicographically ordered.

1. $\langle\!\langle \tau \rangle\!\rangle \subseteq \langle\!\langle \tau' \rangle\!\rangle$ and $(d : \tau)^{\mathrm{G}} \implies (d : \tau')^{\mathrm{G}}$. By cases on τ.
   - $\underline{\tau = b.}$ By Definition 12.5, we have $d = c^g$ such that $c \in \mathbb{B}(b)$. By Definition 12.16, we have $c^! \in \langle\!\langle \tau \rangle\!\rangle$. By hypothesis, we deduce that $c^! \in \langle\!\langle \tau' \rangle\!\rangle$. By Definition 12.16, we have that necessarily $\tau' = b'$ such that $c \in \mathbb{B}(b)$. And by Definition 12.5, we conclude that $(d : \tau')^{\mathrm{G}}$.
   - $\underline{\tau = ?.}$ By Definition 12.16, we have $\{\}^? \in \langle\!\langle \tau \rangle\!\rangle$. By hypothesis, we have $\{\}^? \in \langle\!\langle \tau' \rangle\!\rangle$. Necessarily, by Definition 12.16, $\tau' = ?$ since $?$ is the only type whose interpretation contains values tagged with $?$. Hence, $\tau = \tau' = ?$ and the result follows.
   - $\underline{\tau = \tau_1 \to \tau_2.}$ By Definition 12.16, $\{\}^! \in \langle\!\langle \tau \rangle\!\rangle$. By hypothesis, we have $\{\}^! \in \langle\!\langle \tau' \rangle\!\rangle$. Necessarily, by Definition 12.16, we obtain that $\tau' = \tau'_1 \to \tau'_2$. Now suppose that $\langle\!\langle \tau'_1 \rangle\!\rangle \not\subseteq \langle\!\langle \tau_1 \rangle\!\rangle$. There exists $d' \in \langle\!\langle \tau'_1 \rangle\!\rangle \setminus \langle\!\langle \tau_1 \rangle\!\rangle$. By Definition 12.16, we have that $\{\{d'\}, \Omega\}^! \in \langle\!\langle \tau \rangle\!\rangle \setminus \langle\!\langle \tau' \rangle\!\rangle$, which contradicts the hypothesis. Hence, $\langle\!\langle \tau'_1 \rangle\!\rangle \subseteq \langle\!\langle \tau_1 \rangle\!\rangle$. Similarly, suppose that $\langle\!\langle \tau_2 \rangle\!\rangle \not\subseteq \langle\!\langle \tau'_2 \rangle\!\rangle$. There exists $d_2 \in \langle\!\langle \tau_2 \rangle\!\rangle \setminus \langle\!\langle \tau'_2 \rangle\!\rangle$. By Definition 12.16, we have that $\{\{\mho\}, d_2\}^! \in \langle\!\langle \tau \rangle\!\rangle \setminus \langle\!\langle \tau' \rangle\!\rangle$, which once again contradicts the hypothesis. Hence, $\langle\!\langle \tau_2 \rangle\!\rangle \subseteq \langle\!\langle \tau'_2 \rangle\!\rangle$.
     Now by Definition 12.5, we have that $d = \{(S_i, \partial_i) \mid i \in I\}^g$ such that for every $i \in I$, $(\exists d' \in S_i.\ (d'^{\overline{g}} : \tau_1)^{\mathrm{G}}) \implies (\partial_i^g : \tau_2)^{\mathrm{G}}$. Let $i \in I$ and suppose that there exists $d' \in S_i$. $(d'^{\overline{g}} : \tau'_1)^{\mathrm{G}}$. By IH, we deduce that $(d'^{\overline{g}} : \tau_1)^{\mathrm{G}}$. By hypothesis, we have $(\partial_i^g : \tau_2)^{\mathrm{G}}$. And by IH again, we have $(\partial_i^g : \tau'_2)^{\mathrm{G}}$. Hence $(d : \tau')^{\mathrm{G}}$.

2. $\tau \leq \tau'$ and $\langle d : \tau \rangle \implies \langle d : \tau' \rangle$. By cases on τ.
   - $\underline{\tau = b.}$ By Definition 12.16, we have $d = c^!$ such that $c \in \mathbb{B}(b)$. By Definition 12.5, we have that $c^! \in [\![\tau]\!]^{\mathrm{G}} \subseteq [\![\tau']\!]^{\mathrm{G}}$, hence necessarily $\tau' = b'$ where $c \in \mathbb{B}(b)$. Hence the result by Definition 12.16.
   - $\underline{\tau = ?.}$ By Definition 12.5, for every $c \in \mathscr{C}$, we have $c^? \in [\![\tau]\!]^{\mathrm{G}}$ as well as $\{\}^? \in [\![\tau]\!]^{\mathrm{G}}$. Thus, we have $\{c^?, \{\}^?\} \subseteq [\![\tau']\!]^{\mathrm{G}}$. We deduce that necessarily $\tau' = ?$ by inversion of Definition 12.5 in the absence of union types. Hence the result since $\tau = \tau' = ?$.
   - $\underline{\tau = \tau_1 \to \tau_2.}$ By Definition 12.5, $\{\}^! \in [\![\tau]\!]^{\mathrm{G}}$. By hypothesis, we have $\{\}^! \in [\![\tau']\!]^{\mathrm{G}}$. Necessarily, by Definition 12.5, we obtain that $\tau' = \tau'_1 \to \tau'_2$. Now suppose that $[\![\tau'_1]\!]^{\mathrm{G}} \not\subseteq [\![\tau_1]\!]^{\mathrm{G}}$. There exists $d' \in [\![\tau'_1]\!]^{\mathrm{G}} \setminus [\![\tau_1]\!]^{\mathrm{G}}$. By Proposition 12.6, we have $d'^? \in [\![\tau'_1]\!]^{\mathrm{G}} \setminus [\![\tau_1]\!]^{\mathrm{G}}$. By Definition 12.5, we have that $\{\{d'^?\}, \Omega\}^! \in [\![\tau]\!]^{\mathrm{G}} \setminus [\![\tau']\!]^{\mathrm{G}}$, which contradicts the hypothesis. Hence, $[\![\tau'_1]\!]^{\mathrm{G}} \subseteq [\![\tau_1]\!]^{\mathrm{G}}$. Similarly, suppose that $[\![\tau_2]\!]^{\mathrm{G}} \not\subseteq [\![\tau'_2]\!]^{\mathrm{G}}$. There exists $d_2 \in [\![\tau_2]\!]^{\mathrm{G}} \setminus [\![\tau'_2]\!]^{\mathrm{G}}$. By Proposition 12.6, we have $d_2^? \in [\![\tau_2]\!]^{\mathrm{G}}$. By Definition 12.5, we have that $\{\{\mho\}, d_2^?\}^! \in [\![\tau]\!]^{\mathrm{G}} \setminus [\![\tau']\!]^{\mathrm{G}}$, which once again contradicts the hypothesis. Hence, $[\![\tau_2]\!]^{\mathrm{G}} \subseteq [\![\tau'_2]\!]^{\mathrm{G}}$.
     Now by Definition 12.16, we have that $d = \{(\iota_i, \partial_i) \mid i \in I\}^!$ such that for every

$i \in I$, $\iota_i = S_i \implies (\exists d' \in S_i.\ \langle d' : \tau_1 \rangle) \implies \langle \partial_i : \tau_2 \rangle$ and $\iota_i = \mho \implies \langle \partial_i : \tau_2 \rangle$. Let $i \in I$ and suppose that $\iota_i = S_i$ and there exists $d' \in S_i.\ \langle d' : \tau_1' \rangle$. By IH, we deduce that $\langle d' : \tau_1 \rangle$. By hypothesis, we have $\langle \partial_i : \tau_2 \rangle$. And by IH again, we have $\langle \partial_i : \tau_2' \rangle$. Now if $\iota_i = \mho$, we have $\langle \partial_i : \tau_2 \rangle$ by hypothesis, which yields $\langle \partial_i : \tau_2' \rangle$ by IH. Hence $\langle d : \tau' \rangle$.

□

Lemma A.50. *For every cast $\langle \tau \Rightarrow_p \tau' \rangle$ and every $\partial \in \mathscr{D}_\Omega^G$, if $\partial \in \langle\!\langle \tau \rangle\!\rangle$ then $\partial \langle \tau \Rightarrow_p \tau' \rangle \subseteq \langle\!\langle \tau' \rangle\!\rangle$.*

Proof. By induction on ∂ and cases on τ, τ'. In every case, we consider a $\partial' \in \partial \langle \tau \Rightarrow_p \tau' \rangle$ and prove that $\partial' \in \langle\!\langle \tau' \rangle\!\rangle$.

- $\underline{\partial = c^g.}$ We distinguish the following cases on τ'.
  - $\tau' = ?$. In that case, $c^g \langle \tau \Rightarrow_p ? \rangle = \{c^?\}$ by Definition 12.18, and the result follows since $c^? \in \langle\!\langle ? \rangle\!\rangle$.
  - $b_c \leq \tau'$, in which case $c^g \langle \tau \Rightarrow_p \tau' \rangle = \{c^!\}$, and by Definition 12.16, we have $c^! \in \langle\!\langle b_c \rangle\!\rangle$, and the result follows by Proposition 12.17.
  - $b_c \not\leq \tau'$. We have $c^g \langle \tau \Rightarrow_p \tau' \rangle = \{\text{blame } p\}$ by Definition 12.18, and the result is immediate by Definition 12.16.
- $\underline{\partial = \{(S_i, \partial_i) \mid i \in I\}^g.}$ We distinguish the following cases.
  - $\tau = \tau_1 \to \tau_2$ and $\tau' = \tau_1' \to \tau_2'$. By hypothesis and Definition 12.16, necessarily $g = !$. By Definition 12.18, ∂' is necessarily a finite relation, hence we can write $\partial' = \{(S_j, \partial_j) \mid j \in J\}^!$. Let $j \in J$ such that $S_j \cap \langle\!\langle \tau_1' \rangle\!\rangle \neq \emptyset$. By Definition 12.18, we obtain that necessarily $S_j \subseteq \langle\!\langle \tau_1' \rangle\!\rangle$ and we distinguish two cases.
    1. $\exists i \in I$ such that $S_i \subseteq S_j \langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle$ and $\partial_j \in \partial_i \langle \tau_2 \Rightarrow_p \tau_2' \rangle$. By IH, we deduce that $S_i \subseteq \langle\!\langle \tau_1 \rangle\!\rangle$. By Definition 12.16, we obtain that $\partial_i \in \langle\!\langle \tau_2 \rangle\!\rangle$. By IH, this yields that $\partial_j \in \langle\!\langle \tau_2' \rangle\!\rangle$.
    2. blame $q \in S_j \langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle$ and $\partial_j = \text{blame } q$. By Definition 12.16, we immediately have $\partial_j \in \langle\!\langle \tau_2' \rangle\!\rangle$. Hence, $\partial' \in \langle\!\langle \tau_1' \to \tau_2' \rangle\!\rangle$.
  - $\tau = \tau_1 \to \tau_2$ and $\tau' = ?$. By hypothesis and Definition 12.16, necessarily $g = !$. By Definition 12.18, we have that $\partial' = \partial''^?$ for some $\partial'' \in \partial \langle \tau_1 \to \tau_2 \Rightarrow_p ? \to ? \rangle$. By applying the same reasoning as in the previous case, we deduce that $\partial'' \in \langle\!\langle ? \to ? \rangle\!\rangle$, which yields that $\partial' \in \langle\!\langle ? \rangle\!\rangle$ by Definition 12.16.
  - $\tau = ?$ and $\tau' = \tau_1' \to \tau_2'$. By hypothesis and Definition 12.16, necessarily $g = ?$, and we have that $\partial^! \in \langle\!\langle ? \to ? \rangle\!\rangle$. Thus, by Definition 12.18, we deduce that $\partial' \in \partial^! \langle ? \to ? \Rightarrow_p \tau_1' \to \tau_2' \rangle$, and by IH, this yields $\partial' \in \langle\!\langle \tau_1' \to \tau_2 \rangle\!\rangle$.
  - $\tau = ?$ and $\tau' = ?$. Immediate by Definition 12.18.
  - $\tau = ?$ and $\tau' = b$. By Definition 12.18, we have $\partial' = \text{blame } p$ and the result follows immediately by Definition 12.16.
- $\underline{\partial = \text{blame } q.}$. We have $\text{blame } q \langle \tau \Rightarrow_p \tau' \rangle = \{\text{blame } q\}$ by Definition 12.18, and the result follows by Definition 12.16.
- $\underline{\partial = \Omega.}$. Impossible since we cannot have $\Omega \in \langle\!\langle \tau \rangle\!\rangle$.

□

Theorem A.51 (Type soundness for λ_{G})**.** *For every type environment* $\Gamma \in \mathsf{TEnvs}$ *and every term* $\mathbf{E} \in \mathsf{Terms}^G$*, if* $\Gamma \vdash \mathbf{E} : \tau$ *then for every* $\rho \in [\![\Gamma]\!]^G$, $[\![\mathbf{E}]\!]^G_\rho \subseteq \langle\!\langle \tau \rangle\!\rangle$.

Proof. By induction on $\mathbf{E}$, generalized on Γ. All cases are proven by inversion of the typing rules.

- $\underline{\mathbf{E} = c.}$ By inversion of the typing rules, $b_c \leq \tau$. By Definition 12.19, $[\![c]\!]^{\mathrm{G}}_\rho = \{c^!\} \subseteq \langle\!\langle b_c \rangle\!\rangle$, and the result follows by Proposition 12.17.
- $\underline{\mathbf{E} = x.}$. By inversion of the typing rules, $\Gamma(x) \leq \tau$. The result follows immediately by Definition 12.22 and Proposition 12.17.
- $\underline{\mathbf{E} = \lambda x{:}\tau'.\, \mathbf{E}'.}$ By inversion of the typing rules, $\Gamma, x : \tau' \vdash \mathbf{E}' : \tau''$ where $\tau' \to \tau'' \leq \tau$. Now let $d \in [\![\mathbf{E}]\!]^{\mathrm{G}}_\rho$. By inversion of Definition 12.19, we have $d = R^!$ where for every $(S, \partial) \in R$, $\mho \notin S$ and either $S \subseteq \langle\!\langle \tau' \rangle\!\rangle$ and $\partial \in [\![\mathbf{e}']\!]^{\mathrm{G}}_{\rho, x \mapsto S}$, or $S \cap \langle\!\langle \tau' \rangle\!\rangle = \emptyset$ and $\partial = \Omega$. Consider $(S, \partial) \in R$ such that $\exists d \in S,\, d \in \langle\!\langle \tau' \rangle\!\rangle$. This ensures that $S \cap \langle\!\langle \tau' \rangle\!\rangle \neq \emptyset$, thus, necessarily, we have $S \subseteq \langle\!\langle \tau' \rangle\!\rangle$ and $\partial \in [\![\mathbf{e}']\!]^{\mathrm{G}}_{\rho, x \mapsto S}$. We have, by Definition 12.22, $(\rho, x \mapsto S) \in [\![\Gamma, x : \tau']\!]^{\mathrm{G}}$. Hence, by IH, we deduce that $\partial \in \langle\!\langle \tau'' \rangle\!\rangle$, which proves that $R^! \in \langle\!\langle \tau' \to \tau'' \rangle\!\rangle$ by Definition 12.16, and the result follows by Proposition 12.17.
- $\underline{\mathbf{E} = \mathbf{E}_1\, \mathbf{E}_2.}$ By inversion of the typing rules, $\Gamma \vdash \mathbf{E}_1 : \tau'' \to \tau'$, $\Gamma \vdash \mathbf{E}_2 : \tau''$, and $\tau' \leq \tau$. Let $\partial \in [\![\mathbf{E}]\!]^{\mathrm{G}}_\rho$. If $\partial \in \mathbb{B}\mathsf{lame}$, the result is immediate since $\mathbb{B}\mathsf{lame} \subseteq \langle\!\langle \tau \rangle\!\rangle$ by Definition 12.16. Moreover, by induction hypothesis, $\Omega \notin [\![\mathbf{E}_1]\!]^{\mathrm{G}}_\rho \cup [\![\mathbf{E}_2]\!]^{\mathrm{G}}_\rho$, and by IH, $[\![\mathbf{E}_1]\!]^{\mathrm{G}}_\rho \subseteq \langle\!\langle \tau'' \to \tau' \rangle\!\rangle$. This ensures by Definition 12.16 and Definition 12.21 that $\Omega \notin \Omega^\rho_{\mathbf{E}_1\, \mathbf{E}_2}$.
  The only case left is when there exists $R^! \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_\rho$ and $S \subseteq [\![\mathbf{E}_2]\!]^{\mathrm{G}}_\rho$ such that $(S, \partial) \in R$. By induction hypothesis, $R \in \langle\!\langle \tau'' \to \tau' \rangle\!\rangle$ and $S \subseteq \langle\!\langle \tau'' \rangle\!\rangle$. By Definition 12.16, we deduce that $\partial \in \langle\!\langle \tau' \rangle\!\rangle$, hence the result by Proposition 12.17.
- $\underline{\mathbf{E} = \mathbf{E}'\langle \tau' \Rightarrow_p \tau'' \rangle}$. Suppose that $p = \ell$. The case $p = \bar{\ell}$ is proven identically. By inversion of the typing rules, $\Gamma \vdash \mathbf{E}' : \tau'$ and $\tau' \preccurlyeq \tau''$. By IH, we obtain that $[\![\mathbf{E}']\!]^{\mathrm{G}}_\rho \subseteq \langle\!\langle \tau' \rangle\!\rangle$. Let $\partial \in [\![\mathbf{E}]\!]^{\mathrm{G}}_\rho$. By Definition 12.19, there exists $\partial' \in [\![\mathbf{E}']\!]^{\mathrm{G}}_\rho$ such that $\partial \in \partial'\langle \tau' \Rightarrow_p \tau'' \rangle$. By Lemma A.50, we deduce that $\partial \in \langle\!\langle \tau'' \rangle\!\rangle$, and the result follows by Proposition 12.17.

□

Lemma A.52. *For every term* $\mathbf{E} \in \mathsf{Terms}^G$*,* $x \in \mathsf{Vars}$*,* $\rho \in \mathsf{Envs}$*, and* $S_1, S_2 \in \mathscr{P}(\mathscr{D}^G)$*, if* $S_1 \subseteq S_2$ *then* $[\![\mathbf{E}]\!]^G_{\rho, x \mapsto S_1} \subseteq [\![\mathbf{E}]\!]^G_{\rho, x \mapsto S_2}$

Proof. The proof is done by structural induction on $\mathbf{E} \in \mathsf{Terms}^{\mathrm{G}}$, generalized over ρ.

- $\underline{\mathbf{E} = c.}$ Immediate since x does not appear in $\mathbf{E}$.
- $\underline{\mathbf{E} = y.}$ If $y \neq x$ then the result is immediate. Otherwise, $[\![x]\!]^{\mathrm{G}}_{\rho, x \mapsto S_1} = S_1 \subseteq S_2 = [\![x]\!]^{\mathrm{G}}_{\rho, x \mapsto S_2}$ which gives the result.

- $\underline{\mathbf{E} = \lambda y{:}t.\,\mathbf{E}'}$. Let $R^! \in [\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$. Let $(S,\partial) \in R$. If $S \subseteq \langle\!\langle t \rangle\!\rangle$ then $\partial \in [\![\mathbf{E}']\!]^{\mathrm{G}}_{\rho,x\mapsto S_1,y\mapsto S}$. By generalized induction hypothesis, $\partial \in [\![\mathbf{E}']\!]^{\mathrm{G}}_{\rho,x\mapsto S_2,y\mapsto S}$. Moreover, if $S \not\subseteq \langle\!\langle t \rangle\!\rangle$ then $\partial = \Omega$ independently of S_1 and S_2. Therefore, $R^! \in [\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2}$.

- $\underline{\mathbf{E} = \mathbf{E}_1\,\mathbf{E}_2}$. Let $\partial \in [\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$. We distinguish the following cases.
  1. There exists $R^! \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$ and $S \subseteq [\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$ such that $(S,\partial) \in R$. By induction hypothesis, $R^! \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2}$ and $S \subseteq [\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2}$, thus $\partial \in [\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2}$.
  2. $\partial = \Omega$ and $\Omega \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$. By induction hypothesis, $\Omega \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2}$ and the result follows.
  3. $\partial = \Omega$ and $\Omega \in [\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$ and $[\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1} \neq \emptyset$. By induction hypothesis, $\Omega \in [\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2}$ and $[\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2} \neq \emptyset$. Thus, the result follows.
  4. $\partial = \Omega$ and there exists $d \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$ such that $\nexists R \in \mathscr{P}_f(\mathscr{F} \times \mathscr{D}^{\mathrm{G}}_{\Omega})$, $d = R^!$, and $[\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1} \neq \emptyset$. By induction hypothesis, $d \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2}$ and $[\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2} \neq \emptyset$. This yields the result.
  5. $\partial = \mathsf{blame}\ p$ and $\mathsf{blame}\ p \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$. By IH, $\mathsf{blame}\ p \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2}$ and the result follows by Definition 12.20.
  6. $\partial = \mathsf{blame}\ p$ and $\mathsf{blame}\ p \in [\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$ and $[\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1} \neq \emptyset$. By IH, $\mathsf{blame}\ p \in [\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2}$ and $[\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2} \neq \emptyset$. The result follows by Definition 12.20.

- $\underline{\mathbf{E} = \mathbf{E}'\langle \tau \Rightarrow_p \tau' \rangle}$. Let $\partial \in [\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$. By Definition 12.19, there exists $\partial' \in [\![\mathbf{E}']\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$ such that $\partial \in \partial'\langle \tau \Rightarrow_p \tau' \rangle$. By IH, $\partial' \in [\![\mathbf{E}']\!]^{\mathrm{G}}_{\rho,x\mapsto S_2}$ and by Definition 12.19, $\partial \in [\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2}$.

□

Lemma A.53. *For every term* $\mathbf{E} \in \mathsf{Terms}^G$, $\mathbf{v} \in \mathsf{Values}^G$, $x \in \mathsf{Vars}$, $\rho \in \mathsf{Envs}$,

$$[\![\mathbf{E}\,[\mathbf{v}/x]]\!]^G_\rho = \bigcup_{S \in \mathscr{P}_f([\![\mathbf{v}]\!]^G_\rho)} [\![\mathbf{E}]\!]^G_{\rho,x\mapsto S}$$

Proof. The proof is done by structural induction on $\mathbf{E} \in \mathsf{Terms}^{\mathrm{G}}$.

- $\underline{\mathbf{E} = c}$. Immediate since x does not appear in c.

- $\underline{\mathbf{E} = y}$. If $y \neq x$ the result is immediate. Otherwise, if $y = x$ we have $[\![\mathbf{E}\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_\rho = [\![\mathbf{v}]\!]^{\mathrm{G}}_\rho$ and we reason by double inclusion.
  - Consider $d \in [\![\mathbf{v}]\!]^{\mathrm{G}}_\rho$. Then we immediately have the result taking $S = \{d\}$: $[\![x]\!]^{\mathrm{G}}_{\rho,x\mapsto\{d\}} = \{d\}$.
  - Let $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{G}}_\rho)$ and consider $d \in [\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho,x\mapsto S}$. Since $[\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho,x\mapsto S} = S$, we have $d \in S$. And since $S \subseteq [\![\mathbf{v}]\!]^{\mathrm{G}}_\rho$, $d \in [\![\mathbf{v}]\!]^{\mathrm{G}}_\rho$.

- $\underline{\mathbf{E} = \lambda y{:}t.\,\mathbf{E}'}$. By definition, $[\![\mathbf{E}\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_\rho = [\![\lambda y{:}t.\,(\mathbf{E}'\,[\mathbf{v}/x])]\!]^{\mathrm{G}}_\rho$. We proceed by double inclusion.
  - Let $R^! \in [\![\mathbf{E}\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_\rho$, and let us write $R = \{(S_i,\partial_i) \mid i \in I\}$. Let $i \in I$. If $S_i \subseteq \langle\!\langle t \rangle\!\rangle$,

then by Definition 12.19 we have $\partial_i \in [\![\mathbf{E}'\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho,y\mapsto S_i}$. By induction hypothesis, there exists $S_i^{\mathrm{v}} \subseteq [\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho}$ such that $\partial_i \in [\![\mathbf{E}']\!]^{\mathrm{G}}_{\rho,y\mapsto S_i,x\mapsto S_i^{\mathrm{v}}}$. Moreover, if $S_i \not\subseteq \langle\!\langle t \rangle\!\rangle$ then $\partial = \Omega$ by definition. So in this case, we just pose $S_i^{\mathrm{v}} = \emptyset$.
Now consider $S^{\mathrm{v}} = \bigcup_{i\in I} S_i^{\mathrm{v}}$. Since I is finite, it is immediate that $S^{\mathrm{v}} \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho})$. Moreover, by Lemma A.52, we deduce that $\forall i \in I$, if $S_i \subseteq \langle\!\langle t \rangle\!\rangle$, then $\partial_i \in [\![\mathbf{E}']\!]^{\mathrm{G}}_{\rho,y\mapsto S_i,x\mapsto S^{\mathrm{v}}}$. Thus, this proves that $R^! \in [\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho,x\mapsto S^{\mathrm{v}}}$.

– Let $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho})$ and $R^! \in [\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho,x\mapsto S}$. Let $(S_i, \partial_i) \in R$. Note that if $S_i \not\subseteq \langle\!\langle t \rangle\!\rangle$ then $\partial = \Omega$ by definition, so this part is immediate. Now suppose that $S_i \subseteq \langle\!\langle t \rangle\!\rangle$. By definition, $\partial \in [\![\mathbf{E}']\!]^{\mathrm{G}}_{\rho,x\mapsto S,y\mapsto S_i}$. By induction hypothesis, this yields $\partial \in [\![\mathbf{E}'\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho,y\mapsto S_i}$ and thus $R^! \in [\![\mathbf{E}\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho}$.

- $\underline{\mathbf{E} = \mathbf{E}_1\,\mathbf{E}_2.}$ We prove the result by double inclusion. Let $\partial \in [\![\mathbf{E}\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho}$. We distinguish several cases.

  1. There exists $R^! \in [\![\mathbf{E}_1\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho}$ and $S \in \mathscr{P}_f([\![\mathbf{E}_2\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho})$ such that $(S, \partial) \in R$. By induction hypothesis, we deduce that there exists $S_1, S_2 \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho})$ such that $R^! \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$ and $S \subseteq [\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2}$. Taking $S = S_1 \cup S_2$ and applying Lemma A.52 yields that $R^! \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S}$ and $S \subseteq [\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S}$, thus $\partial \in [\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho,x\mapsto S}$.
  2. $\partial = \Omega$ where $\Omega \in [\![\mathbf{E}_1\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho}$. By induction hypothesis, there exists $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho})$ such that $\Omega \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S}$ and the result follows.
  3. $\partial = \Omega$ where $\Omega \in [\![\mathbf{E}_2\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho}$ and $[\![\mathbf{E}_1\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho} \neq \emptyset$. By induction hypothesis, there exists $S_1, S_2 \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho})$ such that $\Omega \in [\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$ and $[\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2} \neq \emptyset$. Taking $S = S_1 \cup S_2$ and applying Lemma A.52 yields that $\Omega \in [\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S}$ and $[\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S} \neq \emptyset$, hence the result.
  4. $\partial = \Omega$ where there exists $d \in [\![\mathbf{E}_1\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho}$ such that $\nexists R \in \mathscr{P}_f(\mathscr{F} \times \mathscr{D}^{\mathrm{G}}_{\Omega})$, $d = R^!$, and $[\![\mathbf{E}_2\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho} \neq \emptyset$. By induction hypothesis, there exists $S_1, S_2 \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho})$ such that $d \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1}$ and $[\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2} \neq \emptyset$. Taking $S = S_1 \cup S_2$ and applying Lemma A.52 yields that $d \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S}$ and $[\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S} \neq \emptyset$. Hence the result by Definition 12.21.
  5. $\partial = \mathsf{blame}\,p$ where $\mathsf{blame}\,p \in [\![\mathbf{E}_1\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho}$. By induction hypothesis, there exists $S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho})$ such that $\mathsf{blame}\,p \in [\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S}$, thus $\mathsf{blame}\,p \in [\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho,x\mapsto S}$.
  6. $\partial = \mathsf{blame}\,p$ where $\mathsf{blame}\,p \in [\![\mathbf{E}_2\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho}$ and $[\![\mathbf{E}_1\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho} \neq \emptyset$. By induction hypothesis, there exists $S_1, S_2 \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho})$ such that $\mathsf{blame}\,p \in [\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S_2}$ and $[\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S_1} \neq \emptyset$. Taking $S = S_1 \cup S_2$ and applying Lemma A.52 yields that $\mathsf{blame}\,p \in [\![\mathbf{E}_2]\!]^{\mathrm{G}}_{\rho,x\mapsto S}$ and $[\![\mathbf{E}_1]\!]^{\mathrm{G}}_{\rho,x\mapsto S} \neq \emptyset$. Thus, we have $\mathsf{blame}\,p \in [\![\mathbf{E}]\!]^{\mathrm{G}}_{\rho,x\mapsto S}$.

  The same reasoning proves the other inclusion.

- $\underline{\mathbf{E} = \mathbf{E}'\langle \tau \Rightarrow_p \tau' \rangle.}$ We have

$$[\![\mathbf{E}\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho} = [\![\mathbf{E}'\,[\mathbf{v}/x]\langle \tau \Rightarrow_p \tau' \rangle]\!]^{\mathrm{G}}_{\rho} = \bigcup_{\partial \in [\![\mathbf{E}'\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho}} \partial\langle \tau \Rightarrow_p \tau' \rangle$$

By IH, we have $[\![\mathbf{E}'\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho} = \bigcup_{S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho})} [\![\mathbf{E}']\!]^{\mathrm{G}}_{\rho, x \mapsto S}$. Thus, we have

$$[\![\mathbf{E}\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho} = \bigcup_{S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho})} \bigcup_{\partial \in [\![\mathbf{E}']\!]^{\mathrm{G}}_{\rho, x \mapsto S}} \partial\langle \tau \Rightarrow_p \tau' \rangle$$

And by Definition 12.19, this yields exactly

$$[\![\mathbf{E}\,[\mathbf{v}/x]]\!]^{\mathrm{G}}_{\rho} = \bigcup_{S \in \mathscr{P}_f([\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho})} [\![\mathbf{E}'\langle \tau \Rightarrow_p \tau' \rangle]\!]^{\mathrm{G}}_{\rho, x \mapsto S}$$

□

Lemma A.54. *For every value* $\mathbf{v} \in \mathsf{Values}^G$ *and every* $\rho \in \mathsf{Envs}$, $[\![\mathbf{v}]\!]^G_{\rho} \neq \emptyset$.

Proof. By induction on $\mathbf{v}$.

- $\mathbf{v} = c$. By Definition 12.19, $[\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho} = \{c^{!}\}$.
- $\mathbf{v} = \lambda x{:}\tau.\,\mathbf{E}$. By Definition 12.19, $\{\} \in [\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho}$.
- $\mathbf{v} = \mathbf{v}'\langle \eta \Rightarrow_p ? \rangle$. By IH, there exists $\partial \in [\![\mathbf{v}']\!]^{\mathrm{G}}_{\rho}$. And for every τ, τ', by Definition 12.18, it holds that $\partial\langle \tau \Rightarrow_{\neq} \tau' \rangle \emptyset$. Hence, $\partial\langle \eta \Rightarrow_p ? \rangle \neq \emptyset$, and the result follows by Definition 12.19.
- $\mathbf{v} = \mathbf{v}'\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle$. A reasoning identical to the previous case yields the result.

□

Lemma A.55. *For every value* $\mathbf{v} \in \mathsf{Values}^G$ *and every* $\rho \in \mathsf{Envs}$, $[\![\mathbf{v}]\!]^G_{\rho} \cap \mathbb{B}\mathsf{lame} = \emptyset$.

Proof. By induction on $\mathbf{v}$.

- $\mathbf{v} = c$. By Definition 12.19, $[\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho} = \{c^{!}\}$ therefore $[\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho} \cap \mathbb{B}\mathsf{lame} = \emptyset$.
- $\mathbf{v} = \lambda x{:}\tau.\,\mathbf{E}$. Immediate by Definition 12.19.
- $\mathbf{v} = \mathbf{v}'\langle \eta \Rightarrow_p ? \rangle$. By IH, there exists $[\![\mathbf{v}']\!]^{\mathrm{G}}_{\rho} \cap \mathbb{B}\mathsf{lame} = \emptyset$. And by Definition 12.18, there is no case where $\partial\langle \tau \Rightarrow_p \tau' \rangle \cap \mathbb{B}\mathsf{lame} \neq \emptyset$ where $\partial \notin \mathbb{B}\mathsf{lame}$ and $\tau' \neq ?$.
- $\mathbf{v} = \mathbf{v}'\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle$. By IH, there exists $[\![\mathbf{v}']\!]^{\mathrm{G}}_{\rho} \cap \mathbb{B}\mathsf{lame} = \emptyset$. And by Definition 12.18, we immediately have that for every $\partial \notin \mathbb{B}\mathsf{lame}$, $\partial\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle \cap \mathbb{B}\mathsf{lame} = \emptyset$.

□

Theorem A.56 (Computational soundness for λ_{G})**.** *For every term* $\mathbf{E} \in \mathsf{Terms}^G$ *such that*

$\Gamma \vdash \mathbf{E} : \tau$ *and every environment* $\rho \in [\![\Gamma]\!]^G$,

$$\mathbf{E} \rightsquigarrow \mathbf{E}' \implies [\![\mathbf{E}]\!]^G_\rho = [\![\mathbf{E}']\!]^G_\rho$$
$$\mathbf{E} \rightsquigarrow \mathsf{blame}\ p \implies [\![\mathbf{E}]\!]^G_\rho = \{\mathsf{blame}\ p\}$$

Proof. The proof is done by structural induction on $\mathbf{E} \in \mathsf{Terms}^{\mathrm{G}}$ and cases over the reduction rule used for $\mathbf{E} \rightsquigarrow \mathbf{E}'$.

- $[\mathrm{R}^{\mathrm{G}}_{\mathsf{App}}]$. $(\lambda x{:}\tau'.\,\mathbf{E})\,\mathbf{v} \rightsquigarrow \mathbf{E}\,[\mathbf{v}/x]$. Follows from Lemma 12.26 and Definition 12.19.

- $[\mathrm{R}^{\mathrm{G}}_{\mathsf{CApp}}]$. $(\mathbf{v}\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle)\,\mathbf{v}' \rightsquigarrow (\mathbf{v}\mathbf{v}'\langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle)\langle \tau_2 \Rightarrow_p \tau_2' \rangle$.
  Let $\partial' \in [\![(\mathbf{v}\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle)\,\mathbf{v}']\!]^{\mathrm{G}}_\rho$. By Theorem 12.24, $\partial' \neq \Omega$. By Definition 12.19, we distinguish two cases.
  - $\exists R'^{!} \in [\![\mathbf{v}\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle]\!]^{\mathrm{G}}_\rho$ and $S' \subseteq [\![\mathbf{v}']\!]^{\mathrm{G}}_\rho$ such that $(S', \partial') \in R'$. By Definition 12.19, there exists $R^{!} \in [\![\mathbf{v}]\!]^{\mathrm{G}}_\rho$ such that $R'^{!} \in R^{!}\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle$. By Definition 12.18, there are two more cases.
    - There exists $(S, \partial) \in R$ such that $S \subseteq S'\langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle$ and $\partial' \in \partial\langle \tau_2 \Rightarrow_p \tau_2' \rangle$. Since $S' \subseteq [\![\mathbf{v}']\!]^{\mathrm{G}}_\rho$, we have that $S \subseteq [\![\mathbf{v}'\langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle]\!]^{\mathrm{G}}_\rho$ by Definition 12.19. Thus, since $R^{!} \in [\![\mathbf{v}]\!]^{\mathrm{G}}_\rho$, we deduce that $\partial \in [\![\mathbf{v}\mathbf{v}'\langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle]\!]^{\mathrm{G}}_\rho$, and we conclude that $\partial' \in [\![(\mathbf{v}\mathbf{v}'\langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle)\langle \tau_2 \Rightarrow_p \tau_2' \rangle]\!]^{\mathrm{G}}_\rho$.
    - $\partial' = \mathsf{blame}\ q$ where $\mathsf{blame}\ q \in S'\langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle$. Since $S' \subseteq [\![\mathbf{v}']\!]^{\mathrm{G}}_\rho$, we have that $\mathsf{blame}\ q \in [\![\mathbf{v}'\langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle]\!]^{\mathrm{G}}_\rho$. And by Definition 12.20, since $[\![\mathbf{v}]\!]^{\mathrm{G}}_\rho \neq \emptyset$ by Lemma A.54, we have $\mathsf{blame}\ q \in [\![\mathbf{v}\mathbf{v}'\langle \tau_1' \Rightarrow_{\overline{p}} \tau_1 \rangle]\!]^{\mathrm{G}}_\rho$, and we conclude that $\partial' = \mathsf{blame}\ q \in [\![(\mathbf{v}\mathbf{v}'\langle \tau_1' \Rightarrow_p \tau_1 \rangle)\langle \tau_2 \Rightarrow_p \tau_2' \rangle]\!]^{\mathrm{G}}_\rho$ by Definition 12.18.
  - $\partial' = \mathsf{blame}\ q$ where $\mathsf{blame}\ q \in [\![\mathbf{v}\langle \tau_1 \to \tau_2 \Rightarrow_p \tau_1' \to \tau_2' \rangle]\!]^{\mathrm{G}}_\rho$ or $\mathsf{blame}\ q \in [\![\mathbf{v}']\!]^{\mathrm{G}}_\rho$. By Lemma A.55, the latter is impossible. For the former to hold, by Definition 12.18, this would mean that $\mathsf{blame}\ q \in [\![\mathbf{v}]\!]^{\mathrm{G}}_\rho$. Once again, this cannot hold by Lemma A.55.

  A similar reasoning proves the second inclusion.

- $[\mathrm{R}^{\mathrm{G}}_{\mathsf{Id}}]$. $\mathbf{v}\langle ? \Rightarrow_p ? \rangle \rightsquigarrow \mathbf{v}$. By Theorem 12.24, we have $[\![\mathbf{v}]\!]^{\mathrm{G}}_\rho \subseteq \langle\!\langle ? \rangle\!\rangle$. By Definition 12.18, it is straightforward to see that for every $\partial \in \langle\!\langle ? \rangle\!\rangle$, $\partial\langle ? \Rightarrow_p ? \rangle = \{\partial\}$, and the result immediately follows from this equality.

- $[\mathrm{R}^{\mathrm{G}}_{\mathsf{ExpandL}}]$. $\mathbf{v}\langle \tau_1 \to \tau_2 \Rightarrow_p ? \rangle \rightsquigarrow \mathbf{v}\langle \tau_1 \to \tau_2 \Rightarrow_p ? \to ? \rangle\langle ? \to ? \Rightarrow_p ? \rangle$ where $\tau_1 \to \tau_2 \neq ? \to ?$. Let $\partial \in [\![\mathbf{v}\langle \tau_1 \to \tau_2 \Rightarrow_p ? \rangle]\!]^{\mathrm{G}}_\rho$. By Definition 12.19, there exists $\partial' \in [\![\mathbf{v}]\!]^{\mathrm{G}}_\rho$ such that $\partial \in \partial'\langle \tau_1 \to \tau_2 \Rightarrow_p ? \rangle$. By Theorem 12.24, we have $[\![\mathbf{v}]\!]^{\mathrm{G}}_\rho \subseteq \langle\!\langle \tau_1 \to \tau_2 \rangle\!\rangle$. Thus, necessarily $\partial' = R'^{!}$, and by Definition 12.18, we have $\partial = R^{?}$ where $R^{!} \in R'^{!}\langle \tau_1 \to \tau_2 \Rightarrow_p ? \to ? \rangle$. Thus we have $R^{!} \in [\![\mathbf{v}\langle \tau_1 \to \tau_2 \Rightarrow_p ? \to ? \rangle]\!]^{\mathrm{G}}_\rho$ and we then deduce that $\partial \in [\![\mathbf{v}\langle \tau_1 \to \tau_2 \Rightarrow_p ? \to ? \rangle\langle ? \to ? \Rightarrow_p ? \rangle]\!]^{\mathrm{G}}_\rho$. The same reasoning yields the other inclusion.

- $\underline{[\mathrm{R}^{\mathrm{G}}_{\mathrm{ExpandR}}]}$. $\mathbf{v}\langle ? \Rightarrow_p \tau_1 \to \tau_2\rangle \rightsquigarrow \mathbf{v}\langle ? \Rightarrow_p ? \to ?\rangle\langle ? \to ? \Rightarrow_p \tau_1 \to \tau_2\rangle$ where $\tau_1 \to \tau_2 \neq ? \to ?$. Let $\partial \in [\![\mathbf{v}\langle ? \Rightarrow_p \tau_1 \to \tau_2\rangle]\!]^{\mathrm{G}}_\rho$. By Definition 12.19, there exists $\partial' \in [\![\mathbf{v}]\!]^{\mathrm{G}}_\rho$ such that $\partial \in \partial'\langle ? \Rightarrow_p \tau_1 \to \tau_2\rangle$. By Theorem 12.24, we have $[\![\mathbf{v}]\!]^{\mathrm{G}}_\rho \subseteq \langle\!\langle ? \rangle\!\rangle$, and by Lemma A.55, we deduce that $[\![\mathbf{v}]\!]^{\mathrm{G}}_\rho \cap \mathbb{B}\mathrm{lame} = \emptyset$. Thus, we distinguish the following two cases.
  1. $\partial' = c^?$. By Definition 12.18, we have $c^?\langle ? \Rightarrow_p \tau_1 \to \tau_2\rangle = \{\mathsf{blame}\ p\}$ as well as $c^?\langle ? \Rightarrow_p ? \to ?\rangle = \{\mathsf{blame}\ p\}$. Moreover, we have $\mathsf{blame}\ p\langle ? \to ? \Rightarrow_p \tau_1 \to \tau_2\rangle = \{\mathsf{blame}\ p\}$, which yields that $[\![\mathbf{v}\langle ? \Rightarrow_p \tau_1 \to \tau_2\rangle]\!]^{\mathrm{G}}_\rho = [\![\mathbf{v}\langle ? \Rightarrow_p ? \to ?\rangle\langle ? \to ? \Rightarrow_p \tau_1 \to \tau_2\rangle]\!]^{\mathrm{G}} = \{\mathsf{blame}\ p\}$.
  2. $\partial' = R'^?$. By Definition 12.18, we have $\partial = R^!$ where $R^! \in R'^!\langle ? \to ? \Rightarrow_p \tau_1 \to \tau_2\rangle$. And since $R'^! \in \partial'\langle ? \Rightarrow_p ? \to ?\rangle$, we have $\partial \in [\![\mathbf{v}\langle ? \Rightarrow_p ? \to ?\rangle\langle ? \to ? \Rightarrow_p \tau_1 \to \tau_2\rangle]\!]^{\mathrm{G}}_\rho$. The same reasoning yields the other inclusion.

- $\underline{[\mathrm{R}^{\mathrm{G}}_{\mathrm{Collapse}}]}$. $\mathbf{v}\langle \eta \Rightarrow_p ?\rangle\langle ? \Rightarrow_q \eta'\rangle \rightsquigarrow \mathbf{v}$ where $\mathrm{gnd}(\mathbf{v}) \leq \eta'$. We distinguish two cases on η.
  1. $\eta = b$. By Theorem 12.24, we have $[\![\mathbf{v}]\!]^{\mathrm{G}}_\rho \subseteq \langle\!\langle \eta \rangle\!\rangle$. Thus, the only possibility is $\mathbf{v} = c$ such that $b_c \leq \eta$. Since $\mathrm{gnd}(\mathbf{v}) = b_c$, we have $b_c \leq \eta'$. Moreover, we have $[\![\mathbf{v}\langle \eta \Rightarrow_p ?\rangle]\!]^{\mathrm{G}}_\rho = \{c^?\}$, thus we deduce that $[\![\mathbf{v}\langle \eta \Rightarrow_p ?\rangle\langle ? \Rightarrow_q \eta'\rangle]\!]^{\mathrm{G}}_\rho = \{c^!\}$ by Definition 12.18.
  2. $\eta = ? \to ?$. By hypothesis, we have $\Gamma \vdash \mathbf{v} : ? \to ?$, hence $\mathrm{gnd}(\mathbf{v}) = ? \to ?$. Since $\mathrm{gnd}(\mathbf{v}) \leq \eta'$, the only possibility is $\eta' = \eta = \mathrm{gnd}(\mathbf{v}) = ? \to ?$. By Theorem 12.24, we have $[\![\mathbf{v}]\!]^{\mathrm{G}}_\rho \subseteq \langle\!\langle \eta \rangle\!\rangle$. Thus, we deduce by Definition 12.18 that $[\![\mathbf{v}\langle \eta \Rightarrow_p ?\rangle]\!]^{\mathrm{G}}_\rho = \{R^? \mid R^! \in [\![\mathbf{v}]\!]^{\mathrm{G}}_\rho\}$ and then $[\![\mathbf{v}\langle \eta \Rightarrow_p ?\rangle\langle ? \Rightarrow_q \eta'\rangle]\!]^{\mathrm{G}}_\rho = \{R^! \mid R^! \in [\![\mathbf{v}]\!]^{\mathrm{G}}_\rho\} = [\![\mathbf{v}]\!]^{\mathrm{G}}_\rho$.

- $\underline{[\mathrm{R}^{\mathrm{G}}_{\mathrm{Blame}}]}$. $\mathbf{v}\langle \eta \Rightarrow_p ?\rangle\langle ? \Rightarrow_q \eta'\rangle \rightsquigarrow \mathsf{blame}\ q$ where $\mathrm{gnd}(\mathbf{v}) \not\leq \eta'$. We distinguish two cases on η.
  1. $\eta = b$. By Theorem 12.24, we have $[\![\mathbf{v}]\!]^{\mathrm{G}}_\rho \subseteq \langle\!\langle \eta \rangle\!\rangle$. Thus, the only possibility is $\mathbf{v} = c$ such that $b_c \leq \eta$. Since $\mathrm{gnd}(\mathbf{v}) = b_c$, we have $b_c \not\leq \eta'$. We have $[\![\mathbf{v}\langle \eta \Rightarrow_p ?\rangle]\!]^{\mathrm{G}}_\rho = \{c^?\}$ and since $b_c \not\leq \eta'$, we deduce that $[\![\mathbf{v}\langle \eta \Rightarrow_p ?\rangle\langle ? \Rightarrow_q \eta'\rangle]\!]^{\mathrm{G}}_\rho = \{\mathsf{blame}\ q\}$ by Definition 12.18, hence the result.
  2. $\eta = ? \to ?$. By hypothesis, we have $\Gamma \vdash \mathbf{v} : ? \to ?$, hence $\mathrm{gnd}(\mathbf{v}) = ? \to ?$. Since $\mathrm{gnd}(\mathbf{v}) \not\leq \eta'$, the only possibility is $\eta' = b$ for some $b \in \mathscr{B}$. By Theorem 12.24, we have $[\![\mathbf{v}]\!]^{\mathrm{G}}_\rho \subseteq \langle\!\langle \eta \rangle\!\rangle$. Thus, we deduce by Definition 12.18 that $[\![\mathbf{v}\langle \eta \Rightarrow_p ?\rangle]\!]^{\mathrm{G}}_\rho = \{R^? \mid R^! \in [\![\mathbf{v}]\!]^{\mathrm{G}}_\rho\}$ and then $[\![\mathbf{v}\langle \eta \Rightarrow_p ?\rangle\langle ? \Rightarrow_q \eta'\rangle]\!]^{\mathrm{G}}_\rho = \{\mathsf{blame}\ q\}$ by Definition 12.18 since $\eta' = b$.

- $\underline{[\mathrm{R}^{\mathrm{G}}_{\mathrm{Ctx}}]}$. $\mathscr{E}[\mathbf{E}] \rightsquigarrow \mathscr{E}[\mathbf{E}']$ where $\mathbf{E} \rightsquigarrow \mathbf{E}'$. Immediate by induction on $\mathscr{E}$ and Definition 12.19.

- $\underline{[\mathrm{R}^{\mathrm{G}}_{\mathrm{CtxBlame}}]}$. $\mathscr{E}[\mathbf{E}] \rightsquigarrow \mathsf{blame}\ p$ where $\mathbf{E} \rightsquigarrow \mathsf{blame}\ p$. By induction on $\mathscr{E}$.
  - $\mathscr{E} = [\,]$. By IH on $\mathbf{E}$, we immediately have that $[\![\mathbf{E}]\!]^{\mathrm{G}}_\rho = \{\mathsf{blame}\ p\}$.

- $\mathscr{E} = \mathscr{E}'\,\mathbf{E}'$. We have $\mathscr{E}'\,[\mathbf{E}] \rightsquigarrow \mathsf{blame}\,p$ by $[\mathrm{R}^{\mathrm{G}}_{\mathsf{CtxBlame}}]$. By IH, we have $[\![\mathscr{E}'\,[\mathbf{E}]]\!]^{\mathrm{G}}_{\rho} = \{\mathsf{blame}\,p\}$, which yields $[\![\mathscr{E}\,[\mathbf{E}]]\!]^{\mathrm{G}}_{\rho} = \{\mathsf{blame}\,p\}$ by Definition 12.20.
- $\mathscr{E} = \mathbf{v}\,\mathscr{E}'$. We have $\mathscr{E}'\,[\mathbf{E}] \rightsquigarrow \mathsf{blame}\,p$ by $[\mathrm{R}^{\mathrm{G}}_{\mathsf{CtxBlame}}]$. By IH, we have $[\![\mathscr{E}'\,[\mathbf{E}]]\!]^{\mathrm{G}}_{\rho} = \{\mathsf{blame}\,p\}$. Moreover, by Lemma A.54, we have $[\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho} \neq \emptyset$ and by Lemma A.55, $[\![\mathbf{v}]\!]^{\mathrm{G}}_{\rho} \cap \mathbb{B}\mathsf{lame} = \emptyset$, thus $[\![\mathscr{E}\,[\mathbf{E}]]\!]^{\mathrm{G}}_{\rho} = \{\mathsf{blame}\,p\}$ by Definition 12.20.
- $\mathscr{E} = \mathscr{E}'\langle\tau_1 \Rightarrow_q \tau_2\rangle$. We have $\mathscr{E}'\,[\mathbf{E}] \rightsquigarrow \mathsf{blame}\,p$ by $[\mathrm{R}^{\mathrm{G}}_{\mathsf{CtxBlame}}]$. By IH, we have $[\![\mathscr{E}'\,[\mathbf{E}]]\!]^{\mathrm{G}}_{\rho} = \{\mathsf{blame}\,p\}$. By Definition 12.18, we deduce that $[\![\mathscr{E}\,[\mathbf{E}]]\!]^{\mathrm{G}}_{\rho} = \{\mathsf{blame}\,p\}$.

□

Appendix B.

A conservative operational semantics for a set-theoretic cast calculus

In this section of the appendix, we present the full operational semantics of the set-theoretic cast calculus presented in Chapter 5, as defined by Castagna et al. [18]. We give some additional explanation, and provide the full proofs of the properties we presented throughout 5.

B.1. Ground types and values

The type system, expressions and reduction contexts of the cast language are defined as in Chapter 4.

We recall the definition of the *grounding* operator, which we presented in Chapter 5:

Definition B.1 (Grounding and Relative Ground Types)**.** *For all types $\tau, \tau' \in$ GTypes such that $\tau' \preccurlyeq \tau$, we define the* grounding *of τ with respect to τ', noted τ/τ', as follows:*

$$
\begin{array}{rclcrcll}
(\tau_1 \vee \tau_2)/(\tau_1' \vee \tau_2') & = & (\tau_1/\tau_1') \vee (\tau_2/\tau_2') & \qquad & \neg\tau/\neg\tau' & = & \neg(\tau/\tau') & \\
(\tau_1 \vee \tau_2)/? & = & (\tau_1/?) \vee (\tau_2/?) & & \neg\tau/? & = & \neg(\tau/?) & \\
(\tau_1 \to \tau_2)/? & = & ? \to ? & & (\tau_1 \times \tau_2)/? & = & ? \times ? & \\
b/? & = & b & & \mathbb{0}/? & = & \mathbb{0} & \\
\alpha/? & = & \alpha & & \tau/\tau' & = & \tau' & \textit{otherwise}
\end{array}
$$

A type τ is ground with respect to τ' *if and only if $\tau/\tau' = \tau$.*

Note that $\tau' \preccurlyeq \tau$ is a precondition to computing τ/τ'. Therefore to ease the presentation any further reference to τ/τ' will implicitly imply that $\tau' \preccurlyeq \tau$.

In Chapter 4, *ground types* are types ρ such that $\rho/? = \rho$. They are "skeletons" of types whose only information is the top-level constructor. The values of the form $V\langle \rho \overset{p}{\Rightarrow} ? \rangle$ record the essence of the loss of information induced by materialization. We extend this definition to match the new definition of grounding by saying that a type τ is *ground* with respect to τ' if $\tau/\tau' = \tau$. Then, the expressions of the form $V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle$ are values whenever τ is ground with respect to τ'. Intuitively, casts of this form *lose information* about the top-level constructors of a type: an example is the cast $\langle (\mathsf{Int} \to \mathsf{Int}) \wedge (? \to ?) \overset{p}{\Rightarrow} (\mathsf{Int} \to \mathsf{Int}) \wedge ? \rangle$, where we lose information about the $? \to ?$ part, which becomes $?$. Once again, this kind of cast records the essence of this loss.

We have accounted for one kind of cast value, but we also need to update the definition of cast values of the form $V\langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle$ (and similarly for pairs), because function types are not necessarily syntactic arrows anymore (they can be unions and/or intersections thereof). This can be done by considering the opposite case of the previous definition, that is, types such that $\tau/\tau' = \tau'$. Intuitively, a cast $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle$ where $\tau/\tau' = \tau'$ *does not lose or gain* information about

the top-level constructors of a type: it only acts *below* the top constructors. That is, both the origin and target of such a cast have the same syntactic structure "above" constructors, the same "skeleton". For example, $\langle(\mathsf{Int} \to \mathsf{Int}) \wedge (? \to ?) \overset{p}{\Rightarrow} (\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})\rangle$ is such a cast.

Putting everything together, we obtain the following new definition of values:

$$\begin{aligned} V ::= \quad & c \mid \lambda^{\tau \to \tau} x.\, E \mid (V, V) \mid \Lambda \vec{\alpha}.\, E \\ & \mid V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \quad \text{where } \tau_1 \neq \tau_2 \text{ and where } \tau_1 / \tau_2 = \tau_1 \text{ or } \tau_1 / \tau_2 = \tau_2 \text{ or } \tau_2 / \tau_1 = \tau_1 \end{aligned}$$

We say that a value is *unboxed* if it is not of the form $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$. We next need to define a new operator "type" on values (except type abstractions) to resolve particular casts:

Definition B.2 (Value Type Operator)**.** *We define the operator* type *on values of the cast language (except type abstractions) as follows:*

$$\begin{array}{rclcrcl} \mathsf{type}(c) & = & b_c & & \mathsf{type}(\lambda^{\tau_1 \to \tau_2} x.\, E) & = & \tau_1 \to \tau_2 \\ \mathsf{type}(V_1, V_2) & = & \mathsf{type}(V_1) \times \mathsf{type}(V_2) & & \mathsf{type}(V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle) & = & \tau_2 \end{array}$$

Lemma B.3. *For every value V that is not a type abstraction, $\emptyset \vdash V : \mathsf{type}(V)$.*

Proof. By cases on V.

- $V = c$. $\mathsf{type}(V) = b_c$. By typing rule $[\mathrm{T}_{\mathsf{Const}}]$, $\emptyset \vdash V : b_c$.
- $V = \lambda^{\tau_1 \to \tau_2} x.\, E$. $\mathsf{type}(V) = \tau_1 \to \tau_2$. By typing rule $[\mathrm{T}_{\mathsf{Abstr}}]$, $\emptyset \vdash V : \tau_1 \to \tau_2$.
- $V = (V_1, V_2)$. $\mathsf{type}(V) = \mathsf{type}(V_1) \times \mathsf{type}(V_2)$. By induction, for every $i \in \{1, 2\}$, $\emptyset \vdash V_i : \mathsf{type}(V_i)$. Then by typing rule $[\mathrm{T}_{\mathsf{Pair}}]$, $\emptyset \vdash V : \mathsf{type}(V_1) \times \mathsf{type}(V_2)$.
- $V = V'\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$. $\mathsf{type}(V) = \tau_2$. By typing rule $[\mathrm{T}_{\mathsf{Cast}}]$, $\emptyset \vdash V : \tau_2$.

□

B.2. Operational semantics

The semantics of the cast calculus for set-theoretic types is given in Figure B.1.

The rules $[\mathrm{R}_{\mathsf{ExpandL}}]$ and $[\mathrm{R}_{\mathsf{ExpandR}}]$ are the immediate counterparts of the rules of the same name presented in Chapter 4, adapted for the new grounding operator. The other rules of this group use the information provided by the grounding operator to reduce to types that can be easily compared. For example, consider $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle\langle \tau_1' \overset{q}{\Rightarrow} \tau_2' \rangle$. If $\tau_1 / \tau_2 = \tau_1$, then τ_1 contains all the information about type constructors which the cast lost by going into τ_2. Likewise, if $\tau_2' / \tau_1' = \tau_2'$, then all the information about type constructors is in τ_2', so the second cast *adds* constructor information. Therefore, to simplify the expressions, it suffices to compare τ_1 and τ_2', which is what is done in the rules $[\mathrm{R}_{\mathsf{Collapse}}]$ and $[\mathrm{R}_{\mathsf{Blame}}]$ (the set-theoretic counterparts of their namesakes in Chapter 4). The remaining rules for cast reductions follow the same idea, but handle cases that only arise because of set-theoretic types. For example, we can give a constant a dynamic type by subtyping (e.g., $\mathsf{Int} \leq \mathsf{Int} \vee ?$ implies $3 : \mathsf{Int} \vee ?$), and thus we can immediately cast the type of a constant to a more precise type, as in the expression $3\langle \mathsf{Int} \vee ? \overset{p}{\Rightarrow} \mathsf{Int} \vee (? \to ?) \rangle$. The rules $[\mathrm{R}_{\mathsf{UnboxSimpl}}]$ and $[\mathrm{R}_{\mathsf{UnboxBlame}}]$ handle such cases by checking if the cast can be removed.

Cast Reductions.

$$
\begin{array}{rrcll}
[\mathrm{R}_{\mathsf{ExpandL}}] & V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle & \hookrightarrow & V\langle \tau_1 \overset{p}{\Rightarrow} \tau_1/\tau_2\rangle\langle \tau_1/\tau_2 \overset{p}{\Rightarrow} \tau_2\rangle & \text{if } \tau_1/\tau_2 \neq \tau_1,\ \tau_1/\tau_2 \neq \tau_2 \\
[\mathrm{R}_{\mathsf{ExpandR}}] & V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle & \hookrightarrow & V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2/\tau_1\rangle\langle \tau_2/\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle & \text{if } \tau_2/\tau_1 \neq \tau_1,\ \tau_2/\tau_1 \neq \tau_2 \\
[\mathrm{R}_{\mathsf{CastId}}] & V\langle \tau \overset{p}{\Rightarrow} \tau\rangle & \hookrightarrow & V & (*) \\
[\mathrm{R}_{\mathsf{Collapse}}] & V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle & \hookrightarrow & V & \text{if } \tau_1 \leq \tau_2',\ \tau_2'/\tau_1' = \tau_2' \\
 & & & & \text{and } \tau_1/\tau_2 = \tau_1 \text{ or } \tau_2/\tau_1 = \tau_1 \\
[\mathrm{R}_{\mathsf{Blame}}] & V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle & \hookrightarrow & \texttt{blame}\ q & \text{if } \tau_1 \not\leq \tau_2',\ \tau_2'/\tau_1' = \tau_2' \\
 & & & & \text{and } \tau_1/\tau_2 = \tau_1 \text{ or } \tau_2/\tau_1 = \tau_1 \\
[\mathrm{R}_{\mathsf{UpSimpl}}] & V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle & \hookrightarrow & V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle & \text{if } \tau_2 \leq \tau_2',\ \tau_1/\tau_2 = \tau_2,\ \tau_2'/\tau_1' = \tau_2' \\
[\mathrm{R}_{\mathsf{UpBlame}}] & V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle & \hookrightarrow & \texttt{blame}\ q & \text{if } \tau_2 \not\leq \tau_2',\ \tau_1/\tau_2 = \tau_2,\ \tau_2'/\tau_1' = \tau_2' \\
[\mathrm{R}_{\mathsf{UnboxSimpl}}] & V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle & \hookrightarrow & V & \text{if } \mathrm{type}(V) \leq \tau_2,\ \tau_2/\tau_1 = \tau_2,\ V \text{ is unboxed} \\
[\mathrm{R}_{\mathsf{UnboxBlame}}] & V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle & \hookrightarrow & \texttt{blame}\ p & \text{if } \mathrm{type}(V) \not\leq \tau_2,\ \tau_2/\tau_1 = \tau_2,\ V \text{ is unboxed}
\end{array}
$$

$(*)$ to ease the notation and to avoid redundant conditions, the rule $[\mathrm{R}_{\mathsf{CastId}}]$ takes precedence over the following ones. All other casts are therefore considered to be non-identity casts.

Standard Reductions.

$$
\begin{array}{rrcll}
[\mathrm{R}_{\mathsf{CastApp}}] & V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle V' & \hookrightarrow & (V\ V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1\rangle)\langle \tau_2 \overset{p}{\Rightarrow} \tau_2'\rangle & \text{if } \tau'/\tau = \tau \text{ or } \tau/\tau' = \tau' \\
 & & & \text{where } \langle \tau \overset{p}{\Rightarrow} \tau'\rangle \circ \mathrm{type}(V') = \langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2'\rangle & \\
[\mathrm{R}_{\mathsf{CastProj}}] & \pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) & \hookrightarrow & (\pi_i\ V)\langle \tau_i \overset{p}{\Rightarrow} \tau_i'\rangle & \text{if } \tau'/\tau = \tau \text{ or } \tau/\tau' = \tau' \\
 & & & \text{where } \langle \tau_i \overset{p}{\Rightarrow} \tau_i'\rangle = \pi_i\ (\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) & \\
[\mathrm{R}_{\mathsf{FailApp}}] & V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle V' & \hookrightarrow & \texttt{blame p} & \text{if } \langle \tau \overset{p}{\Rightarrow} \tau'\rangle \circ \mathrm{type}(V') \text{ undef.} \\
[\mathrm{R}_{\mathsf{FailProj}}] & \pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) & \hookrightarrow & \texttt{blame p} & \text{if } \pi_i\ (\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) \text{ undef.} \\
[\mathrm{R}_{\mathsf{SimplApp}}] & V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle V' & \hookrightarrow & V\ V' & \text{if } \tau/\tau' = \tau \\
[\mathrm{R}_{\mathsf{SimplProj}}] & \pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) & \hookrightarrow & \pi_i\ V & \text{if } \tau/\tau' = \tau \\
[\mathrm{R}_{\mathsf{App}}] & (\lambda^{\tau_1 \to \tau_2} x.\, E)V & \hookrightarrow & E[V/x] & \\
[\mathrm{R}_{\mathsf{Proj}}] & \pi_i\ (V_1, V_2) & \hookrightarrow & V_i & \\
[\mathrm{R}_{\mathsf{TypeApp}}] & (\Lambda \vec{\alpha}.\, E)\ [\vec{t}] & \hookrightarrow & E[\vec{t}/\vec{\alpha}] & \\
[\mathrm{R}_{\mathsf{Let}}] & \mathsf{let}\ x = V\ \mathsf{in}\ E & \hookrightarrow & E[V/x] & \\
[\mathrm{R}_{\mathsf{Context}}] & \mathscr{E}[E] & \hookrightarrow & \mathscr{E}[E'] & \text{if } E \hookrightarrow E' \\
[\mathrm{R}_{\mathsf{CtxBlame}}] & \mathscr{E}[E] & \hookrightarrow & \texttt{blame}\ p & \text{if } E \hookrightarrow \texttt{blame}\ p
\end{array}
$$

FIGURE B.1. Semantics of the cast calculus

The intuition is that the dynamic part of such casts is useless since it has been introduced by subtyping.

The rules for applications and projections also need to be updated because function and product types can now be unions and intersections of arrows or products. For applications, we define a new operator, written $\circ$, which, given a function cast and the type of the argument, computes an approximation of the cast such that both its origin and target types are arrows, so that the usual rule for cast applications defined in Chapter 4 can be applied. More formally, the operation $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \tau_v$ computes a cast $\langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle$ such that $\tau_v \leq \tau_1'$, $\tau_2' = \min\{\tau \mid \tau' \leq \tau_v \to \tau\}$, $\tau \leq \tau_1 \to \tau_2$, and such that the precision relation between the two parts of the cast is preserved. This ensures that the resulting approximation is still well-typed. The definition of this operator is quite involved, so we dedicate the next two sections to its definition and to the proofs of its properties. The most important point of this definition is that it requires both types of the cast to be syntactically identical above their constructors, which explains the presence of the grounding condition in $[\mathrm{R}_{\mathsf{CastApp}}]$. Moreover, this operator can also be undefined in some cases, such as if the origin type of the cast is not an arrow type or if the second type is empty (e.g. $\langle (? \to ?) \wedge \neg(\mathsf{Int} \to \mathsf{Int}) \overset{p}{\Rightarrow} (\mathsf{Int} \to \mathsf{Int}) \wedge \neg(\mathsf{Int} \to \mathsf{Int}) \rangle$). Such ill-formed casts are handled by $[\mathrm{R}_{\mathsf{FailApp}}]$. We apply the same idea to projections and define an operator, written π_i, that computes an approximation of the first or second component of a cast between two product types. This yields the rules $[\mathrm{R}_{\mathsf{CastProj}}]$ and $[\mathrm{R}_{\mathsf{FailProj}}]$. The two remaining rules, $[\mathrm{R}_{\mathsf{SimplApp}}]$ and $[\mathrm{R}_{\mathsf{SimplProj}}]$, handle cases that only appear due to the presence of set-theoretic types. For instance, it is now possible to apply (or project) a value that has a dynamic type: $V\langle (\mathsf{Int} \to \mathsf{Int}) \wedge (? \to ?) \overset{p}{\Rightarrow} (\mathsf{Int} \to \mathsf{Int}) \wedge ? \rangle V'$. Here, by subtyping, the function has both type $\mathsf{Int} \to \mathsf{Int}$ and $?$, so it can be applied but it is also dynamic. We show that such casts are unnecessary and can be harmlessly removed; the rules $[\mathrm{R}_{\mathsf{SimplApp}}]$ and $[\mathrm{R}_{\mathsf{SimplProj}}]$ do just that.

B.3. Normal forms and decompositions for type frames

In the following, we use the metavariable a to range over the set $\mathscr{B} \cup \mathscr{A}_{prod} \cup \mathscr{A}_{fun} \cup \mathscr{V}^{\alpha}$.

We also introduce the following two results from Frisch et al. [27] which we will not prove here.

Lemma B.4. *Let P, N be two finite subsets of $\mathscr{A}_{prod}$. Then:*

$$\bigwedge_{T_1 \times T_2 \in P} T_1 \times T_2 \leq \bigvee_{T_1 \times T_2 \in N} T_1 \times T_2 \iff \forall N' \subseteq N. \left(\bigwedge_{T_1 \times T_2 \in P} T_1 \leq \bigvee_{T_1 \times T_2 \in N'} T_1 \right) \vee \left(\bigwedge_{T_1 \times T_2 \in P} T_2 \leq \bigvee_{T_1 \times T_2 \in N \setminus N'} T_2 \right)$$

(with the convention $\bigwedge_{T \in \emptyset} T = \mathbb{1} \times \mathbb{1}$).

Lemma B.5. *Let P, N be two finite subsets of $\mathscr{A}_{fun}$. Then:*

$$\bigwedge_{T_1 \to T_2 \in P} T_1 \to T_2 \leq \bigvee_{T_1 \to T_2 \in N} T_1 \to T_2 \iff \exists (\overline{T}_1 \to \overline{T}_2) \in N. \left(\overline{T}_1 \leq \bigvee_{T_1 \to T_2 \in P} T_1 \right) \wedge \left(\forall P' \subsetneq P. \left(\overline{T}_1 \leq \bigvee_{T_1 \to T_2 \in P'} T_1 \right) \vee \left(\bigwedge_{T_1 \to T_2 \in P \setminus P'} T_2 \leq \overline{T}_2 \right) \right)$$

(with the convention $\bigwedge_{T \in \emptyset} T = \mathbb{0} \to \mathbb{1}$).

We now define *uniform disjunctive normal forms* for type frames, and study their properties.

Definition B.6 (Uniform normal form)**.** *A* uniform (disjunctive) normal form *(UDNF) is a type frame T of the form*

$$\bigvee_{i\in I}\left(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a\right)$$

such that, for all $i \in I$, one of the following three condition holds:

- $P_i \cap \mathscr{B} \neq \emptyset$ *and* $(P_i \cup N_i) \cap (\mathscr{A}_{prod} \cup \mathscr{A}_{fun}) = \emptyset$;
- $P_i \cap \mathscr{A}_{prod} \neq \emptyset$ *and* $(P_i \cup N_i) \cap (\mathscr{B} \cup \mathscr{A}_{fun}) = \emptyset$;
- $P_i \cap \mathscr{A}_{fun} \neq \emptyset$ *and* $(P_i \cup N_i) \cap (\mathscr{B} \cup \mathscr{A}_{prod}) = \emptyset$;

We define here a function $\mathsf{UDNF}(T)$ which, given a type frame T, produces a uniform normal form that is equivalent to T.

We first define two mutually recursive functions $\mathscr{N}$ and $\mathscr{N}'$ on type frames. These are inductive definitions as no recursive uses of the functions occur below type constructors.

$$\begin{aligned}
\mathscr{N}(a) &= a\\
\mathscr{N}(T_1 \vee T_2) &= \mathscr{N}(T_1) \vee \mathscr{N}(T_2)\\
\mathscr{N}(\neg T) &= \mathscr{N}'(T)\\
\mathscr{N}(\mathbb{0}) &= \mathbb{0}\\
\mathscr{N}'(a) &= \neg a\\
\mathscr{N}'(T_1 \vee T_2) &= \bigvee_{i\in I, j\in J}\Big(\bigwedge_{a\in P_i\cup P_j} a \wedge \bigwedge_{a\in N_i\cup N_j} \neg a\Big)\\
&\qquad \text{where } \mathscr{N}'(T_1) = \bigvee_{i\in I}\Big(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a\Big) \text{ and } \mathscr{N}'(T_2) = \bigvee_{j\in J}\Big(\bigwedge_{a\in P_j} a \wedge \bigwedge_{a\in N_j} \neg a\Big)\\
\mathscr{N}'(\neg T) &= \mathscr{N}(T)\\
\mathscr{N}'(\mathbb{0}) &= \mathbb{1}
\end{aligned}$$

In the definition above, we see $\mathbb{0}$ as the empty union $\bigvee_{i\in\emptyset} T_i$ and $\mathbb{1}$ as the singleton union of the empty intersection $\bigvee_{i\in\{i_0\}} \bigwedge_{a\in\emptyset} a$.

The first step in the computation of $\mathsf{UDNF}(T)$ is to compute $\mathscr{N}(T)$. Then, assuming

$$\mathscr{N}(T) = \bigvee_{i\in I}\Big(\underbrace{\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a}_{\mathscr{I}_i}\Big)$$

we define

$$\mathsf{UDNF}(T) \stackrel{\text{def}}{=} \bigvee_{i\in I} \mathscr{I}_i^{\text{basic}} \vee \bigvee_{i\in I} \mathscr{I}_i^{\text{prod}} \vee \bigvee_{i\in I} \mathscr{I}_i^{\text{fun}}$$

where

$$\mathscr{I}_i^{\mathsf{basic}} \stackrel{\text{def}}{=} \mathbb{1}_{\mathscr{B}} \wedge \bigwedge_{a \in P_i \cap (\mathscr{B} \cup \mathscr{V}^\alpha)} a \wedge \bigwedge_{a \in N_i \cap (\mathscr{B} \cup \mathscr{V}^\alpha)} \neg a$$
$$\mathscr{I}_i^{\mathsf{prod}} \stackrel{\text{def}}{=} (\mathbb{1} \times \mathbb{1}) \wedge \bigwedge_{a \in P_i \cap (\mathscr{A}_{prod} \cup \mathscr{V}^\alpha)} a \wedge \bigwedge_{a \in N_i \cap (\mathscr{A}_{prod} \cup \mathscr{V}^\alpha)} \neg a$$
$$\mathscr{I}_i^{\mathsf{fun}} \stackrel{\text{def}}{=} (\mathbb{0} \to \mathbb{1}) \wedge \bigwedge_{a \in P_i \cap (\mathscr{A}_{fun} \cup \mathscr{V}^\alpha)} a \wedge \bigwedge_{a \in N_i \cap (\mathscr{A}_{fun} \cup \mathscr{V}^\alpha)} \neg a$$

Lemma B.7. *Given any type frame T, $\mathsf{UDNF}(T)$ is a uniform normal form and $\mathsf{UDNF}(T) \simeq T$. Moreover, if T is strongly polarized, then $\mathsf{UDNF}(T)$ is strongly polarized.*

Proof. Let T be a type frame. We can check on the definition of $\mathscr{N}$ and $\mathscr{N}'$ that $\mathscr{N}(T)$ is a union of intersection of atoms, assuming that we see $\mathbb{0}$ and $\mathbb{1}$ as described above and that atoms are interpreted as singleton unions of singleton intersections. We can check by induction on T that

$$[\![T]\!] = [\![\mathscr{N}(T)]\!] = [\![\neg\mathscr{N}'(T)]\!] .$$

Moreover, when T is strongly polarized, $\mathscr{N}(T)$ is strongly polarized too, because every atom of T appears in $\mathscr{N}(T)$ with the same polarity.

We now consider $\mathsf{UDNF}(T)$. It is trivial to check that it is always in disjunctive normal form. Preservation of strong polarization is also ensured by the fact that we are maintaining the polarity every atom had in $\mathscr{N}(T)$. The conditions that every intersection contains at least one positive atom and that the intersections are uniform are ensured by construction.

It remains to check $\mathsf{UDNF}(T) \simeq \mathscr{N}(T)$. Note that $\mathbb{1} \simeq \mathbb{1}_{\mathscr{B}} \vee (\mathbb{1} \times \mathbb{1}) \vee (\mathbb{0} \to \mathbb{1})$. We have the following equivalences.

$$\begin{aligned}
\mathscr{I}_i &\simeq \mathbb{1} \wedge \mathscr{I}_i \\
&\simeq \big(\mathbb{1}_{\mathscr{B}} \vee (\mathbb{1} \times \mathbb{1}) \vee (\mathbb{0} \to \mathbb{1})\big) \wedge \mathscr{I}_i \\
&\simeq \big(\mathbb{1}_{\mathscr{B}} \wedge \mathscr{I}_i\big) \vee \big((\mathbb{1} \times \mathbb{1}) \wedge \mathscr{I}_i\big) \vee \big((\mathbb{0} \to \mathbb{1}) \wedge \mathscr{I}_i\big)
\end{aligned}$$

We show the following three results.

$$\begin{aligned}
\mathbb{1}_{\mathscr{B}} \wedge \mathscr{I}_i &\simeq \mathscr{I}_i^{\mathsf{basic}} \\
(\mathbb{1} \times \mathbb{1}) \wedge \mathscr{I}_i &\simeq \mathscr{I}_i^{\mathsf{prod}} \\
(\mathbb{0} \to \mathbb{1}) \wedge \mathscr{I}_i &\simeq \mathscr{I}_i^{\mathsf{fun}}
\end{aligned}$$

For the first implication, we have

$$\begin{aligned}
\mathbb{1}_{\mathscr{B}} \wedge \mathscr{I}_i &= \mathbb{1}_{\mathscr{B}} \wedge \textstyle\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \\
&\simeq \mathbb{1}_{\mathscr{B}} \wedge \textstyle\bigwedge_{a \in P_i \cap (\mathscr{B} \cup \mathscr{V}^\alpha)} a \wedge \bigwedge_{a \in N_i \cap (\mathscr{B} \cup \mathscr{V}^\alpha)} \neg a \wedge \\
&\qquad \textstyle\bigwedge_{a \in P_i \cap (\mathscr{A}_{prod} \cup \mathscr{A}_{fun})} a \wedge \bigwedge_{a \in N_i \cap (\mathscr{A}_{prod} \cup \mathscr{A}_{fun})} \neg a \\
&\simeq \mathbb{1}_{\mathscr{B}} \wedge \textstyle\bigwedge_{a \in P_i \cap (\mathscr{B} \cup \mathscr{V}^\alpha)} a \wedge \bigwedge_{a \in N_i \cap (\mathscr{B} \cup \mathscr{V}^\alpha)} \neg a \wedge \bigwedge_{a \in N_i \cap (\mathscr{A}_{prod} \cup \mathscr{A}_{fun})} \neg a
\end{aligned}$$

(because $P_i \cap (\mathscr{A}_{prod} \cup \mathscr{A}_{fun}) = \emptyset$: otherwise the intersection would be empty because, when $a \in \mathscr{A}_{prod} \cup \mathscr{A}_{fun}$, $[\![a]\!] \cap [\![\mathbb{1}_{\mathscr{B}}]\!] = \emptyset$)

$$\simeq \mathbb{1}_{\mathscr{B}} \wedge \bigwedge_{a \in P_i \cap (\mathscr{B} \cup \mathscr{V}^\alpha)} a \wedge \bigwedge_{a \in N_i \cap (\mathscr{B} \cup \mathscr{V}^\alpha)} \neg a$$

(because, when $a \in \mathscr{A}_{prod} \cup \mathscr{A}_{fun}$, since $[\![a]\!] \cap [\![\mathbb{1}_{\mathscr{B}}]\!] = \emptyset$, we have $[\![\mathbb{1}_{\mathscr{B}}]\!] \subseteq [\![\neg a]\!]$). The other two implications are shown identically.

To conclude, we observe the following equivalence.

$$\begin{aligned}\bigvee_{i \in I} \mathscr{I}_i &\simeq \bigvee_{i \in I} \Big((\mathbb{1}_{\mathscr{B}} \wedge \mathscr{I}_i) \vee ((\mathbb{1} \times \mathbb{1}) \wedge \mathscr{I}_i) \vee ((\mathbb{0} \to \mathbb{1}) \wedge \mathscr{I}_i) \Big) \\ &\simeq \bigvee_{i \in I} (\mathbb{1}_{\mathscr{B}} \wedge \mathscr{I}_i) \vee \bigvee_{i \in I} ((\mathbb{1} \times \mathbb{1}) \wedge \mathscr{I}_i) \vee \bigvee_{i \in I} ((\mathbb{0} \to \mathbb{1}) \wedge \mathscr{I}_i) \\ &\simeq \bigvee_{i \in I} \mathscr{I}_i^{\mathsf{basic}} \vee \bigvee_{i \in I} \mathscr{I}_i^{\mathsf{prod}} \vee \bigvee_{i \in I} \mathscr{I}_i^{\mathsf{fun}}\end{aligned}$$

□

Definition B.8 (Product decomposition and projections)**.** *Given a type frame $T \leq \mathbb{1} \times \mathbb{1}$, we define its decomposition $\pi(T)$ as*

$$\pi(T) \stackrel{\text{def}}{=} \bigcup_{i \in I, \mathscr{I}_i \not\leq \mathbb{0}} \Big\{ \Big(\underbrace{\bigwedge_{T_1 \times T_2 \in \overline{P}_i} T_1 \wedge \bigwedge_{T_1 \times T_2 \in N'} \neg T_1}_{\overline{T}_1}, \underbrace{\bigwedge_{T_1 \times T_2 \in \overline{P}_i} T_2 \wedge \bigwedge_{T_1 \times T_2 \in \overline{N}_i \setminus N'} \neg T_2}_{\overline{T}_2} \Big) \Big| N' \subseteq \overline{N}_i, \overline{T}_1 \not\leq \mathbb{0}, \overline{T}_2 \not\leq \mathbb{0} \Big\}$$

and its i-th projection $\pi_i(T)$ as

$$\pi_i(T) \stackrel{\text{def}}{=} \bigvee_{(T_1, T_2) \in \pi(T)} T_i$$

where

$$\mathsf{UDNF}(T) = \bigvee_{i \in I} \underbrace{\Big(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \Big)}_{\mathscr{I}_i}$$

and where $\overline{P}_i = P_i \cap \mathscr{A}_{prod}$ and $\overline{N}_i = N_i \cap \mathscr{A}_{prod}$.

Lemma B.9. *Let P, N be two finite subsets of $\mathscr{B} \cup \mathscr{A}_{prod} \cup \mathscr{A}_{fun}$ and P', N' be two finite subsets of $\mathscr{V}^\alpha$. If $P' \cap N' = \emptyset$, then*

$$\bigwedge_{a \in P} a \wedge \bigwedge_{a \in N} \neg a \wedge \bigwedge_{a \in P'} a \wedge \bigwedge_{a \in N'} \neg a \leq \mathbb{0} \iff \bigwedge_{a \in P} a \wedge \bigwedge_{a \in N} \neg a \leq \mathbb{0}$$

Proof. The implication $\Leftarrow$ is trivial. We prove the other direction by contrapositive. Assume that the subtyping relation on the right does not hold. Then, we have

$$d \in \Big[\!\Big[\bigwedge_{a \in P} a \wedge \bigwedge_{a \in N} \neg a \Big]\!\Big] .$$

Therefore $d \in [\![a]\!]$ holds for all $a \in P$ and $d \in \mathscr{D} \setminus [\![N]\!]$ holds for all $a \in N$.

Note that every $a \in P \cup N$ is of the forms b, $T_1 \times T_2$, or $T_1 \to T_2$. For such types, if $d \in [\![a]\!]$, then every d' that differs from d only for its outermost set of tags satisfies $d' \in [\![a]\!]$.

We consider the domain element $\overline{d}$ which is d changed to have $\mathsf{tags}(\overline{d}) = P'$. By construction, it is in the interpretation of all variables in P' and in none of the interpretations of the variables in N'. Hence, we have

$$\overline{d} \in [\![\textstyle\bigwedge_{a\in P} a \wedge \bigwedge_{a\in N} \neg a \wedge \bigwedge_{a\in P'} a \wedge \bigwedge_{a\in N'} \neg a]\!] \,. \qquad \square$$

Lemma B.10. *Let T be a type frame such that $T \leq \mathbb{1} \times \mathbb{1}$. Then, for all type frames T_1 and T_2,*

$$T \leq T_1 \times T_2 \iff \bigvee_{(T_1',T_2')\in\pi(T)} T_1' \times T_2' \leq T_1 \times T_2 \,.$$

Proof. Given T, we have

$$\mathsf{UDNF}(T) = \textstyle\bigvee_{i\in I} \underbrace{\left(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a\right)}_{\mathscr{I}_i}$$

and, by Lemma B.7, $T \simeq \mathsf{UDNF}(T)$. Then, since $T \leq \mathbb{1} \times \mathbb{1}$, we have

$$\forall i \in I.\ \mathscr{I}_i = \textstyle\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a \leq \mathbb{1} \times \mathbb{1} \,.$$

Consider a i such that $\mathscr{I}_i \not\leq \mathbb{0}$. Since each P_i must contain an atom, we have that P_i contains a type frame of the form $T_1 \times T_2$. Hence, since intersections are uniform, $P_i \cup N_i \subseteq \mathscr{A}_{prod} \cup \mathscr{V}^\alpha$. Moreover, $\mathscr{V}^\alpha \cap P_i \cap N_i = \emptyset$, otherwise $\mathscr{I}_i$ would be empty.

We have

$$\begin{aligned}
T \leq T_1 \times T_2 &\iff \textstyle\bigvee_{i\in I} \left(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a\right) \leq T_1 \times T_2 \\
&\iff \textstyle\bigvee_{i\in I, \mathscr{I}_i\not\leq\mathbb{0}} \left(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a\right) \leq T_1 \times T_2 \\
&\iff \forall i \in I, \mathscr{I}_i \not\leq \mathbb{0}.\ \textstyle\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a \wedge \neg(T_1 \times T_2) \leq \mathbb{0} \\
&\iff \forall i \in I, \mathscr{I}_i \not\leq \mathbb{0}.\ \textstyle\bigwedge_{a\in P_i\cap\mathscr{A}_{prod}} a \wedge \bigwedge_{a\in N_i\cap\mathscr{A}_{prod}} \neg a \wedge \neg(T_1 \times T_2) \leq \mathbb{0}
\end{aligned}$$

(by Lemma B.9, since $P_i \cap \mathscr{V}^\alpha$ and $N_i \cap \mathscr{V}^\alpha$ are disjoint; let $\overline{P}_i = P_i \cap \mathscr{A}_{prod}$ and $\overline{N}_i = N_i \cap \mathscr{A}_{prod}$)

$$\begin{aligned}
&\iff \forall i \in I, \mathscr{I}_i \not\leq \mathbb{0}.\ \textstyle\bigwedge_{T_1'\times T_2'\in\overline{P}_i} T_1' \times T_2' \wedge \bigwedge_{T_1'\times T_2'\in\overline{N}_i} \neg(T_1' \times T_2') \wedge \neg(T_1 \times T_2) \leq \mathbb{0} \\
&\iff \forall i \in I, \mathscr{I}_i \not\leq \mathbb{0}.\ \textstyle\bigwedge_{T_1'\times T_2'\in\overline{P}_i} T_1' \times T_2' \leq \left(\bigvee_{T_1'\times T_2'\in\overline{N}_i} T_1' \times T_2'\right) \vee (T_1 \times T_2)
\end{aligned}$$

We now apply Lemma B.4 to the above to derive the following equivalence.

$$\begin{aligned}
&T \leq T_1 \times T_2 \iff \forall i \in I, \mathscr{I}_i \not\leq \mathbb{0}. \\
&\left(\forall N' \subseteq \overline{N}_i.\ \left(\textstyle\bigwedge_{T_1'\times T_2'\in\overline{P}_i} T_1' \leq \bigvee_{T_1'\times T_2'\in N'} T_1'\right) \vee \left(\textstyle\bigwedge_{T_1'\times T_2'\in\overline{P}_i} T_2' \leq \left(\bigvee_{T_1'\times T_2'\in\overline{N}_i\setminus N'} T_2'\right) \vee T_2\right)\right) \wedge \\
&\left(\forall N' \subseteq \overline{N}_i.\ \left(\textstyle\bigwedge_{T_1'\times T_2'\in\overline{P}_i} T_1' \leq \left(\bigvee_{T_1'\times T_2'\in N'} T_1'\right) \vee T_1\right) \vee \left(\textstyle\bigwedge_{T_1'\times T_2'\in\overline{P}_i} T_2' \leq \bigvee_{T_1'\times T_2'\in\overline{N}_i\setminus N'} T_2'\right)\right)
\end{aligned}$$

We have split the quantification over all N' into two: we consider the type $T_1 \times T_2$ in the

second case and not in the first.

We also have

$$\bigvee_{(T_1'',T_2'')\in\pi(T)} T_1'' \times T_2'' \leq T_1 \times T_2 \iff \forall (T_1'',T_2'') \in \pi(T).\, T_1'' \times T_2'' \leq T_1 \times T_2$$
$$\iff \forall (T_1'',T_2'') \in \pi(T).\, (T_1'' \leq T_1) \wedge (T_2'' \leq T_2)$$

with

$$\pi(T) = \bigcup_{i\in I, \mathscr{S}_i \not\leq \mathbb{0}} \Big\{ \Big(\underbrace{\bigwedge_{T_1'\times T_2'\in\overline{P}_i} T_1' \wedge \bigwedge_{T_1'\times T_2'\in N'} \neg T_1'}_{\overline{T}_1}, \underbrace{\bigwedge_{T_1'\times T_2'\in\overline{P}_i} T_2' \wedge \bigwedge_{T_1'\times T_2'\in\overline{N}_i\setminus N'} \neg T_2'}_{\overline{T}_2} \Big) \Big| N' \subseteq \overline{N}_i, \overline{T}_1 \not\leq \mathbb{0}, \overline{T}_2 \not\leq \mathbb{0} \Big\}$$

To show

$$T \leq T_1 \times T_2 \iff \bigvee_{(T_1'',T_2'')\in\pi(T)} T_1'' \times T_2'' \leq T_1 \times T_2$$

we first show the implication from left to right. Let $(T_1'', T_2'') \in \pi(T)$. Since $\pi(T)$ is a union, (T_1'', T_2'') must be in at least one set in the union; we assume it is the set indexed by $i_0 \in I$. We must show $T_1'' \leq T_1$ and $T_2'' \leq T_2$.

By definition, (T_1'', T_2'') is a pair corresponding to some $N' \subseteq \overline{N}_{i_0}$. In that case, we must check

$$\bigwedge_{T_1'\times T_2'\in\overline{P}_{i_0}} T_1' \wedge \bigwedge_{T_1'\times T_2'\in N'} \neg T_1' \leq T_1 \qquad \bigwedge_{T_1'\times T_2'\in\overline{P}_{i_0}} T_2' \wedge \bigwedge_{T_1'\times T_2'\in\overline{N}_{i_0}\setminus N'} \neg T_2' \leq T_2\,,$$

which is

$$\bigwedge_{T_1'\times T_2'\in\overline{P}_{i_0}} T_1' \leq \Big(\bigvee_{T_1'\times T_2'\in N'} T_1'\Big) \vee T_1 \qquad \bigwedge_{T_1'\times T_2'\in\overline{P}_{i_0}} T_2' \leq \Big(\bigvee_{T_1'\times T_2'\in\overline{N}_{i_0}\setminus N'} T_2'\Big) \vee T_2\,.$$

We know from the condition on the set that

$$\bigwedge_{T_1'\times T_2'\in\overline{P}_{i_0}} T_1' \not\leq \bigvee_{T_1'\times T_2'\in N'} T_1' \qquad \bigwedge_{T_1'\times T_2'\in\overline{P}_{i_0}} T_2' \not\leq \bigvee_{T_1'\times T_2'\in\overline{N}_{i_0}\setminus N'} T_2'\,.$$

We can check the two relations in the decomposition above obtained by Lemma B.4 using the relations that do not hold to eliminate one case in the disjunction.

To check the other direction of the implication, we assume that, for all $(T_1'', T_2'') \in \pi(T)$, we have $(T_1'' \leq T_1) \wedge (T_2'' \leq T_2)$. We prove that the conditions obtained from the decomposition of subtyping hold. Consider an arbitrary $i \in I$. As $\pi(T)$ is a union, we will consider the set indexed by the same i. For the first condition, we take an arbitrary N'. If there is a pair corresponding to the same N' in $\pi(T)$, then we show the second disjunct. If there is no such pair, it is because $\overline{T}_1$ or $\overline{T}_2$ is empty, which also allows us to conclude. For the second condition, we consider an arbitrary N' and proceed analogously. □

Lemma B.11. *Let T be a type frame such that $T \leq \mathbb{1} \times \mathbb{1}$. Then $T \leq \pi_1(T) \times \pi_2(T)$. Moreover, if $T \leq T_1 \times T_2$, then $\pi_1(T) \leq T_1$ and $\pi_2(T) \leq T_2$.*

Proof. We have $\bigvee_{(T_1',T_2')\in\pi(T)} T_1' \times T_2' \leq (\bigvee_{(T_1',T_2')\in\pi(T)} T_1') \times (\bigvee_{(T_1',T_2')\in\pi(T)} T_2') = \pi_1(T) \times \pi_2(T)$. Hence, by Lemma B.11, we have $T \leq \pi_1(T) \times \pi_2(T)$.

If $T \leq T_1 \times T_2$, again by Lemma B.11, we have $\bigvee_{(T_1',T_2')\in\pi(T)} T_1' \times T_2' \leq T_1 \times T_2$. Hence, for

every $(T_1', T_2') \in \pi(T)$, we have $T_1' \times T_2' \leq T_1 \times T_2$ and, by the definition of subtyping, $T_1' \leq T_1$ and $T_2' \leq T_2$, since T_1' and T_2' are not empty. As a result, we also have $\bigvee_{(T_1', T_2') \in \pi(T)} T_1' \leq T_1$ and $\bigvee_{(T_1', T_2') \in \pi(T)} T_2' \leq T_2$, that is, $\pi_1(T) \leq T_1$ and $\pi_2(T) \leq T_2$. □

Lemma B.12. *Let T be a type frame such that $T \leq \mathbb{1} \times \mathbb{1}$. If T is strongly polarized, then $\pi_1(T)$ and $\pi_2(T)$ are strongly polarized.*

Proof. If T is strongly polarized, then, by Lemma B.7, $\mathsf{UDNF}(T)$ is strongly polarized too. We can check on the definition of $\pi(T)$ that, in every $(T_1, T_2) \in \pi(T)$, subterms of $\mathsf{UDNF}(T)$ appear in T_i with the same polarity as in $\mathsf{UDNF}(T)$. Then, $\pi_i(T)$ also preserves polarity. □

Definition B.13 (Function domain and decomposition)**.** *Given a type frame $T \leq \mathbb{0} \to \mathbb{1}$, we define its* domain $\mathsf{dom}(T)$ *as*

$$\mathsf{dom}(T) \stackrel{\text{def}}{=} \bigwedge_{i \in I, \mathscr{I}_i \not\leq \mathbb{0}} \bigvee_{T_1 \to T_2 \in \overline{P}_i} T_1$$

and its decomposition $\phi(T)$ *as*

$$\phi(T) \stackrel{\text{def}}{=} \bigcup_{i \in I, \mathscr{I}_i \not\leq \mathbb{0}} \left\{ \left(\bigvee_{T_1 \to T_2 \in P'} T_1, \bigwedge_{T_1 \to T_2 \in \overline{P}_i \setminus P'} T_2 \right) \middle| P' \subsetneq \overline{P}_i \right\}$$

where

$$\mathsf{UDNF}(T) = \bigvee_{i \in I} \underbrace{\left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)}_{\mathscr{I}_i}$$

and where $\overline{P}_i = P_i \cap \mathscr{A}_{fun}$ and $\overline{N}_i = N_i \cap \mathscr{A}_{fun}$.

Definition B.14 (Application result type)**.** *Given two type frames T and T' such that $T \leq \mathbb{0} \to \mathbb{1}$ and $T' \leq \mathsf{dom}(T)$, we define the* application result type $T \circ T'$ *as*

$$T \circ T' \stackrel{\text{def}}{=} \bigvee_{\substack{(T_1, T_2) \in \phi(T) \\ T' \not\leq T_1}} T_2 \,.$$

Lemma B.15. *Let T be a type frame such that $T \leq \mathbb{0} \to \mathbb{1}$. Then, for all type frames T' and T'',*

$$T \leq T' \to T'' \iff \begin{cases} \forall (T_1', T_2') \in \phi(T).\ (T' \leq T_1') \vee (T_2' \leq T'') \\ \wedge \\ T' \leq \mathsf{dom}(T) \end{cases}$$

Proof. Given T, we have

$$\mathsf{UDNF}(T) = \textstyle\bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)$$

and, by Lemma B.7, $T \simeq \mathsf{UDNF}(T)$. Then, since $T \leq \mathbb{0} \to \mathbb{1}$, we have

$$\forall i \in I.\ \textstyle\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a \leq \mathbb{0} \to \mathbb{1}\,.$$

For every non-empty intersection $\mathscr{I}_i$, since each P_i must contain an atom, we have that P_i contains a type frame of the form $T_1 \to T_2$. Hence, since intersections are uniform, $P_i \cup N_i \subseteq \mathscr{A}_{fun} \cup \mathscr{V}^\alpha$. Moreover, $\mathscr{V}^\alpha \cap P_i \cap N_i = \emptyset$ otherwise $\mathscr{I}_i$ would be empty.

We have

$$\begin{aligned}
T \leq T' \to T'' &\iff \textstyle\bigvee_{i\in I} \left(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a\right) \leq T' \to T'' \\
&\iff \textstyle\bigvee_{i\in I, \mathscr{I}_i \not\leq \mathbb{0}} \left(\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a\right) \leq T' \to T'' \\
&\iff \forall i \in I, \mathscr{I}_i \not\leq \mathbb{0}.\ \textstyle\bigwedge_{a\in P_i} a \wedge \bigwedge_{a\in N_i} \neg a \wedge \neg(T' \to T'') \leq \mathbb{0} \\
&\iff \forall i \in I, \mathscr{I}_i \not\leq \mathbb{0}.\ \textstyle\bigwedge_{a\in P_i\cap\mathscr{A}_{fun}} a \wedge \bigwedge_{a\in N_i\cap\mathscr{A}_{fun}} \neg a \wedge \neg(T' \to T'') \leq \mathbb{0}
\end{aligned}$$

(by Lemma B.9, since $P_i \cap \mathscr{V}^\alpha$ and $N_i \cap \mathscr{V}^\alpha$ are disjoint; let $\overline{P}_i = P_i \cap \mathscr{A}_{fun}$ and $\overline{N}_i = N_i \cap \mathscr{A}_{fun}$)

$$\begin{aligned}
&\iff \forall i \in I, \mathscr{I}_i \not\leq \mathbb{0}.\ \textstyle\bigwedge_{T_1\to T_2\in\overline{P}_i} T_1 \to T_2 \wedge \bigwedge_{T_1\to T_2\in\overline{N}_i} \neg(T_1 \to T_2) \wedge \neg(T' \to T'') \leq \mathbb{0} \\
&\iff \forall i \in I, \mathscr{I}_i \not\leq \mathbb{0}.\ \textstyle\bigwedge_{T_1\to T_2\in\overline{P}_i} T_1 \to T_2 \leq \left(\bigvee_{T_1\to T_2\in\overline{N}_i} T_1 \to T_2\right) \vee (T' \to T'')
\end{aligned}$$

Now consider the statement of Lemma B.5. Let $\mathbb{P}_i(\overline{T}_1, \overline{T}_2)$ be the proposition

$$\left(\overline{T}_1 \leq \textstyle\bigvee_{T_1\to T_2\in\overline{P}_i} T_1\right) \wedge \left(\forall P' \subsetneq \overline{P}_i.\ \left(\overline{T}_1 \leq \textstyle\bigvee_{T_1\to T_2\in P'} T_1\right) \vee \left(\textstyle\bigwedge_{T_1\to T_2\in\overline{P}_i\setminus P'} T_2 \leq \overline{T}_2\right)\right)$$

By Lemma B.5, we have

$$\textstyle\bigwedge_{T_1\to T_2\in\overline{P}_i} T_1 \to T_2 \leq \bigvee_{T_1\to T_2\in\overline{N}_i} T_1 \to T_2 \iff \exists(\overline{T}_1 \to \overline{T}_2) \in \overline{N}_i.\ \mathbb{P}_i(\overline{T}_1, \overline{T}_2)$$

and, since for all i such that $\mathscr{I}_i \not\leq \mathbb{0}$ the subtyping relation on the left does not hold, we know that $\mathbb{P}_i(\overline{T}_1, \overline{T}_2)$ is false for all such i and all $T_1 \to T_2 \in \overline{N}_i$.

We apply Lemma B.5 again to derive

$$\begin{aligned}
\textstyle\bigwedge_{T_1\to T_2\in\overline{P}_i} T_1 \to T_2 \leq \left(\bigvee_{T_1\to T_2\in\overline{N}_i} T_1 \to T_2\right) \vee (T' \to T'') \iff \\
\left(\exists(\overline{T}_1 \to \overline{T}_2) \in \overline{N}_i.\ \mathbb{P}_i(\overline{T}_1, \overline{T}_2)\right) \vee \mathbb{P}_i(T', T'')
\end{aligned}$$

and hence $\mathbb{P}_i(T', T'')$ must be true for all i verifying $\mathscr{I}_i \not\leq \mathbb{0}$ in order for subtyping to hold.

We have therefore shown

$$\begin{aligned}
T \leq T' \to T'' &\iff \forall i \in I, \mathscr{I}_i \not\leq \mathbb{0}.\ \mathbb{P}_i(T', T'') \\
&\iff \forall i \in I, \mathscr{I}_i \not\leq \mathbb{0}.\ \left(T' \leq \textstyle\bigvee_{T_1\to T_2\in\overline{P}_i} T_1\right) \wedge \\
&\qquad \left(\forall P' \subsetneq \overline{P}_i.\ \left(T' \leq \textstyle\bigvee_{T_1\to T_2\in P'} T_1\right) \vee \left(\textstyle\bigwedge_{T_1\to T_2\in\overline{P}_i\setminus P'} T_2 \leq T''\right)\right)
\end{aligned}$$

and we must now show

$$\forall i \in I, \mathscr{I}_i \not\leq \mathbb{0}.\ \left(T' \leq \bigvee_{T_1 \to T_2 \in \overline{P}_i} T_1\right) \wedge \left(\forall P' \subsetneq \overline{P}_i.\ \left(T' \leq \bigvee_{T_1 \to T_2 \in P'} T_1\right) \vee \left(\bigwedge_{T_1 \to T_2 \in \overline{P}_i \setminus P'} T_2 \leq T''\right)\right) \iff \begin{cases} \forall (T_1', T_2') \in \phi(T).\ (T' \leq T_1') \vee (T_2' \leq T'') \\ \wedge \\ T' \leq \mathsf{dom}(T) \end{cases}$$

where

$$\mathsf{dom}(T) = \bigwedge_{i \in I, \mathscr{I}_i \not\leq \mathbb{0}} \bigvee_{T_1 \to T_2 \in \overline{P}_i} T_1$$

$$\phi(T) = \bigcup_{i \in I, \mathscr{I}_i \not\leq \mathbb{0}} \left\{ \left(\bigvee_{T_1 \to T_2 \in P'} T_1, \bigwedge_{T_1 \to T_2 \in \overline{P}_i \setminus P'} T_2 \right) \middle| P' \subsetneq \overline{P}_i \right\}$$

We first prove the implication from left to right. To prove $T' \leq \mathsf{dom}(T)$, note that

$$\left(\forall i \in I, \mathscr{I}_i \not\leq \mathbb{0}.\ T' \leq \bigvee_{T_1 \to T_2 \in \overline{P}_i} T_1\right) \implies T' \leq \bigwedge_{i \in I, \mathscr{I}_i \not\leq \mathbb{0}} \bigvee_{T_1 \to T_2 \in \overline{P}_i} T_1 \,.$$

To prove the first condition, consider an arbitrary $(T_1', T_2') \in \phi(T)$. We have, for some $i \in I$ and $P' \subsetneq \overline{P}_i$ (verifying $\mathscr{I}_i \not\leq \mathbb{0}$),

$$(T_1', T_2') = \left(\bigvee_{T_1 \to T_2 \in P'} T_1, \bigwedge_{T_1 \to T_2 \in \overline{P}_i \setminus P'} T_2 \right).$$

P' is necessarily also one of the P' considered in the premise of the implication, so the result follows.

To prove the reverse implication, consider an arbitrary $i \in I$. The first condition follows from $T' \leq \mathsf{dom}(T)$. Moreover, for every P', a pair exists in $\phi(T)$ such that the second condition holds. □

Lemma B.16. *Let T be a type frame such that $T \leq \mathbb{0} \to \mathbb{1}$. Then, $T \leq \mathsf{dom}(T) \to \mathbb{1}$. Moreover, if $T \leq T' \to \mathbb{1}$, then $T' \leq \mathsf{dom}(T)$.*

Proof. Consider the equivalence of Lemma B.15, with $T' = \mathsf{dom}(T)$ and $T'' = \mathbb{1}$. The three conditions on the right-hand side are all verified, the first two since $\mathbb{1}$ is the top element of subtyping and the third by reflexivity. Hence, $T \leq \mathsf{dom}(T) \to \mathbb{1}$.

When $T \leq T' \to \mathbb{1}$, again by Lemma B.15 we have $T' \leq \mathsf{dom}(T)$. □

Lemma B.17. *Let T be a type frame such that $T \leq \mathbb{0} \to \mathbb{1}$. If T is strongly polarized, then $\mathsf{dom}(T)$ is strongly negatively polarized.*

Proof. If T is strongly polarized, then, by Lemma B.7, $\mathsf{UDNF}(T)$ is strongly polarized too. We just check on the definition of $\mathsf{dom}(T)$ that every T_1 appears in positive position, whereas it appeared in negative position in $\mathsf{UDNF}(T)$ (because it appeared on the left on an arrow $T_1 \to T_2$ in positive position). □

Lemma B.18. *Let T and T' be type frames such that $T \leq \mathbb{0} \to \mathbb{1}$ and $T' \leq \mathsf{dom}(T)$. Then, $T \leq T' \to (T \circ T')$. Moreover, if $T \leq T' \to T''$, then $T \circ T' \leq T''$.*

Proof. We prove $T \leq T' \to (T \circ T')$ by Lemma B.15. We must show the two conditions

$$\forall (T_1', T_2') \in \phi(T).\ (T' \leq T_1') \vee (T_2' \leq (T \circ T')) \qquad T' \leq \mathsf{dom}(T)$$

the second of which holds by hypothesis. To show the first condition, we take an arbitrary $(T_1', T_2') \in \phi(T)$. Either $T' \leq T_1'$ holds or not. If it does not hold, then $T_2' \leq T \circ T'$ holds because T_2 is a summand in the union of $T \circ T'$.

Now, assuming $T \leq T' \to T''$, we must show $T \circ T' \leq T''$. By Lemma B.15, we have

$$\forall (T_1', T_2') \in \phi(T).\ (T' \leq T_1') \vee (T_2' \leq T'')$$

Hence, for every $(T_1', T_2') \in \phi(T)$ such that $T' \not\leq T_1'$, we have $T_2' \leq T''$. Then the union of all such T_2' is also a subtype of T'', which shows that $T \circ T'$ s a subtype of T'' as well. □

Lemma B.19. *Let T and T' be type frames such that $T \leq \mathbb{0} \to \mathbb{1}$ and $T' \leq \mathsf{dom}(T)$. If T is strongly polarized and T' is strongly negatively polarized, then $T \circ T'$ is strongly polarized.*

Proof. If T is strongly polarized, then, by Lemma B.7, $\mathsf{UDNF}(T)$ is strongly polarized too. We can check on the definition of $T \circ T'$ that subterms of $\mathsf{UDNF}(T)$ in it are all in positive position and they were in positive position also in $\mathsf{UDNF}(T)$. □

B.4. Normal forms on casts and operators

In this section, we introduce the notion of normal forms for set-theoretic gradual types, and use it do define operators on types and casts. The formal definition of the operators are presented in Definition B.36 and Definition B.40, following the introduction of many preliminary results about ground types and disjunctive normal forms for gradual types.

Proposition B.20. *For all types τ, τ' such that $\tau' \preccurlyeq \tau$, it holds that $\tau' \preccurlyeq \tau/\tau' \preccurlyeq \tau$.*

Proof. By induction on the pair (τ', τ), and by cases on τ'.

- $\tau' = ?$. Note that $\tau' \preccurlyeq \tau/\tau'$ always holds. We then reason by cases on τ to prove the second precision relation.
  - $\tau = ?$. The result is immediate since $\tau = \tau/\tau' = ?$.
  - $\tau = \alpha$. Once again, $\tau = \tau/\tau' = \alpha$.
  - $\tau = b$. Once again, $\tau = \tau/\tau' = b$.
  - $\tau = \tau_1 \times \tau_2$. Then $\tau/\tau' = ? \times ?$, and it holds that $? \times ? \preccurlyeq \tau_1 \times \tau_2$.
  - $\tau = \tau_1 \to \tau_2$. Then $\tau/\tau' = ? \to ?$, and it holds that $? \to ? \preccurlyeq \tau_1 \to \tau_2$.
  - $\tau = \tau_1 \vee \tau_2$. Then $\tau/\tau' = \tau_1/? \vee \tau_2/?$. For every $i \in \{1, 2\}$, we have $? \preccurlyeq \tau_i$ thus by induction hypothesis, $\tau_i/? \preccurlyeq \tau_i$. Finally, $\tau_1/? \vee \tau_2/? \preccurlyeq \tau_1 \vee \tau_2$.
  - $\tau = \neg\tau_0$. Then $\tau/? = \neg(\tau_0/?)$. By induction hypothesis, $\tau_0/? \preccurlyeq \tau_0$ thus $\neg(\tau_0/?) \preccurlyeq \neg\tau_0$.
  - $\tau = \mathbb{0}$. Then $\tau/\tau' = \mathbb{0}$ and the result is immediate.

- $\tau' = \tau'_1 \vee \tau'_2$. Then necessarily $\tau = \tau_1 \vee \tau_2$, with $\tau'_i \leqslant \tau_i$ for every $i \in \{1, 2\}$. By induction hypothesis, $\tau'_i \leqslant \tau_i / \tau'_i \leqslant \tau_i$, and the result follows.
- $\tau' = \neg\tau'_0$. Then necessarily $\tau = \neg\tau_0$ with $\tau'_0 \leqslant \tau_0$. By induction hypothesis, $\tau'_0 \leqslant \tau_0 / \tau'_0 \leqslant \tau_0$ and the result follows.
- Otherwise, $\tau / \tau' = \tau'$ and since $\tau' \leqslant \tau$ the result is immediate.

□

Lemma B.21. *For all types τ, τ' that do not contain type connectives, if $\tau \leqslant \tau'$ and $\tau' / \tau \neq \tau$ then $\tau = ?$.*

Proof. Eliminating all cases involving connectives in Definition B.1 as well as the case $\tau' / \tau = \tau$, the only remaining cases are those were $\tau = ?$. □

Lemma B.22. *For all types τ, τ' that do not contain type connectives such that $\tau' / \tau = \tau'$, then $\tau' = \tau$ or $\tau = ?$.*

Proof. Suppose that $\tau \neq ?$. Eliminating all cases involving connectives or where $\tau = ?$ in Definition B.1, the only remaining case is $\tau' / \tau = \tau$. However, by hypothesis, $\tau' / \tau = \tau'$ therefore $\tau = \tau'$. □

Lemma B.23. *For all types τ, τ' such that $\tau' / \tau = \tau'$, the following holds:*

$$\forall d^L \in [\![\tau'^{\mathbb{O}}]\!], d^{L \cup \{X_1\} \setminus \{X_0\}} \in [\![\tau^{\mathbb{O}}]\!]$$

$$\forall d^L \notin [\![\tau'^{\mathbb{O}}]\!], d^{L \cup \{X_1\} \setminus \{X_0\}} \notin [\![\tau^{\mathbb{O}}]\!]$$

Proof. The two results are proved simultaneously by induction over the pair (d^L, τ).

- $\tau = ?$. Since $X_1 \in \mathrm{tags}(d^{L \cup \{X_1\} \setminus \{X_0\}})$, it is immediate that $d^{L \cup \{X_1\} \setminus \{X_0\}} \in [\![\tau^{\mathbb{O}}]\!]$. Moreover, $X_0 \notin \mathrm{tags}(d^{L \cup \{X_1\} \setminus \{X_0\}})$ hence $d^{L \cup \{X_1\} \setminus \{X_0\}} \notin [\![\tau^{\mathbb{O}}]\!]$.
- $\tau = \alpha$. By hypothesis, we have $\tau = \tau' = \alpha$. Therefore, for every $d^L \in [\![\tau'^{\mathbb{O}}]\!]$, it holds that $\alpha \in L$. Thus $\alpha \in \mathrm{tags}(d^{L \cup \{X_1\} \setminus \{X_0\}})$, hence the first result. The second result is proved using the same reasoning.
- $\tau = b$. By hypothesis, since $\tau' / \tau = \tau'$, we have $\tau = \tau' = b$, and the result is immediate since the interpretation of a constant contains all possible sets of labels.
- $\tau = \tau_1 \times \tau_2$. Since $\tau' / \tau = \tau'$, necessarily $\tau = \tau'$ and the result is immediate for the same reason as the previous case.
- $\tau = \tau_1 \to \tau_2$. Once again, necessarily $\tau = \tau'$ and the result is immediate.
- $\tau = \tau_1 \vee \tau_2$. Let $d^L \in [\![\tau^{\mathbb{O}}]\!]$. There exists $i \in \{1, 2\}$ such that $d^L \in [\![\tau_i^{\mathbb{O}}]\!]$. Thus, by induction hypothesis, it holds that $d^{L \cup \{X_1\} \setminus \{X_0\}} \in [\![\tau_i^{\mathbb{O}}]\!]$. Therefore, we have $d^{L \cup \{X_1\} \setminus \{X_0\}} \in [\![\tau^{\mathbb{O}}]\!]$, which is the result. The same reasoning can be done for the second case.
- $\tau = \neg\tau'$. Let $d^L \in [\![\tau^{\mathbb{O}}]\!]$. By definition, $d^L \notin [\![\tau'^{\mathbb{O}}]\!]$. By induction hypothesis, we

therefore have $d^{L\cup\{X_1\}\setminus\{X_0\}} \notin [\![\tau'^{\mathbb{O}}]\!]$. Thus, $d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![\tau^{\mathbb{O}}]\!]$. The same reasoning can be done for the second result.

□

Corollary B.24. *For all types τ, τ' such that $\tau'/\tau = \tau'$, and all types τ_l, τ_r such that $\tau \leq \tau_l \to \tau_r$, then $\tau' \leq \tau_l \to \tau_r$.*

Proof. Let $d^L \in [\![\tau'^{\mathbb{O}}]\!]$. By Lemma B.23, we know that $d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![\tau^{\mathbb{O}}]\!]$. Moreover, by hypothesis, $\tau \leq \tau_l \to \tau_r$, thus, by Theorem 5.18, $d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![(\tau_l \to \tau_r)^{\mathbb{O}}]\!]$. However, the fact that an element of $\mathscr{D}$ belongs to $[\![(\tau_l \to \tau_r)^{\mathbb{O}}]\!]$ is independent of its set of labels, therefore $d^L \in [\![(\tau_l \to \tau_r)^{\mathbb{O}}]\!]$. Thus, we obtain that $\tau'^{\mathbb{O}} \leq (\tau_l \to \tau_r)^{\mathbb{O}}$ and the result follows by Theorem 5.18. □

Corollary B.25. *For all types τ, τ' such that $\tau'/\tau = \tau'$, and all types τ_l, τ_r such that $\tau \leq \tau_l \times \tau_r$, then $\tau' \leq \tau_l \times \tau_r$.*

Proof. Let $d^L \in [\![\tau'^{\mathbb{O}}]\!]$. By Lemma B.23, we know that $d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![\tau^{\mathbb{O}}]\!]$. Moreover, by hypothesis, $\tau \leq \tau_l \times \tau_r$, thus, by Theorem 5.18, $d^{L\cup\{X_1\}\setminus\{X_0\}} \in [\![(\tau_l \times \tau_r)^{\mathbb{O}}]\!]$. However, the fact that an element of $\mathscr{D}$ belongs to $[\![(\tau_l \times \tau_r)^{\mathbb{O}}]\!]$ is independent of its set of labels, therefore $d^L \in [\![(\tau_l \times \tau_r)^{\mathbb{O}}]\!]$. Thus, we obtain that $\tau'^{\mathbb{O}} \leq (\tau_l \times \tau_r)^{\mathbb{O}}$ and the result follows by Theorem 5.18. □

Definition B.26. *We define the function m recursively on $\mathscr{D}$ as follows:*

$$
\begin{aligned}
m : \mathscr{D} \cup \{\Omega\} &\to \mathscr{D} \cup \{\Omega\} \\
m(\Omega) &= \Omega \\
m(c^L) &= c^{L\cup\{X_0\}\setminus\{X_1\}} \\
m((d_l, d_r)^L) &= (m(d_l), m(d_r))^{L\cup\{X_0\}\setminus\{X_1\}} \\
m((d_1, d_1'), \ldots, (d_n, d_n')^L) &= (m(d_1), m(d_1')), \ldots, (m(d_n), m(d_n'))^{L\cup\{X_0\}\setminus\{X_1\}}
\end{aligned}
$$

Lemma B.27. *For all types τ, τ' such that $\tau \preccurlyeq \tau'$, the following holds:*

$$\forall d \in [\![\tau'^{\mathbb{O}}]\!], m(d) \in [\![\tau^{\mathbb{O}}]\!]$$

$$\forall d \notin [\![\tau'^{\mathbb{O}}]\!], m(d) \notin [\![\tau^{\mathbb{O}}]\!]$$

Proof. The two results are proved simultaneously by induction on the pair (d, τ).

- $\tau = ?$. For every $d \in \mathscr{D}$, $X_0 \in \mathrm{tags}(m(d))$ by Definition B.26. Therefore $m(d) \in [\![X_0]\!] = [\![\tau^{\mathbb{O}}]\!]$. Similarly, $X_1 \notin \mathrm{tags}(m(d))$, thus $m(d) \notin [\![X_1]\!] = [\![\tau^{\mathbb{O}}]\!]$.
- $\tau = \alpha$. Immediate since, by hypothesis, $\tau \preccurlyeq \tau'$ therefore $\tau = \tau' = \alpha$.
- $\tau = b$. Immediate since, by hypothesis, $\tau \preccurlyeq \tau'$ therefore $\tau = \tau' = b$.
- $\tau = \tau_1 \times \tau_2$. By hypothesis, $\tau' = \tau_1' \times \tau_2'$ with $\tau_1 \preccurlyeq \tau_1'$ and $\tau_2 \preccurlyeq \tau_2'$. Let $d \in [\![\tau'^{\mathbb{O}}]\!]$. Since

$[\![\tau'^{\mathbb{O}}]\!] = [\![\tau_1'^{\mathbb{O}} \times \tau_2'^{\mathbb{O}}]\!]$, $d = (d_1, d_2)^L$ for some $d_1, d_2 \in \mathscr{D}$, where $d_i \in [\![\tau_i'^{\mathbb{O}}]\!]$, for every $i \in \{1, 2\}$. By induction, it holds that $m(d_i) \in [\![\tau_i^{\mathbb{O}}]\!]$, thus $(m(d_1), m(d_2))^{L'} \in [\![\tau_1^{\mathbb{O}} \times \tau_2^{\mathbb{O}}]\!]$ for every set of tags L'. Hence $m(d) \in [\![\tau^{\mathbb{O}}]\!]$.

Similarly, let $d \notin [\![\tau'^{\mathbb{O}}]\!]$. If $d \neq (d_1, d_2)^L$ for some $d_1, d_2 \in \mathscr{D}$, then it is immediate that $m(d) \notin [\![\tau^{\mathbb{O}}]\!]$ since it only contains pairs. Otherwise, if $d = (d_1, d_2)^L$ for some $d_1, d_2 \in \mathscr{D}$, then $d_i \notin [\![\tau_i'^{\mathbb{O}}]\!]$ for some $i \in \{1, 2\}$. By induction, it holds that $m(d_i) \notin [\![\tau_i^{\mathbb{O}}]\!]$. Therefore, $(m(d_1), m(d_2))^{L'} \notin [\![\tau_1^{\mathbb{O}} \times \tau_2^{\mathbb{O}}]\!]$ for every set of tags L', hence the result.

- $\tau = \tau_1 \to \tau_2$. By hypothesis, $\tau' = \tau_1' \to \tau_2'$ with $\tau_1 \preccurlyeq \tau_1'$ and $\tau_2 \preccurlyeq \tau_2'$. For every $d \in [\![\tau'^{\mathbb{O}}]\!]$, d is a relation $(d_1, d_1'), \dots, (d_n, d_n')^L$. Let $i \in \{1, n\}$ such that $m(d_i) \in [\![\tau_1^{\mathbb{O}}]\!]$. According to the contrapositive of the second induction hypothesis, $d_i \in [\![\tau_1'^{\mathbb{O}}]\!]$. Therefore, by definition of $[\![\tau'^{\mathbb{O}}]\!]$, $d_i' \in [\![\tau_2'^{\mathbb{O}}]\!]$. Applying the first induction hypothesis, $m(d_i') \in [\![\tau_2^{\mathbb{O}}]\!]$.

  To summarize, $m(d_i) \in [\![\tau_1^{\mathbb{O}}]\!] \implies m(d_i') \in [\![\tau_2^{\mathbb{O}}]\!]$.

  Therefore, $(m(d_1), m(d_1')), \dots, (m(d_n), m(d_n'))^{L'} \in [\![\tau^{\mathbb{O}}]\!]$ for every set of tags L, hence the first result.

  Similarly, for every $d \notin [\![\tau'^{\mathbb{O}}]\!]$, if d is not a relation then it is immediate that $m(d) \notin [\![\tau^{\mathbb{O}}]\!]$ since $m(d)$ is also not a relation and $[\![\tau^{\mathbb{O}}]\!]$ only contains relations. Otherwise, if $d = (d_1, d_1'), \dots, (d_n, d_n')^L$, then, by definition of $[\![\tau'^{\mathbb{O}}]\!]$, there exists a $i \in \{1, n\}$ such that $d_i \in \tau_1'^{\mathbb{O}}$ and $d_i' \notin \tau_2'^{\mathbb{O}}$. The induction hypothesis yields $m(d_i) \in \tau_1^{\mathbb{O}}$ and $m(d_i') \notin \tau_2^{\mathbb{O}}$. Thus, $d \notin [\![\tau^{\mathbb{O}}]\!]$ independently of its set of tags, hence the result.

- $\tau = \tau_1 \vee \tau_2$. By hypothesis, $\tau' = \tau_1' \vee \tau_2'$ with $\tau_1 \preccurlyeq \tau_1'$ and $\tau_2 \preccurlyeq \tau_2'$. For every $d \in [\![\tau'^{\mathbb{O}}]\!]$, $d \in [\![\tau_i'^{\mathbb{O}}]\!]$ for some $i \in \{1, 2\}$. Thus, by induction hypothesis, $m(d) \in [\![\tau_i^{\mathbb{O}}]\!] \subset [\![\tau^{\mathbb{O}}]\!]$, hence the result.

  Similarly, for every $d \notin [\![\tau'^{\mathbb{O}}]\!]$, $d \notin [\![\tau_i'^{\mathbb{O}}]\!]$ for every $i \in \{1, 2\}$. Thus, by induction hypothesis, $m(d) \notin [\![\tau_i^{\mathbb{O}}]\!]$ for every $i \in \{1, 2\}$, hence the result.

- $\tau = \neg\tau_0$. By hypothesis, $\tau' = \neg\tau_0'$ with $\tau_0 \preccurlyeq \tau_0'$. Let $d \in [\![\tau'^{\mathbb{O}}]\!]$. By definition, $d \notin [\![\tau_0'^{\mathbb{O}}]\!]$. By induction, we have $m(d) \notin [\![\tau_0^{\mathbb{O}}]\!]$, hence $m(d) \in [\![\tau^{\mathbb{O}}]\!]$. We can do the same reasoning for $d \notin [\![\tau'^{\mathbb{O}}]\!]$, which concludes this proof.

□

Corollary B.28. *For all types τ, τ' such that $\tau \preccurlyeq \tau'$, if $\tau \leq \mathbb{0} \to \mathbb{1}$ then $\tau' \leq \mathbb{0} \to \mathbb{1}$.*

Proof. Let $d \in [\![\tau'^{\mathbb{O}}]\!]$. By Lemma B.27, it holds that $m(d) \in [\![\tau^{\mathbb{O}}]\!]$. Since $\tau \leq \mathbb{0} \to \mathbb{1}$, Theorem 5.18 yields $\tau^{\mathbb{O}} \leq \mathbb{0} \to \mathbb{1}$. Thus, $m(d) \in [\![\mathbb{0} \to \mathbb{1}]\!]$, which implies that $m(d) = R^L$ for some relation $R \subset \mathscr{D} \times \mathscr{D} \cup \{\Omega\}$.

By inversion of Definition B.26, $d = R'^L$ for some relation $R' \subset \mathscr{D} \times \mathscr{D} \cup \{\Omega\}$. Thus $d \in [\![\mathbb{0} \to \mathbb{1}]\!]$, which yields that $\tau'^{\mathbb{O}} \leq \mathbb{0} \to \mathbb{1}$. Theorem 5.18 then gives the result. □

Corollary B.29. *For all types τ, τ' such that $\tau \preccurlyeq \tau'$, if $\tau \leq \mathbb{1} \times \mathbb{1}$ then $\tau' \leq \mathbb{1} \times \mathbb{1}$.*

Proof. Let $d \in [\![\tau'^{\mathbb{O}}]\!]$. By Lemma B.27, it holds that $m(d) \in [\![\tau^{\mathbb{O}}]\!]$. Since $\tau \leq \mathbb{1} \times \mathbb{1}$, Theorem 5.18 yields $\tau^{\mathbb{O}} \leq \mathbb{1} \times \mathbb{1}$. Thus, $m(d) \in [\![\mathbb{1} \times \mathbb{1}]\!]$, which implies that $m(d) = (d_l, d_r)^L$ for some $d_l, d_r \in \mathscr{D}$.

By inversion of Definition B.26, $d = (d'_l, d'_r)^L$ for some $d'_l, d'_r \in \mathscr{D}$. Thus $d \in [\![\mathbb{1} \times \mathbb{1}]\!]$, which yields that $\tau'^{\mathbb{O}} \leq \mathbb{1} \times \mathbb{1}$. Theorem 5.18 then gives the result. □

Corollary B.30. *For all types τ, τ' such that $\tau \preccurlyeq \tau'$, if $\tau' \not\leq \mathbb{0}$ then $\tau \not\leq \mathbb{0}$.*

Proof. Since $\tau' \not\leq \mathbb{0}$, by Theorem 5.18, it holds that $\tau'^{\mathbb{O}} \not\leq \mathbb{0}$. Thus, there exists $d \in [\![\tau'^{\mathbb{O}}]\!]$. Applying Lemma B.27 yields $m(d) \in [\![\tau^{\mathbb{O}}]\!]$, therefore $\tau^{\mathbb{O}} \not\leq \mathbb{0}$. Theorem 5.18, then yields the result. □

We now extend the previous definition of atoms to gradual types. That is, we refer to a gradual type of the form b, $\tau_1 \times \tau_2$, or $\tau_1 \to \tau_2$ as an atom. We write $\mathscr{B}^?$, $\mathscr{A}^?_{prod}$, and $\mathscr{A}^?_{fun}$ for the set of gradual types of the forms b, $\tau_1 \times \tau_2$, and $\tau_1 \to \tau_2$, respectively.

In the following, the metavariable a ranges over the set $\mathscr{B}^? \cup \mathscr{A}^?_{prod} \cup \mathscr{A}^?_{fun} \cup \mathscr{V}^\alpha \cup \{?\}$.

Definition B.31 (Uniform gradual normal form)**.** *A* uniform gradual (disjunctive) normal form *(UGDNF) is a gradual type τ of the form*

$$\bigvee_{i \in I} \left(\bigwedge_{a \in P_i} a \wedge \bigwedge_{a \in N_i} \neg a \right)$$

such that, for all $i \in I$, one of the following three condition holds:

- $P_i \cap \mathscr{B}^? \neq \emptyset$ *and* $(P_i \cup N_i) \cap (\mathscr{A}^?_{prod} \cup \mathscr{A}^?_{fun}) = \emptyset$*;*
- $P_i \cap \mathscr{A}^?_{prod} \neq \emptyset$ *and* $(P_i \cup N_i) \cap (\mathscr{B}^? \cup \mathscr{A}^?_{fun}) = \emptyset$*;*
- $P_i \cap \mathscr{A}^?_{fun} \neq \emptyset$ *and* $(P_i \cup N_i) \cap (\mathscr{B}^? \cup \mathscr{A}^?_{prod}) = \emptyset$*;*

For every type τ, we define $\mathsf{UGDNF}(\tau) = (\mathsf{UDNF}(\tau^\oplus))^\dagger$.

Lemma B.32. *For every type τ, $\mathsf{UGDNF}(\tau)$ is in uniform gradual normal form and $\mathsf{UGDNF}(\tau) \simeq \tau$.*

Proof. We define $\tau' = \mathsf{UGDNF}(\tau)$. From the definition of UGDNF, it is immediate that τ' is in uniform gradual normal form.

Lemma B.7 ensures that $\mathsf{UDNF}(\tau^\oplus) \simeq \tau^\oplus$. Moreover, since UDNF preserves the strong polarization, $\mathsf{UDNF}(\tau^\oplus)$ is strongly polarized. By unicity of the strong polarization, $((\mathsf{UDNF}(\tau^\oplus))^\dagger)^\oplus = \mathsf{UDNF}(\tau^\oplus) \simeq \tau^\oplus$. Theorem 5.18 then yields that $(\mathsf{UDNF}(\tau^\oplus))^\dagger \simeq \tau$, that is, $\tau' \simeq \tau$. □

Lemma B.33. *For every pair of types τ, τ' such that $\tau / \tau' = \tau'$, the following results hold:*

- $\tau' \in \mathscr{B}^? \iff \tau \in \mathscr{B}^?$
- $\tau' \in \mathscr{A}^?_{fun} \iff \tau \in \mathscr{A}^?_{fun}$
- $\tau' \in \mathscr{A}^?_{prod} \iff \tau \in \mathscr{A}^?_{prod}$
- $\tau' = ? \iff \tau = ?$
- $\tau' \in \mathscr{V}^\alpha \iff \tau \in \mathscr{V}^\alpha$

Proof. The result follows immediately from the definition of τ / τ'. □

We now prove the following lemma about the function $\mathscr{N}$ defined in Subsection B.3.

Lemma B.34. *For every pair of type frames T, T' such that $T^\dagger / T'^\dagger = T'^\dagger$, the following holds:*

$$\mathscr{N}(T)^\dagger / \mathscr{N}(T')^\dagger = \mathscr{N}(T')^\dagger$$

$$\mathscr{N}'(T)^\dagger / \mathscr{N}'(T')^\dagger = \mathscr{N}'(T')^\dagger$$

Proof. By induction on the pair (T, T'), and by cases on T'. Since $\neg\tau / \neg\tau' = \neg(\tau / \tau')$, most of the cases are proved similarly for $\mathscr{N}$ and $\mathscr{N}'$, and may be omitted.

- $T' = X'$. Then $T'^\dagger = ?$. Thus, by hypothesis, $T^\dagger = ?$ and therefore $T = X$. In this case $\mathscr{N}(T) = X$ and $\mathscr{N}(T') = X'$, and the result follows.
- $T' = \alpha$. Then necessarily $T = \alpha$ and $\mathscr{N}$ leaves T and T' unchanged, and the result is immediate.
- $T' = b$. Same as previous case.
- $T' = T'_1 \times T'_2$. By hypothesis, T is of the form $T_1 \times T_2$. Thus $\mathscr{N}$ leaves T and T' unchanged, and the result follows.
- $T' = T'_1 \to T'_2$. Same as previous case.
- $T' = T'_1 \vee T'_2$. By hypothesis, T is of the form $T_1 \vee T_2$ where for every $i \in \{1, 2\}$, $T_i / T'_i = T'_i$. By induction and definition of $\mathscr{N}$, the first result is immediate.

  For the second result, consider $k \in \{1, 2\}$. We have $\mathscr{N}'(T_k) = \bigvee_{i \in I_k} \left(\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n \right)$. The induction hypothesis ensures that we also have $\mathscr{N}'(T'_k) = \bigvee_{i \in I_k} \left(\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n \right)$ where for every $i \in I_k$, for every $p \in P_i$, $a_p^\dagger / a_p'^\dagger = a_p'^\dagger$ and similarly for every $n \in N_i$.

  Thus, for every pair $(i_1, i_2) \in (I_1 \times I_2)$, noting $T_I = \bigwedge_{p \in P_{i_1} \cup P_{i_2}} a_p \wedge \bigwedge_{n \in N_{i_1} \cup N_{i_2}} \neg a_n$ and $T'_I = \bigwedge_{p \in P_{i_1} \cup P_{i_2}} a'_p \wedge \bigwedge_{n \in N_{i_1} \cup N_{i_2}} \neg a'_n$, we have $T_I^\dagger / T_I'^\dagger = T_I'^\dagger$. Taking the union over all pairs $(i_1, i_2) \in (I_1 \times I_2)$ yields the result.
- $T' = \neg T'_0$. By hypothesis, T is of the form $\neg T_0$, where $T_0^\dagger / T_0'^\dagger = T_0'^\dagger$. By induction hypothesis, $\mathscr{N}'(T_0)^\dagger / \mathscr{N}'(T'_0)^\dagger = \mathscr{N}'(T'_0)^\dagger$. Thus $\mathscr{N}(\neg T_0)^\dagger / \mathscr{N}(\neg T'_0)^\dagger = \mathscr{N}(\neg T'_0)^\dagger$, which yields the result. The same reasoning can be done with $\mathscr{N}'$.
- $T' = \mathbb{0}$. Necessarily $T = \mathbb{0}$, thus $\mathscr{N}$ leaves T and T' unchanged, and the result follows.

□

Proposition B.35. *For every pair of types τ, τ' such that $\tau / \tau' = \tau'$, $\mathsf{UGDNF}(\tau) / \mathsf{UGDNF}(\tau') = \mathsf{UGDNF}(\tau')$.*

Proof. Let τ, τ' be two types such that $\tau / \tau' = \tau'$. Then applying Lemma B.34 to $\tau^{\oplus}$ and $\tau'^{\oplus}$ immediately yields that $\mathcal{N}(\tau^{\oplus})^{\dagger} / \mathcal{N}(\tau'^{\oplus})^{\dagger} = \mathcal{N}(\tau'^{\oplus})^{\dagger}$.

Now, assuming that

$$\mathcal{N}(\tau'^{\oplus}) = \bigvee_{i \in I} \Big(\underbrace{\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n}_{\mathscr{I}'_i} \Big)$$

By definition of the grounding operation, we have

$$\mathcal{N}(\tau^{\oplus}) = \bigvee_{i \in I} \Big(\underbrace{\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n}_{\mathscr{I}_i} \Big)$$

where for every $i \in I$, for every $(p, n) \in P_i \times N_i$, $a_p^{\dagger} / {a'_p}^{\dagger} = {a'_p}^{\dagger}$ and $a_n^{\dagger} / {a'_n}^{\dagger} = {a'_n}^{\dagger}$.

Lemma B.33 then guarantees that for every $i \in I$, $a_p \in P_i \cap (\mathscr{B} \cup \mathscr{V}^{\alpha}) \iff a'_p \in P_i \cap (\mathscr{B} \cup \mathscr{V}^{\alpha})$. Therefore, following the definitions of Subsection B.3, posing $T_1 = \mathscr{I}_i^{\text{basic}}$ and $T_2 = \mathscr{I}_i'^{\text{basic}}$, it holds that $T_1^{\dagger} / T_2^{\dagger} = T_2^{\dagger}$. The same reasoning can be done for the product and function intersections, yielding $\mathsf{UDNF}(\tau^{\oplus})^{\dagger} / \mathsf{UDNF}(\tau'^{\oplus})^{\dagger} = \mathsf{UDNF}(\tau'^{\oplus})^{\dagger}$, and the result follows by definition of UGDNF. □

Definition B.36 (Function Cast Approximation)**.** *For every pair of types τ, τ' such that $\tau' \leq \mathbb{0} \to \mathbb{1}$, and every type σ, if*

$$\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n}_{\mathscr{I}_i}$$

$$\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n}_{\mathscr{I}'_i}$$

$$\forall i \in I, \mathscr{I}_i \not\leq \mathbb{0} \implies \mathscr{I}'_i \not\leq \mathbb{0}$$

$$\forall i \in I, \forall p \in P_i, a_p \in \mathscr{A}^{?}_{fun} \iff a'_p \in \mathscr{A}^{?}_{fun}$$

then we define the approximation of $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle$ applied to σ, noted $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \sigma$ as follows.

$$\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \sigma = \Bigg\langle \bigwedge_{\substack{i \in I \\ \mathscr{I}'_i \not\leq \mathbb{0}}} \bigwedge_{\sigma \leq \bigvee_{p \in S} \sigma'_p} \bigvee_{S \subseteq \overline{P_i}} \bigvee_{p \in S} \sigma_p \to \bigvee_{\substack{i \in I \\ \mathscr{I}'_i \not\leq \mathbb{0}}} \bigvee_{\sigma \not\leq \bigvee_{p \in S} \sigma'_p} \bigvee_{S \subseteq \overline{P_i}} \bigwedge_{p \in \overline{P_i} \setminus S} \tau_p$$

$$\overset{p}{\Rightarrow}$$

$$\bigwedge_{\substack{i \in I \\ \mathscr{I}'_i \not\leq \mathbb{0}}} \bigwedge_{\sigma \leq \bigvee_{p \in S} \sigma'_p} \bigvee_{S \subseteq \overline{P_i}} \bigvee_{p \in S} \sigma'_p \to \bigvee_{\substack{i \in I \\ \mathscr{I}'_i \not\leq \mathbb{0}}} \bigvee_{\sigma \not\leq \bigvee_{p \in S} \sigma'_p} \bigvee_{S \subseteq \overline{P_i}} \bigwedge_{p \in \overline{P_i} \setminus S} \tau'_p \Bigg\rangle$$

where, to ease the notation, we pose $\overline{P_i} = \{p \in P_i \mid a_p \in \mathscr{A}^{?}_{fun}\} = \{p \in P_i \mid a'_p \in \mathscr{A}^{?}_{fun}\}$ and for every $p \in \overline{P_i}$, $a_p = \sigma_p \to \tau_p$ and $a'_p = \sigma'_p \to \tau'_p$.

Otherwise, $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \sigma$ is undefined.

In the future, we use $P_i \cap \mathscr{A}^?_{fun}$ as a shorthand for both $\{p \in P_i \mid a_p \in \mathscr{A}^?_{fun}\}$ and $\{p \in P_i \mid a'_p \in \mathscr{A}^?_{fun}\}$, provided the fourth condition of the above definition holds.

Lemma B.37. *For every pair of types τ, τ', and every type σ, if $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \sigma = \langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau'_1 \to \tau'_2 \rangle$ then the following holds:*

$$\tau / \tau' = \tau' \implies \forall i, \tau_i / \tau'_i = \tau'_i$$
$$\tau' / \tau = \tau \implies \forall i, \tau'_i / \tau_i = \tau_i$$

Proof. Given two types τ, τ' such that $\tau / \tau' = \tau'$, and any type σ, Proposition B.35 ensures that

$$\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n$$

and

$$\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n$$

where, for every $i \in I$ and every $p \in P_i$, $a_p / a'_p = a'_p$. Therefore, for every $p \in P_i \cap \mathscr{A}^?_{fun}$, we know that $a_p = \sigma_p \to \tau_p$ and $a'_p = \sigma'_p \to \tau'_p$, and we have by definition of the grounding operator $\tau_p / \tau'_p = \tau'_p$ and $\sigma_p / \sigma'_p = \sigma'_p$. The result then immediately follows from Definition B.36, and from the definition of the grounding operator. The same reasoning can be done for $\tau' / \tau = \tau$. □

Lemma B.38. *For every pair of types τ, τ', and every type σ, if $\tau / \tau' = \tau'$ and $\tau' \leq \mathbb{0} \to \mathbb{1}$, then $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \sigma$ is well-defined.*

Proof. Since $\tau' \leq \mathbb{0} \to \mathbb{1}$ and $\tau' \preccurlyeq \tau$, Corollary B.28 yields that $\tau \leq \mathbb{0} \to \mathbb{1}$. Proposition B.35 then immediately ensures that

$$\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n}_{\mathscr{I}_i}$$

and

$$\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n}_{\mathscr{I}'_i}$$

We now prove the third condition of Definition B.36, that is,

$$\forall i \in I, \mathscr{I}_i \not\leq \mathbb{0} \implies \mathscr{I}'_i \not\leq \mathbb{0}$$

Let $i \in I$. By hypothesis, $\mathscr{I}_i / \mathscr{I}'_i = \mathscr{I}'_i$, which implies that $\mathscr{I}'_i \preccurlyeq \mathscr{I}_i$. Applying Corollary B.30 then yields the result.

For the fourth condition of Definition B.36, knowing that $\tau / \tau' = \tau'$, we have for every $i \in I$ and every $p \in P_i$, $a_p / a'_p = a'_p$. The result then follows by definition of the grounding operator. □

Lemma B.39. *For every pair of types τ, τ', and every type σ, if $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \sigma = \langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle$ then the following holds:*

1. $\sigma \leq \tau_1'$
2. $\tau_2' = \min\{\tau \mid \tau' \leq \sigma \to \tau\}$
3. $\tau \leq \tau_1 \to \tau_2$

Proof. In all the following, we pose

$$\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n}_{\mathscr{I}_i}$$

and

$$\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n}_{\mathscr{I}'_i}$$

as well as $\overline{P_i} = P_i \cap \mathscr{A}^?_{fun}$ and for every $p \in \overline{P_i}$, $a_p = \sigma_p \to \tau_p$ and $a'_p = \sigma'_p \to \tau'_p$.

Moreover, we know that, by hypothesis,

$$\forall i \in I, \mathscr{I}_i \not\leq \mathbb{0} \implies \mathscr{I}'_i \not\leq \mathbb{0}$$

1. Immediate by definition of τ_1', since it is an intersection of supertypes of σ.
2. Let τ_0 a type such that $\tau' \leq \sigma \to \tau_0$. By Theorem 5.18, we have $\tau'^{\oplus} \leq \sigma^{\ominus} \to \tau_0^{\oplus}$. By Lemma B.18, this implies that $\tau'^{\oplus} \circ \sigma^{\ominus} \leq \tau_0^{\oplus}$. Plugging in the definition of the result type, this gives:
$$\bigvee_{\substack{i \in I \\ \mathscr{I}'_i \not\leq \mathbb{0}}} \bigvee_{\sigma^{\ominus} \not\leq \bigvee_{p \in S} \sigma'^{\ominus}_p} \bigwedge_{p \in \overline{P_i} \setminus S} \tau'^{\oplus}_p \leq \tau_0^{\oplus}$$
According to Theorem 5.18, the condition $\sigma^{\ominus} \not\leq \bigvee_{p \in S} \sigma'^{\ominus}_p$ is equivalent to $\sigma \not\leq \bigvee_{p \in S} \sigma'_p$. Applying Theorem 5.18 a second time to the whole inequality then yields
$$\bigvee_{\substack{i \in I \\ \mathscr{I}'_i \not\leq \mathbb{0}}} \bigvee_{\sigma \not\leq \bigvee_{p \in S} \sigma'_p} \bigwedge_{p \in \overline{P_i} \setminus S} \tau'_p \leq \tau_0$$
that is, $\tau_2' \leq \tau_0$, hence the result.
3. - We first prove that $\tau \leq \tau_1 \to \mathbb{1}$. Let $i \in I$ and $S \subseteq \overline{P_i}$. It holds that $\bigvee_{p \in S} \sigma_p \leq \bigvee_{p \in \overline{P_i}} \sigma_p$ since the union in the left hand side contains fewer elements. This implies that
$$\bigwedge_{\substack{S \subseteq \overline{P_i} \\ \sigma \leq \bigvee_{p \in S} \sigma'_p}} \bigvee_{p \in S} \sigma_p \leq \bigvee_{p \in \overline{P_i}} \sigma_p$$

Thus taking the intersection for all $i \in I$ where $\mathscr{I}_i' \not\leq \mathbb{0}$,

$$\bigwedge_{\substack{i\in I\\ \mathscr{I}_i'\not\leq\mathbb{0}}} \bigwedge_{\sigma\leq\bigvee_{p\in S}\sigma'_p} \bigvee_{p\in S} \sigma_p \leq \bigwedge_{\substack{i\in I\\ \mathscr{I}_i'\not\leq\mathbb{0}}} \bigvee_{p\in\overline{P_i}} \sigma_p$$

which is

$$\tau_1 \leq \bigwedge_{\substack{i\in I\\ \mathscr{I}_i'\not\leq\mathbb{0}}} \bigvee_{p\in\overline{P_i}} \sigma_p$$

Moreover, since $\forall i \in I, \mathscr{I}_i \not\leq \mathbb{0} \implies \mathscr{I}_i' \not\leq \mathbb{0}$, we have

$$\tau_1 \leq \bigwedge_{\substack{i\in I\\ \mathscr{I}_i'\not\leq\mathbb{0}}} \bigvee_{p\in\overline{P_i}} \sigma_p \leq \bigwedge_{\substack{i\in I\\ \mathscr{I}_i\not\leq\mathbb{0}}} \bigvee_{p\in\overline{P_i}} \sigma_p$$

since the intersection on the left hand side contains more elements. Now applying Theorem 5.18 and remarking that the right hand side of the previous inequality corresponds to the definition of the domain operator, we get $\tau_1^{\ominus} \leq \mathrm{dom}(\tau^{\oplus})$. Lemma B.16 then yields $\tau^{\oplus} \leq \tau_1^{\ominus} \to \mathbb{1}$, and the result follows from Theorem 5.18.

- We then show that, for every $i \in I$, and every $S \subsetneq \overline{P_i}$,

$$\tau_1 \not\leq \bigvee_{p\in S} \sigma_p \implies \sigma \not\leq \bigvee_{p\in S} \sigma'_p \qquad (*)$$

Suppose that the left hand side holds. Plugging in the definition of τ_1, we have

$$\bigwedge_{\substack{i\in I\\ \mathscr{I}_i'\not\leq\mathbb{0}}} \bigwedge_{\sigma\leq\bigvee_{p\in S}\sigma'_p} \bigvee_{p\in S} \sigma_p \not\leq \bigvee_{p\in S} \sigma_p$$

This inequality must also hold for every term of the intersections on the left hand side, and in particular for i and S:

$$\sigma \leq \bigvee_{p\in S} \sigma'_p \implies \bigvee_{p\in S} \sigma_p \not\leq \bigvee_{p\in S} \sigma_p$$

Since the right hand side is always false, the left hand side cannot hold thus $\sigma \not\leq \bigvee_{p\in S} \sigma'_p$.

- Now consider $i \in I$. It holds that

$$\bigvee_{\substack{S\subsetneq\overline{P_i}\\ \tau_1\not\leq\bigvee_{p\in S}\sigma_p}} \bigwedge_{p\in\overline{P_i}\setminus S} \tau_p \leq \bigvee_{\substack{S\subsetneq\overline{P_i}\\ \sigma\not\leq\bigvee_{p\in S}\sigma'_p}} \bigwedge_{p\in\overline{P_i}\setminus S} \tau_p$$

since, according to $(*)$, the union on the right contains more elements. Now, taking the union on both sides for every $i \in I$ such that $\mathscr{I}_i' \not\leq \mathbb{0}$,

$$\bigvee_{\substack{i\in I\\ \mathscr{I}_i'\not\leq\mathbb{0}}} \bigvee_{\substack{S\subsetneq\overline{P_i}\\ \tau_1\not\leq\bigvee_{p\in S}\sigma_p}} \bigwedge_{p\in\overline{P_i}\setminus S} \tau_p \leq \bigvee_{\substack{i\in I\\ \mathscr{I}_i'\not\leq\mathbb{0}}} \bigvee_{\substack{S\subsetneq\overline{P_i}\\ \sigma\not\leq\bigvee_{p\in S}\sigma'_p}} \bigwedge_{p\in\overline{P_i}\setminus S} \tau_p = \tau_2$$

Using the condition $\forall i \in I, \mathscr{I}_i \not\leq \mathbb{0} \implies \mathscr{I}'_i \not\leq \mathbb{0}$, the union on the left contains more elements than the same union on $\mathscr{I}_i \not\leq \mathbb{0}$, yielding

$$\bigvee_{\substack{i \in I \\ \mathscr{I}_i \not\leq \mathbb{0}}} \bigvee_{\substack{S \subseteq \overline{P_i} \\ \tau_1 \not\leq \bigvee_{p \in S} \sigma_p}} \bigwedge_{p \in \overline{P_i} \setminus S} \tau_p \leq \tau_2$$

Using Theorem 5.18 and remarking that the left hand side corresponds to the definition of the result operator, we deduce that $\tau^{\oplus} \circ \tau_1^{\ominus} \leq \tau_2^{\oplus}$, thus $\tau^{\oplus} \leq \tau_1^{\ominus} \to \tau^{\oplus} \circ \tau_1^{\ominus} \leq \tau_1^{\ominus} \to \tau_2^{\oplus}$, and applying Theorem 5.18 yields the result.

□

Definition B.40 (Cast Projection)**.** *For every pair of types τ, τ' such that $\tau' \leq \mathbb{1} \times \mathbb{1}$ if*

(1) $\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n}_{\mathscr{I}_i}$

(2) $\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n}_{\mathscr{I}'_i}$

(3) $\forall i \in I, \mathscr{I}_i \not\leq \mathbb{0} \implies \mathscr{I}'_i \not\leq \mathbb{0}$

(4) $\forall j \in I, \forall N \subseteq \overline{N_j}, \forall i \in \{1, 2\}, \pi_i\left(\tau_N^j\right) \not\leq \mathbb{0} \implies \pi_i\left(\tau_N'^j\right) \not\leq \mathbb{0}$

(5) $\forall i \in I, \forall p \in P_i, a_p \in \mathscr{A}^?_{prod} \iff a'_p \in \mathscr{A}^?_{prod}$

(6) $\forall i \in I, \forall n \in N_i, a_n \in \mathscr{A}^?_{prod} \iff a'_n \in \mathscr{A}^?_{prod}$

then we define the i-th projection of $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle$, noted $\pi_i\left(\langle \tau \overset{p}{\Rightarrow} \tau' \rangle\right)$ as follows.

$$\pi_i\left(\langle \tau \overset{p}{\Rightarrow} \tau' \rangle\right) = \left\langle \bigvee_{\substack{j \in I \\ \mathscr{I}'_j \not\leq \mathbb{0}}} \bigvee_{\substack{N \subseteq \overline{N_j} \\ \pi_1\left(\tau_N'^j\right) \not\leq \mathbb{0} \\ \pi_2\left(\tau_N'^j\right) \not\leq \mathbb{0}}} \pi_i\left(\tau_N^j\right) \overset{p}{\Rightarrow} \bigvee_{\substack{j \in I \\ \mathscr{I}'_j \not\leq \mathbb{0}}} \bigvee_{\substack{N \subseteq \overline{N_j} \\ \pi_1\left(\tau_N'^j\right) \not\leq \mathbb{0} \\ \pi_2\left(\tau_N'^j\right) \not\leq \mathbb{0}}} \pi_i\left(\tau_N'^j\right) \right\rangle$$

where

$$\overline{P_i} = \{p \in P_i \mid a_p \in \mathscr{A}^?_{prod}\} = \{p \in P_i \mid a'_p \in \mathscr{A}^?_{prod}\}$$

$$\overline{N_i} = \{n \in N_i \mid a_n \in \mathscr{A}^?_{prod}\} = \{n \in N_i \mid a'_n \in \mathscr{A}^?_{prod}\}$$

$$\tau_N^i = \left(\bigwedge_{\substack{p \in \overline{P_i} \\ a_p = \tau_1 \times \tau_2}} \tau_1 \wedge \bigwedge_{\substack{n \in N \\ a_n = \tau_1 \times \tau_2}} \neg \tau_1, \bigwedge_{\substack{p \in \overline{P_i} \\ a_p = \tau_1 \times \tau_2}} \tau_2 \wedge \bigwedge_{\substack{n \in N_i \setminus N \\ a_n = \tau_1 \times \tau_2}} \neg \tau_2 \right)$$

$$\tau_N'^i = \left(\bigwedge_{\substack{p \in P_i \\ a'_p = \tau'_1 \times \tau'_2}} \tau'_1 \wedge \bigwedge_{\substack{n \in N \\ a'_n = \tau'_1 \times \tau'_2}} \neg \tau'_1, \bigwedge_{\substack{p \in P_i \\ a'_p = \tau'_1 \times \tau'_2}} \tau'_2 \wedge \bigwedge_{\substack{n \in N_i \setminus N \\ a'_n = \tau'_1 \times \tau'_2}} \neg \tau'_2 \right)$$

otherwise, $\pi_i\left(\langle \tau \overset{p}{\Rightarrow} \tau' \rangle\right)$ is undefined.

In the future, we use $P_i \cap \mathscr{A}^?_{prod}$ as a shorthand for both $\{p \in P_i \mid a_p \in \mathscr{A}^?_{prod}\}$ and $\{p \in P_i \mid a'_p \in \mathscr{A}^?_{prod}\}$, provided the fourth condition of the above definition holds; and similarly for

$N_i \cap \mathscr{A}^?_{prod}$ provided the fifth condition above holds.

Lemma B.41. *For every pair of types τ, τ', if $\pi_i \langle \tau \overset{p}{\Rightarrow} \tau' \rangle = \langle \tau_i \overset{p}{\Rightarrow} \tau'_i \rangle$ then the following holds:*

$$\tau / \tau' = \tau' \implies \tau_i / \tau'_i = \tau'_i$$
$$\tau' / \tau = \tau \implies \tau'_i / \tau_i = \tau_i$$

Proof. Given two types τ, τ' such that $\tau / \tau' = \tau'$, Proposition B.35 ensures that

$$\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n$$

and

$$\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n$$

where, for every $i \in I$ and every $p \in P_i$, $a_p / a'_p = a'_p$, and for every $n \in N_i$, $a_n / a'_n = a'_n$.

Therefore, for every $p \in P_i \cap \mathscr{A}^?_{prod}$, we know that $a_p = \tau_1 \times \tau_2$ and $a'_p = \tau'_1 \to \tau'_2$, and we have by definition of the grounding operator $\tau_1 / \tau'_1 = \tau'_1$ and $\tau_2 / \tau'_2 = \tau'_2$. The result then immediately follows from Definition B.40, and from the definition of the grounding operator. The same reasoning can be done for $\tau' / \tau = \tau$. □

Lemma B.42. *For every pair of types τ, τ', if $\tau / \tau' = \tau'$ and $\tau' \leq \mathbb{1} \times \mathbb{1}$, then $\pi_i \langle \tau \overset{p}{\Rightarrow} \tau' \rangle$ is well-defined.*

Proof. Since $\tau' \leq \mathbb{1} \times \mathbb{1}$ and $\tau' \preccurlyeq \tau$, Corollary B.29 yields that $\tau \leq \mathbb{1} \times \mathbb{1}$. Proposition B.35 then immediately ensures that

$$\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n}_{\mathscr{I}_i}$$

and

$$\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n}_{\mathscr{I}'_i}$$

We now prove the third condition of Definition B.40, that is,

$$\forall i \in I, \mathscr{I}_i \not\leq \mathbb{0} \implies \mathscr{I}'_i \not\leq \mathbb{0}$$

Let $i \in I$. By hypothesis, $\mathscr{I}_i / \mathscr{I}'_i = \mathscr{I}'_i$, which implies that $\mathscr{I}'_i \preccurlyeq \mathscr{I}_i$. Applying Corollary B.30 then yields the result.

The fourth condition is proven similarly, by remarking that for every $j \in I$ and every $N \subseteq \overline{N_j}$, $\pi_i (\tau'^j_N) \preccurlyeq \pi_i (\tau^j_N)$.

For the fifth condition of Definition B.36, knowing that $\tau / \tau' = \tau'$, we have for every $i \in I$ and every $p \in P_i$, $a_p / a'_p = a'_p$. The result then follows by definition of the grounding

operator.

The sixth condition can be proven using the same reasoning. □

Lemma B.43. *For every pair of types τ, τ' such that $\tau \leq \mathbb{1} \times \mathbb{1}$, if $\pi_1 \langle \tau \overset{p}{\Rightarrow} \tau' \rangle = \langle \tau_1 \overset{p}{\Rightarrow} \tau'_1 \rangle$ then the following holds:*

1. $\tau \leq (\tau_1 \times \mathbb{1})$
2. $\tau'_1 = \min\{\tau \mid \tau' \leq \tau \times \mathbb{1}\}$

Proof. In all the following, we pose

$$\mathsf{UGDNF}(\tau) = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n}_{\mathscr{I}_i}$$

and

$$\mathsf{UGDNF}(\tau') = \bigvee_{i \in I} \underbrace{\bigwedge_{p \in P_i} a'_p \wedge \bigwedge_{n \in N_i} \neg a'_n}_{\mathscr{I}'_i}$$

We also pose $\overline{P_i} = P_i \cap \mathscr{A}^?_{prod}$ and $\overline{N_i} = N_i \cap \mathscr{A}^?_{prod}$.
Finally, as in Definition B.40, we pose, for every $i \in I$ and every set $N \subseteq N_i$:

$$\tau^i_N = \Big(\bigwedge_{\substack{p \in \overline{P_i} \\ a_p = \tau_1 \times \tau_2}} \tau_1 \wedge \bigwedge_{\substack{n \in N \\ a_n = \tau_1 \times \tau_2}} \neg \tau_1, \bigwedge_{\substack{p \in \overline{P_i} \\ a_p = \tau_1 \times \tau_2}} \tau_2 \wedge \bigwedge_{\substack{n \in N_i \setminus N \\ a_n = \tau_1 \times \tau_2}} \neg \tau_2 \Big)$$

$$\tau'^i_N = \Big(\bigwedge_{\substack{p \in P_i \\ a'_p = \tau'_1 \times \tau'_2}} \tau'_1 \wedge \bigwedge_{\substack{n \in N \\ a'_n = \tau'_1 \times \tau'_2}} \neg \tau'_1, \bigwedge_{\substack{p \in P_i \\ a'_p = \tau'_1 \times \tau'_2}} \tau'_2 \wedge \bigwedge_{\substack{n \in N_i \setminus N \\ a'_n = \tau'_1 \times \tau'_2}} \neg \tau'_2 \Big)$$

1. Since $\tau \leq \mathbb{1} \times \mathbb{1}$, Theorem 5.18 yields $\tau^\oplus \leq \mathbb{1} \times \mathbb{1}$. Thus, Lemma B.11 gives $\tau^\oplus \leq \pi_1(\tau^\oplus) \times \mathbb{1}$. Plugging in the definition of π_1 on type frames, we obtain:

$$\tau^\oplus \leq \Big(\bigvee_{\substack{i \in I \\ \mathscr{I}_i \not\leq \mathbb{0}}} \bigvee_{\substack{N \subseteq \overline{N_i} \\ \pi_1(\tau^{\oplus i}_N) \not\leq \mathbb{0} \\ \pi_2(\tau^{\oplus i}_N) \not\leq \mathbb{0}}} \pi_1(\tau^{\oplus i}_N) \Big) \times \mathbb{1}$$

Now, remarking that $\pi_1(\tau^{\oplus i}_N) = (\pi_1(\tau^i_N))^\oplus$ and applying Theorem 5.18, we obtain that $\pi_1(\tau^{\oplus i}_N) \not\leq \mathbb{0} \iff \pi_1(\tau^i_N) \not\leq \mathbb{0}$. Condition (4) of Definition B.40 then yields $\pi_1(\tau^{\oplus i}_N) \not\leq \mathbb{0} \implies \pi_1(\tau'^i_N) \not\leq \mathbb{0}$. The same reasoning for the second projection yields $\pi_2(\tau^{\oplus i}_N) \not\leq \mathbb{0} \implies \pi_2(\tau'^i_N) \not\leq \mathbb{0}$. Using this and Condition (3) of Definition B.40, we deduce

$$\Big(\bigvee_{\substack{i \in I \\ \mathscr{I}_i \not\leq \mathbb{0}}} \bigvee_{\substack{N \subseteq \overline{N_i} \\ \pi_1(\tau^{\oplus i}_N) \not\leq \mathbb{0} \\ \pi_2(\tau^{\oplus i}_N) \not\leq \mathbb{0}}} \pi_1(\tau^{\oplus i}_N) \Big) \leq \Big(\bigvee_{\substack{i \in I \\ \mathscr{I}'_i \not\leq \mathbb{0}}} \bigvee_{\substack{N \subseteq \overline{N_i} \\ \pi_1(\tau'^i_N) \not\leq \mathbb{0} \\ \pi_2(\tau'^i_N) \not\leq \mathbb{0}}} (\pi_1(\tau^i_N))^\oplus \Big)$$

Since the unions on the right contain more elements than the unions on the left. Fi-

nally, we have

$$\tau^{\oplus} \leq \Big(\bigvee_{\substack{i \in I \\ \mathscr{I}'_i \not\simeq \mathbb{0}}} \bigvee_{\substack{N \subseteq \overline{N_i} \\ \pi_1(\tau'^i_N) \not\simeq \mathbb{0} \\ \pi_2(\tau'^i_N) \not\simeq \mathbb{0}}} (\pi_1(\tau^i_N))^{\oplus} \Big) \times \mathbb{1}$$

And applying Theorem 5.18 yields

$$\tau \leq \Big(\bigvee_{\substack{i \in I \\ \mathscr{I}'_i \not\simeq \mathbb{0}}} \bigvee_{\substack{N \subseteq \overline{N_i} \\ \pi_1(\tau'^i_N) \not\simeq \mathbb{0} \\ \pi_2(\tau'^i_N) \not\simeq \mathbb{0}}} \pi_1(\tau^i_N) \Big) \times \mathbb{1}$$

which is the result.

2. Let τ_0 such that $\tau' \leq \tau_0 \times \mathbb{1}$. We show that $\tau'_1 \leq \tau_0$.
   By Theorem 5.18, we have $\tau'^{\oplus} \leq \tau_0^{\oplus} \times \mathbb{1}$. Thus, by Lemma B.11, we have $\pi_1((\tau')^{\oplus}) \leq \tau_0^{\oplus}$. Plugging in the definition of the projection of a type frame yields:

$$\bigvee_{\substack{i \in I \\ \mathscr{I}'_i \not\simeq \mathbb{0}}} \bigvee_{\substack{N \subseteq \overline{N_i} \\ \pi_1(\tau'^{\oplus i}_N) \not\simeq \mathbb{0} \\ \pi_2(\tau'^{\oplus i}_N) \not\simeq \mathbb{0}}} \pi_1(\tau'^{\oplus i}_N) \leq \tau_0^{\oplus}$$

   Remarking that, for every $i \in \{1,2\}$ and every $j \in I$, $\pi_i(\tau'^{\oplus j}_N) = (\pi_i(\tau'^j_N))^{\oplus}$, and applying Theorem 5.18, we obtain

$$\pi_i(\tau'^{\oplus j}_N) \not\simeq \mathbb{0} \iff (\pi_i(\tau'^j_N))^{\oplus} \not\simeq \mathbb{0} \iff \pi_i(\tau'^j_N) \not\simeq \mathbb{0}$$

   Thus we have:

$$\bigvee_{\substack{i \in I \\ \mathscr{I}'_i \not\simeq \mathbb{0}}} \bigvee_{\substack{N \subseteq \overline{N_i} \\ \pi_1(\tau'^i_N) \not\simeq \mathbb{0} \\ \pi_2(\tau'^i_N) \not\simeq \mathbb{0}}} (\pi_1(\tau'^i_N))^{\oplus} \leq \tau_0^{\oplus}$$

   Then, applying Theorem 5.18 yields

$$\bigvee_{\substack{i \in I \\ \mathscr{I}'_i \not\simeq \mathbb{0}}} \bigvee_{\substack{N \subseteq \overline{N_i} \\ \pi_1(\tau'^i_N) \not\simeq \mathbb{0} \\ \pi_2(\tau'^i_N) \not\simeq \mathbb{0}}} \pi_1(\tau'^i_N) \leq \tau_0$$

   Remarking that the left hand side corresponds to the definition of τ'_1 yields the result: $\tau'_1 \leq \tau_0$.

□

Lemma B.44. *For every pair of types τ, τ', if $\tau \leq \mathbb{1} \times \mathbb{1}$ and $\pi_2 \langle \tau \overset{p}{\Rightarrow} \tau' \rangle = \langle \tau_2 \overset{p}{\Rightarrow} \tau'_2 \rangle$ then the following holds:*

1. $\tau \leq (\mathbb{1} \times \tau_2)$
2. $\tau'_2 = \min\{\tau \mid \tau' \leq \mathbb{1} \times \tau\}$

Proof. Same proof as Lemma B.43. □

B.5. Soundness results and proofs

Lemma B.45 (Progress for Cast Values)**.** *For every value V, every label p, and all types τ_1, τ_2, if $\emptyset \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$, then one of the following cases holds:*

- *$V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$ is a value*
- *there exists a term E such that $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow E$*
- *$V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow \texttt{blame}\ p$*

Proof. By hypothesis, $\emptyset \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$. Therefore, by inversion of the typing rules, it holds that $\emptyset \vdash V : \tau_1$ and we distinguish two main cases: $\tau_1 \preccurlyeq \tau_2$ or $\tau_2 \preccurlyeq \tau_1$. The proof is then done by subcases over τ_2 / τ_1 or τ_1 / τ_2. The case where $\tau_1 = \tau_2$ is a particular case that is handled separately.

- $\tau_1 = \tau_2$. In this case, $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V$ by rule $[\mathrm{R}_{\mathsf{CastId}}]$.
- $\tau_1 \preccurlyeq \tau_2$ and $\tau_1 \neq \tau_2$. We distinguish the following subcases:
  - $\tau_2 / \tau_1 = \tau_1$. Then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$ is a value.
  - $\tau_2 / \tau_1 = \tau_2$. We proceed by case disjunction over V:
    - $V = V'\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle$ where $\tau_1' / \tau_2' = \tau_1'$. If $\tau_1' \leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V'$ by rule $[\mathrm{R}_{\mathsf{Collapse}}]$. Otherwise, if $\tau_1' \not\leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow \texttt{blame}\ p$ by rule $[\mathrm{R}_{\mathsf{Blame}}]$.
    - $V = V'\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle$ where $\tau_2' / \tau_1' = \tau_1'$. This case is identical to the previous one. If $\tau_1' \leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V'$ by rule $[\mathrm{R}_{\mathsf{Collapse}}]$. Otherwise, if $\tau_1' \not\leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow \texttt{blame}\ p$ by rule $[\mathrm{R}_{\mathsf{Blame}}]$.
    - $V = V'\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle$ where $\tau_1' / \tau_2' = \tau_2'$. If $\tau_2' \leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V$ by rule $[\mathrm{R}_{\mathsf{UpSimpl}}]$. Otherwise, if $\tau_2' \not\leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow \texttt{blame}\ p$ by rule $[\mathrm{R}_{\mathsf{UpBlame}}]$.
    - V is unboxed. If $\mathrm{type}(V) \leq \tau_2$ then $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V$ by rule $[\mathrm{R}_{\mathsf{UnboxSimpl}}]$. Otherwise, $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow \texttt{blame}\ p$ by rule $[\mathrm{R}_{\mathsf{UnboxBlame}}]$.
  - $\forall i \in \{1, 2\}, \tau_2 / \tau_1 \neq \tau_i$. In this case, $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 / \tau_1\rangle\langle \tau_2 / \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$ by rule $[\mathrm{R}_{\mathsf{ExpandR}}]$.
- $\tau_2 \preccurlyeq \tau_1$ and $\tau_1 \neq \tau_2$. We distinguish the following subcases:
  - $\tau_1 / \tau_2 = \tau_1$. In this case, $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$ is a value.
  - $\tau_1 / \tau_2 = \tau_2$. In this case, $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$ is a value.
  - $\forall i, \tau_1 / \tau_2 \neq \tau_i$. In this case, $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_1 / \tau_2\rangle\langle \tau_1 / \tau_2 \overset{p}{\Rightarrow} \tau_2\rangle$ by rule $[\mathrm{R}_{\mathsf{ExpandL}}]$.

□

Lemma B.46 (Progress)**.** *For every term E such that $\emptyset \vdash E : S$, one of the following cases holds:*

- *there exists a value V such that $E = V$*
- *there exists a term E' such that $E \hookrightarrow E'$*
- *there exists a label p such that $E \hookrightarrow \texttt{blame}\ p$*

Proof. By complete induction over the expression E.

- Case x. Impossible by hypothesis since a single variable cannot be well-typed in the empty environment.
- Case c. Immediate since c is a value.
- Case $\lambda^{\tau_1 \to \tau_2} x.\, E$. Immediate since $\lambda^{\tau_1 \to \tau_2} x.\, E$ is a value.
- Case $E_1\ E_2$. By inversion of the typing rules, we deduce that $\emptyset \vdash E_1 : \tau_1 \to \tau_2$ and $\emptyset \vdash E_2 : \tau_1$. We can thus apply the induction hypothesis on both E_1 and E_2, which yields the following subcases.
  - $\exists E_2'$ such that $E_2 \hookrightarrow E_2'$. By rule $[\mathrm{R}_{\mathsf{Context}}]$ and since $E_1\ \square$ is a valid reduction context, $E_1\ E_2 \hookrightarrow E_1\ E_2'$.
  - $E_2 \hookrightarrow \texttt{blame}\ p$. By rule $[\mathrm{R}_{\mathsf{CtxBlame}}]$ and since $E_1\ \square$ is a valid reduction context, $E_1\ E_2 \hookrightarrow \texttt{blame}\ p$.
  - E_2 is a value and $\exists E_1'$ such that $E_1 \hookrightarrow E_1'$. Since E_2 is a value, $\square\ E_2$ is a valid reduction context, thus $E_1\ E_2 \hookrightarrow E_1'\ E_2$ by rule $[\mathrm{R}_{\mathsf{Context}}]$.
  - E_2 is a value and $E_1 \hookrightarrow \texttt{blame}\ p$. Since E_2 is a value, $\square\ E_2$ is a valid reduction context, thus $E_1\ E_2 \hookrightarrow \texttt{blame}\ p$ by rule $[\mathrm{R}_{\mathsf{CtxBlame}}]$.
  - Both E_1 and E_2 are values. Reasoning by case analysis on E_1 and not considering ill-typed cases:
    - $E_1 = \lambda^{\tau_1' \to \tau_2'} x.\, E_1'$ where $\tau_1' \to \tau_2' \leq \tau_1 \to \tau_2$. In this case, $E_1\ E_2$ reduces to $E_1'[E_2/x]$ by rule $[\mathrm{R}_{\mathsf{App}}]$.
    - $E_1 = V\langle \tau_1' \overset{p}{\Rightarrow} \tau_2' \rangle$ where $\tau_2' \leq \tau_1 \to \tau_2$ and $\tau_2'/\tau_1' = \tau_1'$ or $\tau_1'/\tau_2' = \tau_2'$. If $\tau_1' \leq \mathbb{0} \to \mathbb{1}$ then $E_1 E_2$ reduces to $(V\, E_2 \langle \tau_l' \overset{\bar{p}}{\Rightarrow} \tau_l \rangle) \langle \tau_r \overset{p}{\Rightarrow} \tau_r' \rangle$ where $\langle \tau_1' \overset{p}{\Rightarrow} \tau_2' \rangle \circ \mathrm{type}(E_2) = \langle \tau_l \to \tau_r \overset{p}{\Rightarrow} \tau_l' \to \tau_r' \rangle$ by rule $[\mathrm{R}_{\mathsf{CastApp}}]$.
      Otherwise, if $\tau_1' \not\leq \mathbb{0} \to \mathbb{1}$, then $E_1\ E_2 \hookrightarrow \texttt{blame}\ p$ by rule $[\mathrm{R}_{\mathsf{FailApp}}]$.
    - $E_1 = V\langle \tau_1' \overset{p}{\Rightarrow} \tau_2' \rangle$ where $\tau_2' \leq \tau_1 \to \tau_2$ and $\tau_1'/\tau_2' = \tau_1'$. Then $E_1\ E_2 \hookrightarrow V\ E_2$ by rule $[\mathrm{R}_{\mathsf{SimplApp}}]$.
- Case $\Lambda\vec{\alpha}.\, E$. Immediate since $\Lambda\vec{\alpha}.\, E$ is a value.
- Case $E\ [\vec{t}]$. By inversion of the typing rule $[\mathrm{R}_{\mathsf{TApp}}]$, we deduce that $\emptyset \vdash E : \forall\vec{\alpha}.\tau$. We can thus apply the induction hypothesis on E which yields the following subcases:
  - $E \hookrightarrow E'$. Since $\square\ [\vec{t}]$ is a valid reduction context, $E\ [\vec{t}]$ reduces to $E'\ [\vec{t}]$ by $[\mathrm{R}_{\mathsf{Context}}]$.
  - $E \hookrightarrow \texttt{blame}\ p$. Since $\square\ [\vec{t}]$ is a valid reduction context, $E\ [\vec{t}]$ also reduces to $\texttt{blame}\ p$ by $[\mathrm{R}_{\mathsf{CtxBlame}}]$.

  - E is a value. In this case, by inversion of the typing rules, E is necessarily of the form $\Lambda\vec{\alpha}.\, E'$. Therefore, $E\ [\vec{t}] \hookrightarrow E'[\vec{t}/\vec{\alpha}]$ by $[\mathrm{R}_{\mathsf{TypeApp}}]$, concluding this case.

- Case (E_1, E_2). By inversion of the typing rule $[\mathrm{R}_{\mathsf{Pair}}]$, we deduce that $\emptyset \vdash E_i : \tau_i$, for $i \in \{1, 2\}$. Thus, we can apply the induction hypothesis on both E_1 and E_2, yielding the following subcases:
  - $E_2 \hookrightarrow E_2'$. Since $(E_1, \square)$ is a valid reduction context, $(E_1, E_2) \hookrightarrow (E_1, E_2')$ by rule $[\mathrm{R}_{\mathsf{Context}}]$.
  - $E_2 \hookrightarrow \mathtt{blame}\ p$. Since $(E_1, \square)$ is a valid reduction context, $(E_1, E_2) \hookrightarrow \mathtt{blame}\ p$ by rule $[\mathrm{R}_{\mathsf{CtxBlame}}]$.
  - E_2 is a value and $E_1 \hookrightarrow E_1'$. Since E_2 is a value, $(\square, E_2)$ is a valid reduction context, thus $(E_1, E_2) \hookrightarrow (E_1', E_2)$ by rule $[\mathrm{R}_{\mathsf{Context}}]$.
  - E_2 is a value and $E_1 \hookrightarrow \mathtt{blame}\ p$. Since E_2 is a value, $(\square, E_2)$ is a valid reduction context, thus $(E_1, E_2) \hookrightarrow \mathtt{blame}\ p$ by rule $[\mathrm{R}_{\mathsf{CtxBlame}}]$.
  - Both E_1 and E_2 are values. In this case, (E_1, E_2) is itself a value, concluding this case.

- Case $\pi_i\ E$. By inversion of the typing rule $[\mathrm{R}_{\mathsf{Proj}}]$, we deduce that $\emptyset \vdash E : \tau_1 \times \tau_2$. Thus, we can apply the induction hypothesis to E, yielding the following subcases:
  - $E \hookrightarrow E'$. Since $\pi_i\ \square$ is a valid reduction context, $\pi_i\ E \hookrightarrow \pi_i\ E'$ by rule $[\mathrm{R}_{\mathsf{Context}}]$.
  - $E \hookrightarrow \mathtt{blame}\ p$. Since $\pi_i\ \square$ is a valid reduction context, $\pi_i\ E$ reduces to $\mathtt{blame}\ p$ by rule $[\mathrm{R}_{\mathsf{CtxBlame}}]$.
  - E is a value. By cases on E, not considering the ill-typed cases:
    * $E = (V_1, V_2)$. In this case, $\pi_i\ E$ reduces to V_i by rule $[\mathrm{R}_{\mathsf{Proj}}]$.
    * $E = V\langle \tau_1' \overset{p}{\Rightarrow} \tau_2' \rangle$ where $\tau_2' \leq \tau_1 \times \tau_2$ and $\tau_2'/\tau_1' = \tau_1'$ or $\tau_1'/\tau_2' = \tau_2'$. In this case, if $\tau_1' \leq \mathbb{1} \times \mathbb{1}$ then $\pi_i\ E$ reduces to $(\pi_i\ V)\langle \tau_p \overset{p}{\Rightarrow} \tau_p' \rangle$ where $\langle \tau_p \overset{p}{\Rightarrow} \tau_p' \rangle = \pi_i\ (\langle \tau_1' \overset{p}{\Rightarrow} \tau_2' \rangle)$ by rule $[\mathrm{R}_{\mathsf{CastProj}}]$. Otherwise, if $\tau_1' \not\leq \mathbb{1} \times \mathbb{1}$, then $\pi_i\ E \hookrightarrow \mathtt{blame}\ p$ by rule $[\mathrm{R}_{\mathsf{FailProj}}]$.
    * $E = V\langle \tau_1' \overset{p}{\Rightarrow} \tau_2' \rangle$ where $\tau_2' \leq \tau_1 \times \tau_2$ and $\tau_1'/\tau_2' = \tau_1'$. Then $E \hookrightarrow \pi_i\ V$ by rule $[\mathrm{R}_{\mathsf{SimplProj}}]$.

- Case $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$. By inversion of the typing rule $[\mathrm{R}_{\mathsf{Let}}]$, we deduce that $\emptyset \vdash E_1 : \tau_1$. Therefore, we can apply the induction hypothesis to E_1, yielding the following subcases:
  - $E_1 \hookrightarrow E_1'$. Since $\mathsf{let}\ x = \square\ \mathsf{in}\ E_2$ is a valid reduction context, $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \hookrightarrow \mathsf{let}\ x = E_1'\ \mathsf{in}\ E_2$ by rule $[\mathrm{R}_{\mathsf{Context}}]$.
  - $E_1 \hookrightarrow \mathtt{blame}\ p$. Since $\mathsf{let}\ x = \square\ \mathsf{in}\ E_2$ is a valid reduction context, $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \hookrightarrow \mathtt{blame}\ p$ by rule $[\mathrm{R}_{\mathsf{CtxBlame}}]$.
  - E_1 is a value. We immediately deduce that $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \hookrightarrow E_2[E_1/x]$ by rule $[\mathrm{R}_{\mathsf{Let}}]$.

- Case $E\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$. By inversion of the typing rule $[\mathrm{R}_{\mathsf{Cast}}]$, we deduce that $\emptyset \vdash E : \tau_1$. Therefore, we can apply the induction hypothesis to E, yielding the following subcases:
  - $E \hookrightarrow E'$. Since $\square\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$ is a valid reduction context, $E\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \hookrightarrow E'\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$ by rule $[\mathrm{R}_{\mathsf{Context}}]$.
  - $E \hookrightarrow \mathtt{blame}\ p$. Since $\square\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$ is a valid reduction context, $E\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \hookrightarrow \mathtt{blame}\ p$ by rule $[\mathrm{R}_{\mathsf{CtxBlame}}]$.
  - E is a value. In this case, we can apply Lemma B.45 to $E\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$ which yields the result and concludes the proof.

□

Lemma B.47. *If $\Gamma, x : S' \vdash E : S$, then for every expression E' such that $\Gamma \vdash E' : S'$, we have $\Gamma \vdash E[E'/x] : S$.*

Proof. By induction on E.

- x. We have $S = S'$ and the result follows from $\Gamma \vdash E' : S'$ since $E[E'/x] = E'$.
- y. Immediate since $E[E'/x] = E$.
- c. Immediate since $E[E'/x] = E$.
- $\lambda^{\tau_1 \to \tau_2} y.\, E_y$. By inversion of the typing rules, we have $\tau_1 \to \tau_2 \leq S$, and $\Gamma, x : S', y : \tau_1 \vdash E_y : \tau_2$. Thus, by induction hypothesis, $\Gamma, y : \tau_1 \vdash E_y[E'/x] : \tau_2$. This implies that $\Gamma \vdash \lambda^{\tau_1 \to \tau_2} y.\, (E_y[E'/x]) : \tau_1 \to \tau_2$ by rule $[\mathrm{T}_{\mathsf{Abstr}}]$, and the result follows since $E[E'/x] = \lambda^{\tau_1 \to \tau_2} y.\, (E_y[E'/x])$.
- $E_1\ E_2$. By hypothesis, we have $\Gamma, x : S' \vdash E_1 : \tau_1 \to S$ and $\Gamma, x : S' \vdash E_2 : \tau_1$. By induction hypothesis, we deduce that $\Gamma \vdash E_1[E'/x] : \tau_1 \to S$ and $\Gamma \vdash E_2[E'/x] : \tau_1$. Therefore, $\Gamma \vdash (E_1[E'/x])(E_2[E'/x]) : S$ by rule $[\mathrm{T}_{\mathsf{App}}]$, hence the result.
- (E_1, E_2). By hypothesis, we have $\Gamma, x : S' \vdash E_1 : \tau_1$ and $\Gamma, x : S' \vdash E_2 : \tau_2$, where $\tau_1 \times \tau_2 \leq S$. By induction hypothesis, we deduce that $\Gamma \vdash E_1[E'/x] : \tau_1$ and $\Gamma \vdash E_2[E'/x] : \tau_2$. Therefore, $\Gamma \vdash (E_1[E'/x], E_2[E'/x]) : \tau_1 \times \tau_2$ by rule $[\mathrm{T}_{\mathsf{Pair}}]$, and the result follows.
- $\pi_i\ E_p$. By hypothesis, we have $\Gamma, x : S' \vdash E_p : (\tau_1 \times \tau_2)$, where $\tau_i \leq S$. By induction hypothesis, we deduce that $\Gamma \vdash E_p[E'/x] : (\tau_1 \times \tau_2)$. Therefore, $\Gamma \vdash \pi_i\ (E_p[E'/x]) : \tau_i$ by rule $[\mathrm{T}_{\mathsf{Proj}}]$, and the result follows.
- $\mathsf{let}\ y = E_1\ \mathsf{in}\ E_2$. By hypothesis, we have $\Gamma, x : S' \vdash E_1 : \forall \vec{\alpha}.\tau_1$ and $\Gamma, x : S', y : \forall \vec{\alpha}.\tau_1 \vdash E_2 : S$. Therefore, by induction hypothesis, we deduce $\Gamma \vdash E_1[E'/x] : \forall \vec{\alpha}.\tau_1$ and $\Gamma, y : \forall \vec{\alpha}.\tau_1 \vdash E_2[E'/x] : S$. This yields $\Gamma \vdash \mathsf{let}\ y = E_1[E'/x]\ \mathsf{in}\ E_2[E'/x] : S$ by rule $[\mathrm{T}_{\mathsf{Let}}]$, hence the result.
- $\Lambda \vec{\alpha}.\, E$. By hypothesis, $\Gamma, x : S' \vdash E : \tau$ where $\forall \vec{\alpha}.\tau \leq S$. By induction hypothesis, we deduce $\Gamma \vdash E[E'/x] : \tau$. Hence, rule $[\mathrm{T}_{\mathsf{TAbstr}}]$ yields $\Gamma \vdash \Lambda \vec{\alpha}.\, E[E'/x] : \forall \vec{\alpha}.\tau$, hence the result.

- $E\ [\vec{t}]$. By hypothesis, $\Gamma, x : S' \vdash E : \forall\vec{\alpha}.\tau$, and $\tau[\vec{t}/\vec{\alpha}] \leq S$. The induction hypothesis then yields $\Gamma \vdash E[E'/x] : \forall\vec{\alpha}.\tau$. Applying rule $[\mathrm{T}_{\mathsf{TApp}}]$ yields $\Gamma \vdash (E[E'/x])\ [\vec{t}] : \tau[\vec{t}/\vec{\alpha}]$, hence the result.
- $E\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$. By hypothesis, $\Gamma, x : S' \vdash E : \tau_1$, and $\tau_2 \leq S$. The induction hypothesis then yields $\Gamma \vdash E[E'/x] : \tau_1$. Applying rule $[\mathrm{T}_{\mathsf{Cast}}]$ gives $\Gamma \vdash (E[E'/x])\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$ and the result follows.

□

Lemma B.48. *If $\Gamma \vdash E : S$ and $\Gamma \vdash \mathscr{E}[E] : S'$ then for every expression E' such that $\Gamma \vdash E' : S$, we have $\Gamma \vdash \mathscr{E}[E'] : S'$.*

Proof. By complete induction over $\mathscr{E}$.

- □. Immediate with $S = S'$.
- $E_f\ \mathscr{E}$. By hypothesis and inversion of rule $[\mathrm{T}_{\mathsf{App}}]$, we have $\Gamma \vdash E_f : \tau \to S'$ and $\Gamma \vdash \mathscr{E}[E] : \tau$. By induction hypothesis, it holds that $\Gamma \vdash \mathscr{E}[E'] : \tau$. Therefore, by $[\mathrm{T}_{\mathsf{App}}]$, $\Gamma \vdash E_f\ \mathscr{E}[E'] : S'$.
- $\mathscr{E}\ V$. By hypothesis and inversion of rule $[\mathrm{T}_{\mathsf{App}}]$, we have $\Gamma \vdash \mathscr{E}[E] : \tau \to S'$ and $\Gamma \vdash V : \tau$. By induction hypothesis, it holds that $\Gamma \vdash \mathscr{E}[E'] : \tau \to S'$. Therefore, by $[\mathrm{T}_{\mathsf{App}}]$, $\Gamma \vdash \mathscr{E}[E']\ V : S'$.
- $\mathscr{E}\ [\vec{t}]$. By hypothesis and inversion of $[\mathrm{T}_{\mathsf{TApp}}]$, we have $\Gamma \vdash \mathscr{E}[E] : \forall\vec{\alpha}.\tau$ where $\tau[\vec{t}/\vec{\alpha}] \leq S'$. By IH, it holds that $\Gamma \vdash \mathscr{E}[E'] : \forall\vec{\alpha}.\tau$. Therefore, by rule $[\mathrm{T}_{\mathsf{TApp}}]$, we have $\Gamma \vdash \mathscr{E}[E']\ [\vec{t}] : \tau[\vec{t}/\vec{\alpha}]$ and the result follows by $[\mathrm{T}_{\mathsf{Subsume}}]$.
- $(E_l, \mathscr{E})$. By hypothesis and inversion of $[\mathrm{T}_{\mathsf{Pair}}]$, we have $\Gamma \vdash E_l : \tau_1$ and $\Gamma \vdash \mathscr{E}[E] : \tau_2$ where $\tau_1 \times \tau_2 \leq S'$. By IH, we deduce $\Gamma \vdash \mathscr{E}[E'] : \tau_2$. Therefore, it holds by rule $[\mathrm{T}_{\mathsf{Pair}}]$ that $\Gamma \vdash (E_l, \mathscr{E}[E']) : \tau_1 \times \tau_2$ and the result follows by rule $[\mathrm{T}_{\mathsf{Subsume}}]$.
- $(\mathscr{E}, V)$. By hypothesis and inversion of $[\mathrm{T}_{\mathsf{Pair}}]$, we have $\Gamma \vdash \mathscr{E}[E] : \tau_1$ and $\Gamma \vdash V : \tau_2$ where $\tau_1 \times \tau_2 \leq S'$. By IH, we deduce $\Gamma \vdash \mathscr{E}[E'] : \tau_1$. Therefore, it holds by rule $[\mathrm{T}_{\mathsf{Pair}}]$ that $\Gamma \vdash (\mathscr{E}[E'], V) : \tau_1 \times \tau_2$ and the result follows by rule $[\mathrm{T}_{\mathsf{Subsume}}]$.
- $\pi_i\ \mathscr{E}$. By hypothesis and inversion of $[\mathrm{T}_{\mathsf{Proj}}]$, we have $\Gamma \vdash \mathscr{E}[E] : \tau_1 \times \tau_2$ where $\tau_i \leq S'$. By IH, we deduce $\Gamma \vdash \mathscr{E}[E'] : \tau_1 \times \tau_2$ thus $[\mathrm{T}_{\mathsf{Proj}}]$ yields that $\Gamma \vdash \pi_i\ (\mathscr{E}[E']) : \tau_i$, and the result follows by subsumption.
- $\mathsf{let}\ x = \mathscr{E}\ \mathsf{in}\ E_l$. By hypothesis and inversion of $[\mathrm{T}_{\mathsf{Let}}]$, we have $\Gamma \vdash \mathscr{E}[E] : \forall\vec{\alpha}.\tau$ and $\Gamma, x : \forall\vec{\alpha}.\tau \vdash E_l : S'$. By IH, we deduce $\Gamma \vdash \mathscr{E}[E'] : \forall\vec{\alpha}.\tau$. Therefore, it holds by rule $[\mathrm{T}_{\mathsf{Let}}]$ that $\Gamma \vdash \mathsf{let}\ x = \mathscr{E}[E']\ \mathsf{in}\ E_l : S'$.
- $\mathscr{E}\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$. By hypothesis and inversion of $[\mathrm{T}_{\mathsf{Cast}}]$, we have $\Gamma \vdash \mathscr{E}[E] : \tau_1$ and $\tau_2 \leq S'$. By IH, it holds that $\Gamma \vdash \mathscr{E}[E'] : \tau_1$. Therefore, by rule $[\mathrm{T}_{\mathsf{Cast}}]$, we have $\Gamma \vdash \mathscr{E}[E']\langle\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$, and the result follows by $[\mathrm{T}_{\mathsf{Subsume}}]$.

□

Lemma B.49. *If $\Gamma \vdash E : \tau$, then for every type substitution θ, $\Gamma\theta \vdash E\theta : \tau\theta$.*

Proof. By induction on the derivation of $\Gamma \vdash E : \tau$ and by case on the last rule applied.

- $[\mathrm{T}_{\mathsf{Var}}]$. We have $\Gamma \vdash x\colon \forall\vec{\alpha}.\,\tau$ and $\Gamma(x) = \forall\vec{\alpha}.\,\tau$. We deduce $(\Gamma\theta)(x) = \forall\vec{\alpha}.\,\tau\theta$. Since $x\theta = x$, we apply $[\mathrm{T}_{\mathsf{Var}}]$ to deduce the result: $\Gamma\theta \vdash x : \forall\vec{\alpha}.\,\tau\theta$.
- $[\mathrm{T}_{\mathsf{Const}}]$. Immediate since $b_c\theta = b_c$.
- $[\mathrm{T}_{\mathsf{Abstr}}]$, $[\mathrm{T}_{\mathsf{App}}]$, $[\mathrm{T}_{\mathsf{Pair}}]$, $[\mathrm{T}_{\mathsf{Proj}}]$, $[\mathrm{T}_{\mathsf{TAbstr}}]$, $[\mathrm{T}_{\mathsf{TApp}}]$, $[\mathrm{T}_{\mathsf{Let}}]$. Direct application of the induction hypothesis.
- $[\mathrm{T}_{\mathsf{Subsume}}]$. By Proposition 5.19, $\tau' \leq \tau$ implies $\tau'\theta \leq \tau\theta$ for any static type substitution θ, and the result follows.
- $[\mathrm{T}_{\mathsf{Cast}}]$. By Proposition 5.3, $\tau' \preccurlyeq \tau$ implies $\tau'\theta \preccurlyeq \tau\theta$ for any type substitution θ, and the result follows.

□

Lemma B.50 (Subject Reduction)**.** *For every terms E, E' and every context Γ, if $\Gamma \vdash E : S$ and $E \hookrightarrow E'$ then $\Gamma \vdash E' : S$.*

Proof. By case disjunction over the rule used in the reduction $E \hookrightarrow E'$.

- $[\mathrm{R}_{\mathsf{ExpandL}}]$ $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_1/\tau_2\rangle\langle \tau_1/\tau_2 \overset{p}{\Rightarrow} \tau_2\rangle$. By inversion of the typing rules, $\tau_2 \leq S$. By hypothesis of the reduction rule, $\tau_2 \preccurlyeq \tau_1$. By inversion of the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$, we deduce that $\Gamma \vdash V : \tau_1$ and $p = l$. By Proposition B.20, we have $\tau_2 \preccurlyeq \tau_1/\tau_2 \preccurlyeq \tau_1$. Therefore, applying the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$ twice yields $\Gamma \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_1/\tau_2\rangle : \tau_1/\tau_2$ and then $\Gamma \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_1/\tau_2\rangle\langle \tau_1/\tau_2 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$. Since $\tau_2 \leq S$, applying $[\mathrm{T}_{\mathsf{Subsume}}]$ yields the result.
- $[\mathrm{R}_{\mathsf{ExpandR}}]$. $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2/\tau_1\rangle\langle \tau_2/\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$. By inversion of the typing rules, $\tau_2 \leq S$. By hypothesis of the reduction rule, $\tau_1 \preccurlyeq \tau_2$. By inversion of the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$, we deduce that $\Gamma \vdash V : \tau_1$ and $p = l$. By Proposition B.20, we have $\tau_1 \preccurlyeq \tau_2/\tau_1 \preccurlyeq \tau_2$. Therefore, applying the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$ twice yields $\Gamma \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2/\tau_1\rangle : \tau_2/\tau_1$ and then $\Gamma \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2/\tau_1\rangle\langle \tau_2/\tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$. Since $\tau_2 \leq S$, applying $[\mathrm{T}_{\mathsf{Subsume}}]$ yields the result.
- $[\mathrm{R}_{\mathsf{CastId}}]$ $V\langle \tau \overset{p}{\Rightarrow} \tau\rangle \hookrightarrow V$. By inversion of the typing rules, $\tau \leq S$. By inversion of the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$, $\Gamma \vdash V : \tau$. Applying $[\mathrm{T}_{\mathsf{Subsume}}]$ yields $\Gamma \vdash V : S$.
- $[\mathrm{R}_{\mathsf{Collapse}}]$ $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle \hookrightarrow V$. By inversion of the typing rules, $\tau_2' \leq S$. Inversing the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$ twice yields $\Gamma \vdash V : \tau_1$. Since, by hypothesis of $[\mathrm{R}_{\mathsf{Collapse}}]$, $\tau_1 \leq \tau_2' \leq S$, applying $[\mathrm{T}_{\mathsf{Subsume}}]$ yields $\Gamma \vdash V : S$.
- $[\mathrm{R}_{\mathsf{UpSimpl}}]$ $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle$. By inversion of the typing rules, $\tau_2' \leq S$. Inversing the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$ yields $\Gamma \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : \tau_2$. By hypothesis of the reduction rule, $\tau_2 \leq \tau_2' \leq S$. Therefore, applying $[\mathrm{T}_{\mathsf{Subsume}}]$ yields $\Gamma \vdash V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle : S$.
- $[\mathrm{R}_{\mathsf{UnboxSimpl}}]$ $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow V$. By inversion of the typing rules, $\tau_2 \leq S$. By Lemma B.3, $\Gamma \vdash V : \mathrm{type}(V)$. By hypothesis of $[\mathrm{R}_{\mathsf{UnboxSimpl}}]$, $\mathrm{type}(V) \leq \tau_2 \leq S$, thus applying $[\mathrm{T}_{\mathsf{Subsume}}]$ yields $\Gamma \vdash V : S$.

- $[\mathrm{R}_{\mathsf{CastApp}}]$ $V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle V' \hookrightarrow (V\ V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1 \rangle)\langle \tau_2 \overset{p}{\Rightarrow} \tau_2' \rangle$ where $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \mathrm{type}(V') = \langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle$.
  First of all, Lemma B.37 ensures that the casts $\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1 \rangle$ and $\langle \tau_2 \overset{p}{\Rightarrow} \tau_2' \rangle$ are well-formed and respect the precision conditions present in the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$. Lemma B.39 then ensures $\mathrm{type}(V') \leq \tau_1'$, thus $\Gamma \vdash V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1 \rangle : \tau_1$ by rule $[\mathrm{T}_{\mathsf{Cast}}]$.
  Inversing the typing rules yields $\Gamma \vdash V : \tau$ and Lemma B.39 gives $\tau \leq \tau_1 \to \tau_2$. Therefore $\Gamma \vdash V : \tau_1 \to \tau_2$ by rule $[\mathrm{T}_{\mathsf{Subsume}}]$, and $\Gamma \vdash V\ V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1 \rangle : \tau_2$ by rule $[\mathrm{T}_{\mathsf{App}}]$.
  Finally, we obtain $\Gamma \vdash (V\ V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1 \rangle)\langle \tau_2 \overset{p}{\Rightarrow} \tau_2' \rangle : \tau_2'$ by rule $[\mathrm{T}_{\mathsf{Cast}}]$. The last thing we need to prove is $\tau_2' \leq S$. By inversing the typing rules, and using Lemma B.3, we have $\tau' \leq \mathrm{type}(V') \to S$. The result follows by applying Lemma B.39.

- $[\mathrm{R}_{\mathsf{CastProj}}]$ $\pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) \hookrightarrow (\pi_i\ V)\langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle$ where $\pi_i\ (\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) = \langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle$. First of all, Lemma B.41 ensures that the cast $\langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle$ is well-formed and respect the precision conditions present in the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$.
  Now consider $i = 1$ (the case $i = 2$ is proved in the same way). Lemma B.43 then ensures $\tau \leq (\tau_i \times \mathbb{1})$. And, by hypothesis and inversion of the typing rules, we know that $\Gamma \vdash V : \tau$. Therefore, by rule $[\mathrm{T}_{\mathsf{Subsume}}]$, we have $\Gamma \vdash V : \tau_i \times \mathbb{1}$. Then, by rule $[\mathrm{T}_{\mathsf{Proj}}]$, we deduce $\Gamma \vdash \pi_1\ V : \tau_i$. Finally, the rule $[\mathrm{T}_{\mathsf{Cast}}]$ allows us to conclude that $\Gamma \vdash (\pi_1\ V)\langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle : \tau_i'$.
  Now, by hypothesis, we have $\Gamma \vdash \pi_1\ (V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) : S$. Thus, by inversion of the typing rules and subsumption, we deduce $\Gamma \vdash V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle : (S \times \mathbb{1})$. That is, by inversion of $[\mathrm{T}_{\mathsf{Cast}}]$, $\tau' \leq S \times \mathbb{1}$. Now, the second part of Lemma B.43 yields $\tau_i' = \min\{\tau_0 \mid \tau' \leq \tau_0 \times \mathbb{1}\}$. From this, we can deduce $\tau_i' \leq S$, and we conclude that $\Gamma \vdash (\pi_1\ V)\langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle : S$ by subsumption.

- $[\mathrm{R}_{\mathsf{SimplApp}}]$ $V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle V' \hookrightarrow V\ V'$. By inversion of the typing rules, and using Lemma B.3, we have $\tau' \leq \mathrm{type}(V') \to S$. By hypothesis of the reduction rule, we also have $\tau / \tau' = \tau$. Applying Corollary B.24 therefore yields $\tau \leq \mathrm{type}(V') \to S$. Since $\Gamma \vdash V : \tau$ by inversion of the typing rules, we deduce that $\Gamma \vdash V : \mathrm{type}(V') \to S$ by rule $[\mathrm{T}_{\mathsf{Subsume}}]$. We can then conclude by applying rule $[\mathrm{T}_{\mathsf{App}}]$.

- $[\mathrm{R}_{\mathsf{SimplProj}}]$ $\pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) \hookrightarrow \pi_i\ V$. By inversion of the typing rules, we have $\tau' \leq \tau_1 \times \tau_2$ where $\tau_i \leq S$. By hypothesis of the reduction rule, we also have $\tau / \tau' = \tau$. Applying Corollary B.24 therefore yields $\tau \leq \tau_1 \times \tau_2$. Since $\Gamma \vdash V : \tau$ by inversion of the typing rules, we deduce that $\Gamma \vdash V : \tau_1 \times \tau_2$ by rule $[\mathrm{T}_{\mathsf{Subsume}}]$. We can then conclude by applying rule $[\mathrm{T}_{\mathsf{Proj}}]$ and $[\mathrm{T}_{\mathsf{Subsume}}]$.

- $[\mathrm{R}_{\mathsf{App}}]$ $(\lambda^{\tau_1 \to \tau_2} x.\, E)V \hookrightarrow E[V/x]$. By inversion of the typing rules, we have $\Gamma \vdash V : \tau_1$, $\tau_2 \leq S$, as well as $\Gamma, x : \tau_1 \vdash E : \tau_2$. Lemma B.47 immediately yields that $\Gamma \vdash E[V/x] : \tau_2$, and the result follows by $[\mathrm{T}_{\mathsf{Subsume}}]$.

- $[\mathrm{R}_{\mathsf{Proj}}]$ $\pi_i\ (V_1, V_2) \hookrightarrow V_i$. By inversion of the typing rules, we have $\Gamma \vdash (V_1, V_2) : \tau_1 \times \tau_2$ and $\tau_i \leq S$. Inversing the typing rules a second time yields $\Gamma \vdash V_1 : \tau_1$ and $\Gamma \vdash V_2 : \tau_2$, therefore, by $[\mathrm{T}_{\mathsf{Subsume}}]$ we obtain $\Gamma \vdash V_i : S$.

- $[\mathrm{R}_{\mathsf{TypeApp}}]$ $(\Lambda \vec{\alpha}.\, E)\ [\vec{t}] \hookrightarrow E[\vec{t}/\vec{\alpha}]$. We have, by hypothesis, $\Gamma \vdash \Lambda \vec{\alpha}.\, E : \forall \vec{\alpha}.\tau$ where

$\Gamma \vdash E : \tau$ and $\tau[\vec{t}/\vec{\alpha}] \leq S$. Applying Lemma B.49 yields $\Gamma[\vec{t}/\vec{\alpha}] \vdash E[\vec{t}/\vec{\alpha}] : \tau[\vec{t}/\vec{\alpha}]$. However, by hypothesis of the typing rules, $\vec{\alpha} \sharp \Gamma$. Therefore, $\Gamma[\vec{t}/\vec{\alpha}] = \Gamma$, and we have $\Gamma \vdash E[\vec{t}/\vec{\alpha}] : \tau[\vec{t}/\vec{\alpha}]$, which is the result.

- $[\mathrm{R}_{\mathsf{Let}}]$ $\mathsf{let}\ x = V\ \mathsf{in}\ E \hookrightarrow E[V/x]$. By hypothesis, we have $\Gamma \vdash V : \forall\vec{\alpha}.\tau$, and $\Gamma, x : \forall\vec{\alpha}.\tau \vdash E : \tau'$ where $\tau' \leq S$. Lemma B.47 immediately yields that $\Gamma \vdash E[V/x] : \tau'$, and the result follows by $[\mathrm{T}_{\mathsf{Subsume}}]$.
- $[\mathrm{R}_{\mathsf{Context}}]$ $\mathscr{E}[E] \hookrightarrow \mathscr{E}[E']$ where $E \hookrightarrow E'$. Immediate by Lemma B.48.

□

Theorem B.51 (Soundness)**.** *For every term E such that $\emptyset \vdash E : S$, one of the following cases holds:*

- *there exists a value V such that $E \hookrightarrow^* V$*
- *there exists a label p such that $E \hookrightarrow^* \mathtt{blame}\ p$*
- *E diverges*

Proof. Immediate corollary of Lemma B.50 and Lemma B.46. □

Theorem B.52 (Blame Safety)**.** *For every term E such that $\emptyset \vdash E : S$, and every blame label l, $E \not\hookrightarrow^* \mathtt{blame}\ \bar{l}$.*

Proof. Given Lemma B.50, and by induction over E, it is sufficient to prove the result for reductions of length one. The proof is then done by case disjunction over the reduction rules that can produce a blame.

- $[\mathrm{R}_{\mathsf{Blame}}]$ $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle \hookrightarrow \mathtt{blame}\ q$. By hypothesis of the reduction rule, $\tau_1' \preccurlyeq \tau_2'$. Thus, by inversion of the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$, q is a positive label.
- $[\mathrm{R}_{\mathsf{UpBlame}}]$ $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle\langle \tau_1' \overset{q}{\Rightarrow} \tau_2'\rangle \hookrightarrow \mathtt{blame}\ q$. By hypothesis of the reduction rule, $\tau_1' \preccurlyeq \tau_2'$. Thus, by inversion of the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$, q is a positive label.
- $[\mathrm{R}_{\mathsf{UnboxBlame}}]$ $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2\rangle \hookrightarrow \mathtt{blame}\ p$. By hypothesis of the reduction rule, $\tau_1 \preccurlyeq \tau_2$. Thus, by inversion of the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$, p is a positive label.
- $[\mathrm{R}_{\mathsf{FailApp}}]$ $V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle V' \hookrightarrow \mathtt{blame}\ p$. By hypothesis of the reduction rule, the two possible cases are $\tau'/\tau = \tau$ and $\tau/\tau' = \tau'$. Moreover, by inversion of the typing rules, $\tau' \leq \mathbb{0} \to \mathbb{1}$. Thus, by contradiction, if $\tau/\tau' = \tau'$, then $\langle \tau \overset{p}{\Rightarrow} \tau'\rangle \circ \mathrm{type}(V')$ would be well-defined according to Lemma B.38. Therefore, we necessarily have $\tau'/\tau = \tau$, and by inversion of the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$, p is a positive label.
- $[\mathrm{R}_{\mathsf{FailProj}}]$ $\pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau'\rangle) \hookrightarrow \mathtt{blame}\ p$. By hypothesis of the reduction rule, the two possible cases are $\tau'/\tau = \tau$ and $\tau/\tau' = \tau'$. Moreover, by inversion of the typing rules, $\tau' \leq \mathbb{1} \to \mathbb{1}$. Thus, by contradiction, if $\tau/\tau' = \tau'$, then $\pi_i\ (\langle \tau \overset{p}{\Rightarrow} \tau'\rangle)$ would be well-defined according to Lemma B.42. Therefore, we necessarily have $\tau'/\tau = \tau$, and by

inversion of the typing rule $[\mathrm{T}_{\mathsf{Cast}}]$, p is a positive label.

- $[\mathrm{R}_{\mathsf{CtxBlame}}]$ $\mathscr{E}[E] \hookrightarrow \texttt{blame}\ p$. Immediate by induction.

□

Theorem B.53 (Conservativity). *For every term E such that $\emptyset \vdash_{\mathsf{Sub}} E : \tau$, $E \rightsquigarrow_{\mathsf{HM}} E' \iff E \rightsquigarrow_{\mathsf{ST}} E'$ and $E \rightsquigarrow_{\mathsf{HM}} \texttt{blame}\ p \iff E \rightsquigarrow_{\mathsf{ST}} \texttt{blame}\ p$.*

Proof. By cases over the rule used in the reduction of E, and induction on E.

1. First implication, (Sub) $\implies$ (Set).
   - $[\mathrm{R}_{\mathsf{ExpandL}}]$ $V\langle \tau \overset{p}{\Rightarrow} ? \rangle \rightsquigarrow_{\mathsf{HM}} V\langle \tau \overset{p}{\Rightarrow} \tau/? \rangle \langle \tau/? \overset{p}{\Rightarrow} ? \rangle$. By hypothesis of the reduction rule, $\tau/? \neq \tau$, and $\tau \neq ?$. Therefore, $\tau/? \neq ?$ (since only $?/? = ?$). Thus, the rule $[\mathrm{R}_{\mathsf{ExpandL}}]$ can be applied in Set to yield $V\langle \tau \overset{p}{\Rightarrow} ? \rangle \rightsquigarrow_{\mathsf{ST}} V\langle \tau \overset{p}{\Rightarrow} \tau/? \rangle \langle \tau/? \overset{p}{\Rightarrow} ? \rangle$.
   - $[\mathrm{R}_{\mathsf{ExpandR}}]$ $V\langle ? \overset{p}{\Rightarrow} \tau \rangle \rightsquigarrow_{\mathsf{HM}} V\langle ? \overset{p}{\Rightarrow} \tau/? \rangle \langle \tau/? \overset{p}{\Rightarrow} \tau \rangle$. By hypothesis of the reduction rule, $\tau/? \neq \tau$, and $\tau \neq ?$. Therefore, $\tau/? \neq ?$ for the same reason as before. Thus, rule $[\mathrm{R}_{\mathsf{ExpandR}}]$ can be applied in Set to yield $V\langle ? \overset{p}{\Rightarrow} \tau \rangle \rightsquigarrow_{\mathsf{HM}} V\langle ? \overset{p}{\Rightarrow} \tau/? \rangle \langle \tau/? \overset{p}{\Rightarrow} \tau \rangle$.
   - $[\mathrm{R}_{\mathsf{CastId}}]$ $V\langle \tau \overset{p}{\Rightarrow} \tau \rangle \rightsquigarrow_{\mathsf{HM}} V$. Immediate since $[\mathrm{R}_{\mathsf{CastId}}]$ is unchanged in Set.
   - $[\mathrm{R}_{\mathsf{Collapse}}]$ $V\langle \rho \overset{p}{\Rightarrow} ? \rangle \langle ? \overset{q}{\Rightarrow} \rho' \rangle \rightsquigarrow_{\mathsf{HM}} V$. By hypothesis, $\rho \leq \rho'$. Moreover, by definition of ground types, we have $\rho/? = \rho$ and $\rho'/? = \rho'$. All the hypothesis of rule $[\mathrm{R}_{\mathsf{Collapse}}]$ in Set are therefore valid, and the rule can be applied to deduce $V\langle \rho \overset{p}{\Rightarrow} ? \rangle \langle ? \overset{q}{\Rightarrow} \rho' \rangle \rightsquigarrow_{\mathsf{ST}} V$.
   - $[\mathrm{R}_{\mathsf{Blame}}]$ $V\langle \rho \overset{p}{\Rightarrow} ? \rangle \langle ? \overset{q}{\Rightarrow} \rho' \rangle \rightsquigarrow_{\mathsf{HM}} \texttt{blame}\ q$. We have the same hypothesis as before except $\rho \not\leq \rho'$. Therefore, we can apply rule $[\mathrm{R}_{\mathsf{Blame}}]$ in Set to deduce $V\langle \rho \overset{p}{\Rightarrow} ? \rangle \langle ? \overset{q}{\Rightarrow} \rho' \rangle \rightsquigarrow_{\mathsf{ST}} \texttt{blame}\ q$.
   - $[\mathrm{R}_{\mathsf{CastApp}}]$ $V\langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle V' \rightsquigarrow_{\mathsf{HM}} V(V'\langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1 \rangle)\langle \tau_2 \overset{p}{\Rightarrow} \tau_2' \rangle$. We pose $\tau = \tau_1 \to \tau_2$ and $\tau' = \tau_1' \to \tau_2'$. τ and τ' are trivially in disjunctive normal form, and both are not empty (since an arrow cannot be empty). Thus, the cast $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \mathrm{type}(V')$ is well-defined (satisfies the conditions of Definition B.36). Moreover, by hypothesis, we know that either $\tau \preccurlyeq \tau'$ or $\tau' \preccurlyeq \tau$. By definition of the grounding operator, we then either have $\tau'/\tau = \tau$ or $\tau/\tau' = \tau'$. Thus, all the hypothesis of the rule $[\mathrm{R}_{\mathsf{CastApp}}]$ in Set are valid.
     Finally, by inversion of the typing rule $[\mathrm{T}_{\mathsf{App}}]$, we deduce that $\mathrm{type}(V') \leq \tau_1'$. A simple application of Definition B.36 (case were I and P_i are singletons) shows that $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \mathrm{type}(V') = \langle \tau \overset{p}{\Rightarrow} \tau' \rangle$, hence the result.
   - $[\mathrm{R}_{\mathsf{App}}]$ $(\lambda^{\tau_1 \to \tau_2} x.\, E)V \rightsquigarrow_{\mathsf{HM}} E[V/x]$. Immediate since $[\mathrm{R}_{\mathsf{App}}]$ is unchanged in Set.
   - $[\mathrm{R}_{\mathsf{ProjCast}}]$ $\pi_i\ (V\langle \tau_1 \times \tau_2 \overset{p}{\Rightarrow} \tau_1' \times \tau_2' \rangle) \rightsquigarrow_{\mathsf{HM}} (\pi_i\ V)\langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle$. Let $\tau = \tau_1 \times \tau_2$ and $\tau' = \tau_1' \times \tau_2'$. τ and τ' are trivially in disjunctive normal form, and both are not empty (since a product cannot be empty in Sub, as both sides cannot be empty). Thus, the cast $\pi_i\ (\langle \tau \overset{p}{\Rightarrow} \tau' \rangle)$ is well-defined as it satisfies all the conditions of

Definition B.40. Moreover, by hypothesis (inversion of typing rule $[\mathrm{T}_{\mathsf{Cast}}]$), we know that either $\tau \leqslant \tau'$ or $\tau' \leqslant \tau$. By definition of the grounding operator, this yields that either $\tau'/\tau = \tau$ or $\tau/\tau' = \tau'$ respectively. Thus, all the hypothesis of $[\mathrm{R}_{\mathsf{CastProj}}]$ in Set are verified.
Finally, a simple application of Definition B.40 (case were I and P_i are singletons, and $N_i = \emptyset$) shows that $\pi_i\ (\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) = \langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle$, hence the result.

- $[\mathrm{R}_{\mathsf{Proj}}]\ \pi_i\ (V_1, V_2) \rightsquigarrow_{\mathsf{HM}} V_i$. Immediate since $[\mathrm{R}_{\mathsf{Proj}}]$ is unchanged in Set.
- $[\mathrm{R}_{\mathsf{TypeApp}}]\ (\Lambda\vec{\alpha}.\, E)\ [\vec{t}] \rightsquigarrow_{\mathsf{HM}} E[\vec{t}/\vec{\alpha}]$. Immediate since $[\mathrm{R}_{\mathsf{TypeApp}}]$ is unchanged in Set.
- $[\mathrm{R}_{\mathsf{Let}}]\ \mathsf{let}\ x = V\ \mathsf{in}\ E \rightsquigarrow_{\mathsf{HM}} E[V/x]$. Immediate since $[\mathrm{R}_{\mathsf{Let}}]$ is unchanged in Set.
- $[\mathrm{R}_{\mathsf{Context}}]\ \mathscr{E}[E] \rightsquigarrow_{\mathsf{HM}} \mathscr{E}[E']$ where $E \rightsquigarrow_{\mathsf{HM}} E'$. By induction hypothesis, $E \rightsquigarrow_{\mathsf{ST}} E'$. Thus, by rule $[\mathrm{R}_{\mathsf{Context}}]$ in the Set system, $\mathscr{E}[E] \rightsquigarrow_{\mathsf{ST}} \mathscr{E}[E']$.
- $[\mathrm{R}_{\mathsf{CtxBlame}}]\ \mathscr{E}[E] \rightsquigarrow_{\mathsf{HM}} \mathtt{blame}\ p$ where $E \rightsquigarrow_{\mathsf{HM}} \mathtt{blame}\ p$. By induction hypothesis, $E \rightsquigarrow_{\mathsf{ST}} \mathtt{blame}\ p$. Thus, by rule $[\mathrm{R}_{\mathsf{CtxBlame}}]$ in the Set system, $\mathscr{E}[E] \rightsquigarrow_{\mathsf{ST}} \mathtt{blame}\ p$.

2. Second implication, (Set) $\implies$ (Sub). We omit the trivial cases where the same rule is present in both systems.
   - $[\mathrm{R}_{\mathsf{ExpandL}}]\ V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_1/\tau_2 \rangle \langle \tau_1/\tau_2 \overset{p}{\Rightarrow} \tau_2 \rangle$. By hypothesis of the reduction rule, $\tau_2 \leqslant \tau_1$ and $\tau_1/\tau_2 \neq \tau_2$. Therefore, by Lemma B.21, we deduce that $\tau_2 = ?$. Since $\tau_1/\tau_2 \neq \tau_2$, we have $\tau_1 \neq ?$, and by hypothesis of $[\mathrm{R}_{\mathsf{ExpandL}}]$ $\tau_1/\tau_2 \neq \tau_1$ therefore all the conditions of $[\mathrm{R}_{\mathsf{ExpandL}}]$ in Sub are verified, and the result follows.
   - $[\mathrm{R}_{\mathsf{ExpandR}}]\ V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2/\tau_1 \rangle \langle \tau_2/\tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$. By hypothesis of the reduction rule, $\tau_1 \leqslant \tau_2$ and $\tau_2/\tau_1 \neq \tau_1$. Therefore, by Lemma B.21, we deduce that $\tau_1 = ?$. Since $\tau_2/\tau_1 \neq \tau_1$, we have $\tau_2 \neq ?$, and by hypothesis of $[\mathrm{R}_{\mathsf{ExpandR}}]$ $\tau_2/\tau_1 \neq \tau_2$ therefore all the conditions of $[\mathrm{R}_{\mathsf{ExpandR}}]$ in Sub are verified, and the result follows.
   - $[\mathrm{R}_{\mathsf{Collapse}}]\ V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \langle \tau_1' \overset{q}{\Rightarrow} \tau_2' \rangle \hookrightarrow V$. By hypothesis of the reduction rule, $\tau_2'/\tau_1' = \tau_2'$ and $\tau_2' \neq \tau_1'$ (by precedence of $[\mathrm{R}_{\mathsf{CastId}}]$). Thus, by Lemma B.22, we have $\tau_1' = ?$. By typing hypothesis, we also have $\tau_2 \leq \tau_1'$. By definition of subtyping on non-set-theoretic types, we deduce $\tau_2 = ?$. And by hypothesis of the reduction rule, we finally have $\tau_1 \leq \tau_2'$ and $\tau_1/\tau_2 = \tau_1$ (the case $\tau_2/\tau_1 = \tau_1$ being only possible if $\tau_2 = \tau_1 = ?$). Thus, all the conditions for the rule $[\mathrm{R}_{\mathsf{Collapse}}]$ in Sub are verified, and the result follows.
   - $[\mathrm{R}_{\mathsf{Blame}}]\ V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \langle \tau_1' \overset{q}{\Rightarrow} \tau_2' \rangle \hookrightarrow \mathtt{blame}\ q$. Same reasoning as before, except this time $\tau_1 \not\leq \tau_2'$ which allows us to apply rule $[\mathrm{R}_{\mathsf{Blame}}]$ in Sub.
   - $[\mathrm{R}_{\mathsf{UpSimpl}}]\ V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \langle \tau_1' \overset{q}{\Rightarrow} \tau_2' \rangle \hookrightarrow V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle$. By hypothesis, $\tau_2'/\tau_1' = \tau_2'$ and $\tau_1' \neq \tau_2'$. By Lemma B.22, $\tau_1' = ?$. Moreover, by typing hypothesis, $\tau_2 \leq \tau_1'$ thus $\tau_2 = ?$ by definition of subtyping. By hypothesis, $\tau_1/\tau_2 = \tau_2$ but since $\tau_2 = ?$, we necessarily have $\tau_1 = ?$ by Definition B.1. Thus, we have a contradiction since $\tau_1 \neq \tau_2$ by hypothesis, and this rule cannot be applied.

- $[\mathrm{R}_{\mathsf{UpBlame}}]$ $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \langle \tau_1' \overset{q}{\Rightarrow} \tau_2' \rangle \hookrightarrow \texttt{blame}\ q$. Same reasoning as before, this rule cannot be applied.
- $[\mathrm{R}_{\mathsf{UnboxSimpl}}]$ $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \hookrightarrow V$. By hypothesis, $\tau_2 / \tau_1 = \tau_2$ and $\tau_1 \neq \tau_2$. Applying Lemma B.22 yields $\tau_1 = ?$. However, by hypothesis, $\emptyset \vdash V : \tau_1$. A simple case disjunction on V shows that this cannot hold, thus we have a contradiction and this rule cannot be applied.
- $[\mathrm{R}_{\mathsf{UnboxBlame}}]$ $V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \hookrightarrow \texttt{blame}\ p$. Same reasoning as before, this rule cannot be applied.
- $[\mathrm{R}_{\mathsf{CastApp}}]$ $V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle V' \hookrightarrow (V V' \langle \tau_1' \overset{\bar{p}}{\Rightarrow} \tau_1 \rangle) \langle \tau_2 \overset{p}{\Rightarrow} \tau_2' \rangle$ where $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \mathrm{type}(V') = \langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle$. By hypothesis and inversion of rule $[\mathrm{T}_{\mathsf{App}}]$, $\tau' \leq \mathbb{0} \to \mathbb{1}$. Since τ' does not contain connectives, $\tau' = \sigma_1' \to \sigma_2'$ for some σ_1', σ_2'. Moreover, by hypothesis of the reduction we either have $\tau / \tau' = \tau'$ or $\tau' / \tau = \tau$, thus necessarily $\tau = \sigma_1 \to \sigma_2$ for some σ_1, σ_2 by Definition B.1. Moreover, by hypothesis, we have $\mathrm{type}(V') \leq \sigma_1'$. A simple application of Definition B.36 then yields $\langle \tau \overset{p}{\Rightarrow} \tau' \rangle \circ \mathrm{type}(V') = \langle \sigma_1 \to \sigma_2 \overset{p}{\Rightarrow} \sigma_1' \to \sigma_2' \rangle = \langle \tau_1 \to \tau_2 \overset{p}{\Rightarrow} \tau_1' \to \tau_2' \rangle$. Applying rule $[\mathrm{T}_{\mathsf{App}}]$ in Sub yields $V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle V' \hookrightarrow (V\ V' \langle \sigma_1' \overset{\bar{p}}{\Rightarrow} \sigma_1 \rangle) \langle \sigma_2 \overset{p}{\Rightarrow} \sigma_2' \rangle$, hence the result.
- $[\mathrm{R}_{\mathsf{CastProj}}]$ $\pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) \hookrightarrow (\pi_i\ V) \langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle$ where $\langle \tau_i \overset{p}{\Rightarrow} \tau_i' \rangle = \pi_i\ (\langle \tau \overset{p}{\Rightarrow} \tau' \rangle)$. Same reasoning with product types and Definition B.40.
- $[\mathrm{R}_{\mathsf{FailApp}}]$ $V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle V' \hookrightarrow \texttt{blame}\ p$. Using the same reasoning as for $[\mathrm{R}_{\mathsf{CastApp}}]$, we deduce $\tau = \tau_1 \to \tau_2$ and $\tau' = \tau_1' \to \tau_2'$. In particular, both τ and τ' are trivially in disjunctive normal form and are non-empty, and thus verify all the conditions of Definition B.36. Therefore, this rule cannot be applied.
- $[\mathrm{R}_{\mathsf{FailProj}}]$ $\pi_i (V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) \hookrightarrow \texttt{blame}\ p$. Same reasoning as before but with product types. This rule cannot be applied.
- $[\mathrm{R}_{\mathsf{SimplApp}}]$ $V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle V' \hookrightarrow V V'$. By hypothesis, $\tau / \tau' = \tau$, and $\tau \neq \tau'$. Therefore, by Lemma B.22, we have $\tau' = ?$. But $? \not\leq \tau_1 \to \tau_2$ for every τ_1 and τ_2, therefore the reducee cannot be well-typed, and this rule cannot be applied.
- $[\mathrm{R}_{\mathsf{SimplProj}}]$ $\pi_i\ (V\langle \tau \overset{p}{\Rightarrow} \tau' \rangle) \hookrightarrow \pi_i\ V$. Same reasoning as before, this rule cannot be applied.

□

Bibliography

[1] The ℂDuce programming language, 2007. URL `https://www.cduce.org`.

[2] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. *ACM SIGPLAN Notices*, 46 (1):201–214, 2011.

[3] A. Ahmed, D. Jamner, J. G. Siek, and P. Wadler. Theorems for free for free: Parametricity, with and without types. In *International Conference on Functional Programming*, ICFP, September 2017.

[4] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 31–41, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X. doi: 10.1145/165180.165188. URL `http://doi.acm.org/10.1145/165180.165188`.

[5] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1994. ACM Press.

[6] P. Ângelo and M. Florido. Gradual intersection types. In *Workshop on Intersection Types and Related Systems*, 2018.

[7] F. Barbanera, M. Dezani-Ciancaglini, and U. de'Liguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[8] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP '03, 8th ACM International Conference on Functional Programming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press.

[9] G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer Berlin Heidelberg, 2014.

[10] J. P. Campora, S. Chen, M. Erwig, and E. Walkingshaw. Migrating gradual types. *Proc. ACM Program. Lang.*, 2(POPL):15:1–15:29, Dec. 2017.

[11] R. Cartwright and M. Fagan. Soft typing. In *Conference on Programming Language Design and Implementation*, PLDI, pages 278–292. ACM Press, 1991.

[12] G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In Proceedings of *PPDP '05, the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, pages 198-208, ACM Press (full version) and *ICALP '05, 32nd International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science n. 3580, pages 30-34, Springer (summary), Lisboa, Portugal, July 2005. Joint ICALP-PPDP keynote talk.

[13] G. Castagna and V. Lanvin. Gradual typing with union and intersection types. *Proc. ACM Program. Lang.*, 1, Article 41(ICFP '17), Sept. 2017.

[14] G. Castagna and Z. Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP '11: 16th ACM-SIGPLAN International Conference on Functional Programming*, pages 94–106, 2011.

[15] G. Castagna, K. Nguyen, Z. Xu, and P. Abate. Polymorphic functions with set-theoretic types. Part 2: Local type inference and type reconstruction. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages*, POPL '15, pages 289–302. ACM, Jan. 2015.

[16] G. Castagna, T. Petrucciani, and K. Nguyen. Set-theoretic types for polymorphic variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 378–391, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951928. URL `http://doi.acm.org/10.1145/2951913.2951928`.

[17] G. Castagna, G. Duboc, V. Lanvin, and J. Siek. A space-efficient call-by-value virtual machine for gradual set-theoretic types. In *31st International Symposium on Implementation and Application of Functional Languages (IFL'19)*, 2019.

[18] G. Castagna, V. Lanvin, T. Petrucciani, and J. G. Siek. Gradual typing: a new perspective. *Proc. ACM Program. Lang.*, 3, Article 16(POPL '19: 46th ACM Symposium on Principles of Programming Languages), Jan. 2019.

[19] G. Castagna, M. Laurent, V. Lanvin, and K. Nguyen. Revisiting occurrence typing. *Unpublished manuscript*, 2021.

[20] A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, and G. Levi. Fast and precise type checking for javascript. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

[21] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25: 95–169, 1983.

[22] S. Dolan and A. Mycroft. Polymorphism, subtyping, and type inference in mlsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 60–72, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009882. URL `http://doi.acm.org/10.1145/3009837.3009882`.

[23] Facebook. The Flow programming language, 2017. URL `https://flow.org/en/`.

[24] R. B. Findler and M. Felleisen. Contracts for higher-order functions. Technical Report NU-CCS-02-05, Northeastern University, 2002.

[25] A. Frisch. *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. PhD thesis, PhD thesis, Université Paris 7, 2004.

[26] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *LICS '02, 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[27] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):1–64, 2008.

[28] Y.-C. Fuh and P. Mishra. Type inference with subtypes. In H. Ganzinger, editor, *ESOP '88*, pages 94–114, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg. ISBN 978-3-540-38941-5.

[29] R. Garcia. Calculating threesomes, with blame. In *ICFP '13: Proceedings of the International Conference on Functional Programming*, 2013.

[30] R. Garcia and M. Cimini. Principal type schemes for gradual programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 303–315. ACM, 2015.

[31] R. Garcia, A. M. Clark, and É. Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 429–442. ACM, 2016.

[32] N. Gesbert, P. Genevès, and N. Layaïda. A logical approach to deciding semantic subtyping. *ACM Trans. Program. Lang. Syst.*, 38(1):3, 2015. doi: 10.1145/2812805. URL `http://doi.acm.org/10.1145/2812805`.

[33] R. Harper. *Programming Languages: Theory and Practice*. Carnegie Mellon University, 2006. Available on the web: `http://fpl.cs.depaul.edu/jriely/547/extras/misc.pdf`.

[34] F. Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

[35] H. Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, 2001.

[36] H. Hosoya and B. Pierce. Regular expression pattern matching for XML. In *POPL '01, 25th ACM Symposium on Principles of Programming Languages*, 2001.

[37] H. Hosoya and B. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.

[38] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *ICFP '00*, volume 35(9) of *SIGPLAN Notices*, 2000. URL `http://www.cis.upenn.edu/~hahosoya/papers/regsub.ps`.

[39] Y. Igarashi, T. Sekiyama, and A. Igarashi. On polymorphic gradual typing. In *International Conference on Functional Programming*, ICFP. ACM, 2017.

[40] L. Ina and A. Igarashi. Gradual typing for generics. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, 2011.

[41] K. A. Jafery and J. Dunfield. Sums of uncertainty: Refinements go gradual. In *Symposium on Principles of Programming Languages*, POPL, 2017.

[42] JetBrains. The Kotlin programming language, 2018. URL `https://kotlinlang.org`.

[43] M. Keil and P. Thiemann. Blame assignment for higher-order contracts with intersection and union. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 375–386, New York, NY, USA, 2015. ACM.

[44] A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoretical Computer Science*, 311(1):1 – 70, 2004.

[45] G. King. The Ceylon programming language, 2017. URL `https://ceylon-lang.org`.

[46] N. Lehmann and É. Tanter. Gradual refinement types. In *Symposium on Principles of Programming Languages*, POPL, 2017.

[47] A. M. Maidl, F. Mascarenhas, and R. Ierusalimschy. Typed lua: An optional type system for lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, pages 3:1–3:10, New York, NY, USA, 2014. ACM.

[48] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982. ISSN 0164-0925.

[49] Microsoft. The TypeScript programming language, 2017. URL `https://www.typescriptlang.org/index.html`.

[50] J. C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1 (3):245–285, 1991. doi: 10.1017/S0956796800000113.

[51] M. S. New and A. Ahmed. Graduality from embedding-projection pairs. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–30, 2018.

[52] M. S. New, D. R. Licata, and A. Ahmed. Gradual type theory. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '19. ACM, 2019.

[53] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[54] F. Ortin and M. García. Union and intersection types to support both dynamic and static typing. *Information Processing Letters*, 111(6):278 – 286, 2011. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/j.ipl.2010.12.006. URL `http://www.sciencedirect.com/science/article/pii/S0020019010003984`.

[55] D. J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 335–354. Springer, 2013.

[56] T. Petrucciani. *Polymorphic set-theoretic types for functional languages*. PhD thesis, Università di Genova; Université Sorbonne Paris Cité–Université Paris Diderot, 2019.

[57] F. Pottier. Simplifying subtyping constraints: a theory. *Inf. Comput.*, 170(2):153–183, 2001. ISSN 0890-5401.

[58] F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[59] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Symposium on Principles of Programming Languages*, POPL, pages 481–494, January 2012.

[60] J. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.

[61] J. Reynolds. *Programming Methodology*, chapter What do types mean? – From intrinsic to extrinsic semantics. Monographs in Computer Science. Springer, 2003.

[62] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*, pages 513–523, 1983.

[63] S. Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theor. Comput. Sci.*, 59(1-2):181–209, 1988.

[64] T. Sekiyama, A. Igarashi, and M. Greenberg. Polymorphic manifest contracts, revised and resolved. *ACM Trans. Program. Lang. Syst.*, 39(1):3:1–3:36, Feb. 2017.

[65] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of Scheme and Functional Programming Workshop*, pages 81–92. ACM, 2006.

[66] J. G. Siek and W. Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, volume 4609 of *LCNS*, pages 2–27, August 2007.

[67] J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. Technical Report CU-CS-1039-08, University of Colorado at Boulder, January 2008.

[68] J. G. Siek, P. Thiemann, and P. Wadler. Blame and coercion: together again for the first time. In *ACM SIGPLAN Notices*, volume 50, pages 425–435. ACM, 2015.

[69] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[70] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[71] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual typing embedded securely in JavaScript. In *ACM Conference on Principles of Programming Languages (POPL)*, January 2014.

[72] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328486.

[73] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *International Conference on Functional Programming*, ICFP, pages 117–128. ACM, 2010.

[74] M. Toro and É. Tanter. A gradual interpretation of union types. In *Proceedings of the 24th Static Analysis Symposium (SAS 2017)*, volume 10422 of *Lecture Notes in Computer Science*, pages 382–404, New York City, NY, USA, Aug. 2017. Springer-Verlag.

[75] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, pages 1–16. Springer, 2009.

[76] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10: 115–122, 1987.

[77] N. Xie, X. Bi, and B. C. d. S. Oliveira. Consistent subtyping for all. In A. Ahmed, editor, *Programming Languages and Systems*, pages 3–30, Cham, 2018. Springer International Publishing.