Internship Report – M2 MPRI
Gradual Set-Theoretic Types

Victor Lanvin, supervised by Giuseppe Castagna, IRIF

1 Introduction

Set-theoretic types. In programming languages, static type systems are powerful tools that help programmers to ensure that a program is correct, relatively to some specification. The point of a type system is to assign types to every part of a program, using predefined *typing rules*. Types can be seen as denoting sets of values; for example, a possible interpretation of the type `int` of the language OCaml is that it denotes the set of all integers. Set-theoretic type systems make use of this fact to apply the results of set theory to type theory. In such type systems, one can define, e.g., the intersection of the union of two types using their set-theoretic denotations and applying the corresponding set-theoretic operations to them. Set-theoretic types usually restrict themselves to sets where these operations can be implemented efficiently, e.g. regular words or trees (simple types can often be considered as trees).

Most programming languages also support some form of *subtyping*. This provides some flexibility to the type system, and promotes code reusability. In most type systems, subtyping is defined as a binary relation on types using several simple inference rules. However, in set-theoretic type systems, it is possible to define subtyping as a simple set-containment relationship. That is, a type s is a subtype of t if and only if the set of values denoted by s is contained in the set denoted by t [7].

Set-theoretic types are also more powerful than simple types. As it is possible to give "multiple types" to a function (by the means of intersection types), it is possible to type more programs. Specifically, *overloading* is natively supported by set-theoretic types. However, this increase in power comes at a cost: type reconstruction in set-theoretic type systems is usually undecidable. Therefore, the programmer is forced to annotate every part of his code for the type system to check it or to give-up overloading (and impose further restrictions) for the sake of type reconstruction.

Gradual typing. A recent and promising research subject in programming language theory consists of "unifying" the dynamic and static approaches to type-checking. In this domain, particular attention is given to *gradual typing* [9]. Gradual typing proposes to add dynamic type-checking to static type systems by adding an *unknown* type constant (usually denoted by "**?**"), which informs the program that additional dynamic checks may have to be performed at certain places. The idea behind gradual typing is that "only dynamically typed parts of the program can go wrong".

Gradual typing has been used in several projects such as the language Flow [1], or Typescript [2]. It also has been formalized using abstract interpretation [8], and there are methods to automatically add gradual typing to a simple type system [6]. However, a recent study [12] showed that gradual typing may be extremely costly in practice, compared to a fully dynamic or fully static type system.

Motivations. What is interesting with gradual typing is that programmers may add or remove type annotations at their convenience, depending on the amount of information they wants to transmit to the type checker. Therefore, adding gradual typing to a set-theoretic type system would alleviate the syntactic overhead of such a type system, by allowing the programmer to omit certain large type annotations. This would make rapid prototyping possible, while keeping the power of static (set-theoretic) types for critical parts of code.

Contributions. In this report, we present a full type system that integrates gradual typing and set-theoretic types. This type system makes only a few assumptions on the structure of these types and, as we formally prove, it preserves the full power of set-theoretic types. We also present a language that uses

this type system, which is an abstraction of the language the programmer is supposed to program with. We do not define directly the semantics of this language, instead we present a compilation procedure that inserts dynamic type checks in this language, translating its terms into terms of the "cast language". We prove that this translation is sound, that is, every well-typed term of the gradually-typed language is translated into a well-typed term of the cast language. From there, the soundness of the cast language's type system implies a safety property for the type system of the gradually-typed language. To alleviate the performance problems shown in [12], we also integrate a lazy evaluation of casts in our cast-based calculus.

2 Gradual Typing

In this section, we define our static and gradual type system. We also define and study the semantics of the gradual type system by *lifting* the semantics of our static type system.

2.1 Gradual Set-Theoretic Types

The main idea behind set-theoretic types is that types actually denote sets of values. For instance, $\texttt{Int}$ denotes the set of integers, and $\texttt{Int} \to \texttt{Int}$ denotes the set of all functions that when applied to an integer argument they return an integer. Therefore, it is possible to define the union, intersection and negation of types in terms of their set-theoretic counterparts. For example, $\texttt{Int} \vee \texttt{Bool}$ denotes the set containing all integers and booleans. A value of type $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$ is a function that can be applied both to a value of type $\texttt{Int}$ (in that case returning a value of type $\texttt{Int}$) and to a value of type $\texttt{Bool}$ (in that case returning a value of type $\texttt{Bool}$).

On the other hand, gradual typing provides a way for programmers to add or remove type annotations at their convenience, which can make code easier to read or maintain.

The goal of this report is to study the type system obtained by adding gradual typing to a set-theoretic type system. We will consider the following grammar for both kinds of types:

$$t \in \text{SType} ::= t \vee t \mid t \wedge t \mid \neg t \mid t \to t \mid b \mid \mathbb{0} \mid \mathbb{1}$$
$$\tau \in \text{GType} ::= \texttt{?} \mid \tau \vee \tau \mid \tau \wedge \tau \mid \neg t \mid \tau \to \tau \mid b \mid \mathbb{0} \mid \mathbb{1}$$

We also define the *atomic types* (and the *atomic gradual types*) using the following grammar:

$$\textbf{Atom} \quad a ::= b \mid s \to t$$
$$\textbf{Gradual Atom} \quad \alpha ::= \texttt{?} \mid b \mid \sigma \to \tau$$

In this grammar, we use s, t (Latin letters) to range over the set of *static types* SType and σ, τ (Greek letters) to range over the set of *gradual types* GType. The former is formed by basic types (such as $\texttt{Int}, \texttt{Bool}, \ldots$, ranged over by b), a type constructor "$\to$" for function types, type combinators for union, intersection and negation types, as well as $\mathbb{0}$ and $\mathbb{1}$ denoting respectively the empty set of values and the set of all values.

The set GType is simply obtained by adding to the static types a constant $\texttt{?}$, which stands for the absence of type (not to be confused with $\mathbb{0}$ or $\mathbb{1}$). For example, a value of type $\texttt{?} \to \texttt{Int}$ can take an argument of a certain (unknown) type, and return an integer. However, contrary to $\mathbb{1} \to \texttt{Int}$, this application might fail if the argument is not of the expected type. Using the same logic, a value of type $(\texttt{Int} \to \texttt{Int}) \wedge \texttt{?}$ can be used as a function of type $\texttt{Int} \to \texttt{Int}$ but can also be used as a value of *any other type*, (this is useful in practice provided that one adds dynamic type checks).

We also use the standard convention that the connectives $\wedge$ and $\vee$ are given a higher precedence than the constructor $\to$. Therefore, the type $\texttt{Int} \to \texttt{Int} \vee \texttt{Bool} \to \texttt{Bool}$ denotes currified functions that take two arguments of type $\texttt{Int}$ and $\texttt{Int} \vee \texttt{Bool}$ respectively, and return a Boolean value.

Given this grammar, it is easy to see that SType $\subset$ GType. Therefore, any function on gradual types can be implicitely restricted to static types. Moreover, one can remark that we do not consider negations of gradual types. In addition to being difficult to formalize, one can wonder what would be the meaning —intuitively— of a type as simple as $\texttt{Int} \to \texttt{Int} \wedge \neg\, \texttt{?}$.

In the following, we consider that the set of static types is equipped with a subtyping relation $\leq$ (coinciding with set-containment of the denotations), as well as the underlying equivalence relationship $\simeq$, defined as:

$$\forall (t_1, t_2) \in \text{SType}^2, t_1 \simeq t_2 \iff t_1 \leq t_2 \text{ and } t_2 \leq t_1$$

Intuitively, two static types are equivalent if and only if they denote the same set of values.

2.2 Types Semantics

In this part, we will study the semantics of the previously defined gradual types. To achieve this, we adapt the approach proposed in [8] that consists in using abstract interpretation to lift the semantics of the static type system.

2.2.1 Concretisation.

The first point of this approach is to define the *concretisation* of a gradual type, that is, the set of its possible *interpretations*. For example, the constant `?` can be interpreted as any type, and the type `? → ?` can be interpreted as any function type. Formally, we define the concretisation function γ on gradual types as follows:

$$\gamma : \text{GType} \to \mathcal{P}(\text{SType})$$

$$\begin{aligned}
\gamma(\texttt{?}) &= \text{SType} & \gamma(\tau_1 \to \tau_2) &= \{t_1 \to t_2 \mid t_i \in \gamma(\tau_i)\} \\
\gamma(\tau_1 \vee \tau_2) &= \{t_1 \vee t_2 \mid t_i \in \gamma(\tau_i)\} & \gamma(b) &= \{b\} \\
\gamma(\tau_1 \wedge \tau_2) &= \{t_1 \wedge t_2 \mid t_i \in \gamma(\tau_i)\} & \gamma(\mathbb{0}) &= \{\mathbb{0}\} \\
\gamma(\neg\tau) &= \{\neg t \mid t \in \gamma(\tau)\} & \gamma(\mathbb{1}) &= \{\mathbb{1}\}
\end{aligned}$$

Basically, γ returns the set of static types obtained by substituting the occurrences of `?` with all possible static types.

Using this concretisation function, it is possible to give a formal meaning to the precision of a gradual type.

Definition 1. *(Gradual Type Precision) — A gradual type τ is said to be "more precise" than a gradual type σ, noted $\tau \sqsubseteq \sigma$, if for every type $t \in \gamma(\tau)$, there exists a type $s \in \gamma(\sigma)$ such that $t \simeq s$.*

Informally, a gradual type τ is more precise than a gradual type σ if every interpretation of τ is a possible interpretation of σ. For example, `Int → ?` is more precise than `? → ?`, but `Int → ?` and `Nat → ?` are not comparable.

Thanks to our set-theoretic approach, the concretisation of a gradual type has some interesting properties. In particular, for every gradual type τ, the set $\gamma(\tau)$ has a maximum and a minimum (i.e. it is a closed sublattice of SType). This is a consequence of the fact that the set of static types is a complete lattice (containing $\mathbb{1}$ and $\mathbb{0}$).

Proposition 1. *(Extrema) — For every type $\tau \in$ GType, there exists two static types $\tau^{\Uparrow}$ and $\tau^{\Downarrow}$ in $\gamma(\tau)$ such that, for every type $t \in \gamma(\tau)$, $\tau^{\Downarrow} \leq t \leq \tau^{\Uparrow}$.*
$\tau^{\Uparrow}$ (resp. $\tau^{\Downarrow}$) is defined by replacing all positive occurrences[1] of `?` by $\mathbb{1}$ (resp. $\mathbb{0}$) and all negative occurrences of `?` by $\mathbb{0}$ (resp. $\mathbb{1}$).

2.2.2 Collective lifting.

Using the concretisation function, we are now able to give a set of (static) interpretations to a gradual type. Our main objective is to use this concretisation function to extend (or "lift") functions over the static types to gradual types. In particular, we want to define subtyping on gradual types using its definition on static types. To realize this lifting, we first need to define the *collective lifting* of a function to apply it to sets of static types.

[1] A negative occurrence of `?` is an occurrence that is to the left of an odd number of arrows. An occurrence is positive if it is not negative.

Definition 2. *(Collective lifting) — Let $f_S : \text{SType} \to \text{SType}$ be a function on static types. We define its collective lifting $f_C : \mathcal{P}(\text{SType}) \to \mathcal{P}(\text{SType})$ as follows:*

$$\forall S \in \mathcal{P}(\text{SType}),\ f_C(T) = \{f_S(t) \mid t \in T\}$$

Informally, the collective lifting of a function f_S on static types is simply a function f_C that applies f_S to every element of a set of static types.

2.2.3 Abstraction.

Given a function f_S on static types, we are now able to define a lifting that returns the set of the possible interpretations of f_S on a gradual type. The last step needed to properly lift our static semantics is to *abstract* this set of interpretations back to a gradual type.

Definition 3. *(Abstraction) — We define the abstraction function α on $\mathcal{P}(\text{SType})$ as follows:*

$$\alpha : \mathcal{P}(\text{SType}) \to \text{GType}$$

$$\alpha(S) = \min\{\tau \mid S \subset \gamma(\tau)\}$$

Unfortunately, as opposed to the definition proposed in [8], this definition of the abstraction function is not really practical. Indeed, because set-theoretic types do not have a unique representation (`Int` $\wedge$ `Bool` and `Bool` $\wedge$ `Int` represent the same type for example), it is difficult to find a non-syntactic and non-intentional definition of α.

2.2.4 Lifting functions and predicates.

Using the previously defined concretisation and abstraction functions, it is possible to *lift* any function or predicate defined on static types to gradual types [8]. The following definition of the *gradual lifting* handles the case of binary functions and predicates but can be easily generalized for n parameters.

Definition 4. *(Gradual Lifting) [8] — Let $P_S : \text{SType}^2$ be a binary predicate and $f_S : \text{SType}^2 \to \text{SType}$ a binary function on static types. We define their respective gradual liftings $P_G : \text{GType}^2$ and $f_G : \text{GType}^2 \to \text{GType}$ as:*

$$P_G(\tau_1, \tau_2) \Leftrightarrow \exists (t_1, t_2) \in \gamma(\tau_1) \times \gamma(\tau_2)\ s.t.\ P_S(t_1, t_2)$$

$$\begin{aligned} f_G(\tau_1, \tau_2) &= \alpha(f_C(\gamma(\tau_1), \gamma(\tau_2))) \\ &= \alpha(\{f_S(t_1, t_2) \mid t_i \in \gamma(\tau_i)\}) \end{aligned}$$

We can now give precise semantics to the subtyping relation on gradual types by simply lifting its static counterpart. Indeed, according to the previous definition, this relation is defined as $\tau \leq_G \sigma$ if and only if there exists two static types $t \in \gamma(\tau)$ and $s \in \gamma(\sigma)$ such that $t \leq s$. For example, **?** is both a subtype and a supertype of any type ($\mathbb{1}$ and $\mathbb{0}$ included), insofar as any type is a possible concretisation of **?** (i.e. it is an element of $\gamma($ **?**$)$).

Moreover, using the extrema of a gradual type, one can then remark that $\tau \leq_G \sigma$ is equivalent to $\tau^{\Downarrow} \leq \sigma^{\Uparrow}$. This equivalence is important as it implies that the subtyping problem for gradual types can be reduced to the same problem on static types, and that we can re-use the existing algorithms. In particular, since $\tau^{\Uparrow}$ and $\tau^{\Downarrow}$ are of the same size as τ and preserve static types, leq_G and $\leq$ have the same algorithmic complexity. In practice, this amounts to a simple pattern matching on types.

Theorem 1. *(Gradual Subtyping) — Deciding gradual subtyping is equivalent to deciding static subtyping. More precisely, for all gradual types τ and σ, we have*

$$\tau \leq_G \sigma \iff \tau^{\Downarrow} \leq \sigma^{\Uparrow}$$

Proof. Let τ and σ be two gradual types such that $\tau \leq_G \sigma$. By definition, there exists a pair of static types $(t, s) \in \gamma(\tau) \times \gamma(\sigma)$ such that $t \leq s$. Moreover, we know that $\tau^{\Downarrow} \leq t$ and $s \leq \sigma^{\Uparrow}$. By transitivity of the subtyping relation on static types, we have $\tau^{\Downarrow} \leq \sigma^{\Uparrow}$.

Given that $\tau^{\Downarrow} \in \gamma(\tau)$ and $\sigma^{\Uparrow} \in \gamma(\sigma)$, the reverse is immediate. □

Finally, it is important to note that the subtyping relation on gradual types *is not* transitive. If this were the case, the relation would collapse to the trivial relation where $\sigma \leq \tau$ for every types σ and τ. This is a consequence of the fact that both $\sigma \leq \,?$ and $? \leq \tau$ hold for every types σ and τ.

2.3 Gradual Function Types

Now that we are able to give precise semantics to gradual types in terms of static types, we can study more precisely the semantics of the types associated to functions, that is, all types that are subtypes of $\mathbb{0} \to \mathbb{1}$. Such a type will be called a *function type*.

2.3.1 Semantics of static function types.

Defining the semantics of static, set-theoretic function types require a bit of work, as presented in [7]. First of all, to avoid any potential issues with recursive definitions, it is possible to rewrite any static type into an equivalent type written in disjunctive normal form. This is a direct property of the set-theoretic approach, as most of the laws of propositional logic are still valid in this model.

Proposition 2. *(Disjunctive Normal Form) [7] — Every static type t is equivalent to a type in disjunctive normal form, that is, a type of the following form:*

$$t \simeq \bigvee_{i \in I} \Big(\bigwedge_{j \in J_i} a_j \wedge \bigwedge_{n \in N_i} \neg a_n \Big)$$

where a_i and a_n are atomic types. Moreover, every function type t is equivalent to a type of the following form:

$$t \simeq \bigvee_{i \in I} \Big(\bigwedge_{j \in J_i} s_j \to t_j \wedge \bigwedge_{n \in N_i} \neg(s_n \to t_n) \Big)$$

We also define the function $NF : \mathrm{SType} \to \mathrm{SType}$ that writes a type in DNF.

The disjunctive normal form is defined with the convention that $\mathbb{0}$ and $\mathbb{1}$ are, respectively, the empty union and the empty intersection. Types in this form can be seen as "flattened" trees, and are easier to implement and manipulate. In particular, functions and predicates on these types can be defined in a non-recursive way.

Definition 5. *(Domain) — For every function type written in disjunctive normal form, we define its domain as follows:*

$$dom\Big(\bigvee_{i \in I} \Big(\bigwedge_{j \in J_i} s_j \to t_j \wedge \bigwedge_{n \in N_i} \neg(s_n \to t_n) \Big)\Big) = \bigwedge_{i \in I} \bigvee_{j \in J_i} s_j$$

The definition of the domain is quite intuitive: a function whose type is an intersection of arrows (for example, $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$) can be applied as any of those types. Therefore, the domain of an intersection of arrows is the union of their domains (in our example, $\texttt{Int} \vee \texttt{Bool}$). On the other hand, if a function is typed by a union of function types, it only accepts arguments that are compatible with every arrow in the union. Therefore, the domain of a union of arrows is the intersection of their domains. Note that the negations of arrows in the normal form can be safely ignored. Indeed, all meaningful negations can be removed when rewritting the type in disjunctive normal form. For example, $(\texttt{Int} \to \texttt{Int}) \wedge \neg(\texttt{Nat} \to \texttt{Int})$ can be rewritten to $(\texttt{Int} \wedge \neg\texttt{Nat}) \to \texttt{Int}$.

Definition 6. *(Result Type) [7] — For every function type t written in disjunctive normal form, and any type s, we define $t \bullet s$, the result type of the application of t to s, as:*

$$\bigvee_{i \in I} \Big(\bigwedge_{j \in J_i} s_j \to t_j \wedge \bigwedge_{n \in N_i} \neg(s_n \to t_n) \Big) \bullet s = \bigvee_{i \in I} \Big(\bigvee_{\substack{Q \subsetneq J_i \\ s \not\leq \bigvee_{q \in Q} s_q}} \Big(\bigwedge_{p \in J_i \setminus Q} t_p \Big) \Big)$$

This definition is a bit complicated, so we will explain it step-by-step. First of all, for the same reasons as for the domain of a function, the negations of arrows in the normal form can be safely ignored.

Secondly, consider a function that can be typed with an intersection of arrows (in disjunctive normal form) $t = \bigwedge_{j \in J} s_j \to t_j$, and an argument of type s. We know that, for every $j \in J$ such that $s \leq s_j$,

the result of the application has type t_j. Therefore, it would be tempting to define $t \bullet s = \bigwedge_{j \in J \mid s \leq s_j} t_j$. However, this does not take into account the fact that there might be overlaps between s and s_j, even if $s \not\leq s_j$. For example, applying a function of type $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Nat} \to \texttt{Nat})$ to an argument of type `Int` *might* return a value of type $\texttt{Int} \wedge \texttt{Nat} \simeq \texttt{Nat}$ (if the argument is 2 for example). Formally, for every subset Q of J such that s is not a subtype of any type s_q where $q \in Q$ (ie. $s \not\leq \bigvee_{q \in Q} s_q$), the result *can* be of type $\bigwedge_{p \in J \setminus Q} t_p$, hence the innermost union.

Finally, the codomain of an union is simply defined as the union of the codomains, hence the outermost union.

2.3.2 Concretising function types.

Unfortunately, it is difficult to give a simple, structural definition of the gradual lifting of the domain and result type of a function type. This is due to several reasons. First of all, the intentional nature of the abstraction function makes it difficult to deduce simple liftings. Secondly, an interpretation of a gradual type in disjunctive normal form is not necessarily a static type in disjunctive normal form. For example, $? \wedge \texttt{Int}$ is a gradual type in DNF, but $(\texttt{Char} \vee \texttt{Bool}) \wedge \texttt{Int}$ is a valid interpretation of this type which is not in DNF. Therefore, applying the collective lifting of the domain and result type to a set of static types produces a lot of types that have potentially nothing in common.

Moreover, defining an equivalence relationship on gradual types such that gradual types are equivalent to a type in disjunctive normal form, and such that the domain and result type are defined uniquely among an equivalence class proved to be a hard task.

Therefore, we decided to circumvent the problem by remarking that:

- Since we only want to compute the domain of a gradual function type and the result type of the application of a gradual type, it is not necessary to concretise the codomain of a gradual type. Indeed, the codomain will always be left unchanged by an application. For example, the application of a value of type $\tau \to \sigma$ to a value of type (or subtype of) τ will always be σ.

- It is also not necessary to consider all possible interpretations of the domain of a function. Indeed, there are only two cases to consider when computing the possible interpretations of the application of a value of type $\tau \to \sigma$ to a value of type τ': whether every interpretation of τ' is a subtype of every interpretation of τ (meaning the application will never result in an error) or not.

  For example, when applying a function of type $(\texttt{Int} \vee ?) \to \texttt{Int}$ to an argument of type `Int`, we know that the application will succeed, whatever the real (dynamic) type of the function is. However, if we apply the same function to an argument of type `Bool`, the application *may* succeed (if the function admits the more precise type $(\texttt{Int} \vee \texttt{Bool}) \to \texttt{Int}$, for example), but may also fail.

  We already saw that such a subtyping problem can be solved by only considering the maximum and the minimum of a gradual type. Therefore, when dealing with a value of type $\tau \to \sigma$, and we want to study its application to an argument, it is enough to consider the two interpretations $\tau^{\Uparrow} \to \sigma$ and $\tau^{\Downarrow} \to \sigma$.

- When considering a gradual type as a function type, $?$ has the same behaviour as $? \to ?$.

Using the above remarks, we can define a finite set of (gradual) interpretations for a gradual function type.

Definition 7. *(Finite Concretisation of Function Types) — We define the functions $\mathcal{A}^{\Uparrow}$ (maximal concretisation of a function type) and $\mathcal{A}^{\Downarrow}$ (minimal concretisation) on gradual types as follows:*

$$
\begin{aligned}
\mathcal{A}^{\Uparrow} &: \mathrm{GType} \to \mathcal{P}(\mathrm{GType}) \\
\mathcal{A}^{\Uparrow}(\tau_1 \vee \tau_2) &= \mathcal{A}^{\Uparrow}(\tau_1) \sqcup \mathcal{A}^{\Uparrow}(\tau_2) \\
\mathcal{A}^{\Uparrow}(\tau_1 \wedge \tau_2) &= \{\sigma_1 \wedge \sigma_2 \mid \sigma_i \in \mathcal{A}^{\Uparrow}(\tau_i)\} \\
\mathcal{A}^{\Uparrow}(\sigma \to \tau) &= \{\sigma^{\Downarrow} \to \tau; \sigma^{\Uparrow} \to \tau\} \\
\mathcal{A}^{\Uparrow}(\bigvee_{i \in I} \bigwedge_{j \in J_i} s_j \to t_j \wedge \bigwedge_{n \in N_i} \neg(s_n \to t_n)) &= \bigsqcup_{i \in I} \{\bigwedge_{j \in J_i} s_j \to t_j\} \\
\mathcal{A}^{\Uparrow}(t) &= \mathcal{A}^{\Uparrow}(NF(t)) \\
\mathcal{A}^{\Uparrow}(\,?\,) &= \{\mathbb{0} \to \,?\,\} \\
\mathcal{A}^{\Uparrow}(b) &= \{\mathbb{0} \to \mathbb{1}\}
\end{aligned}
$$

Where $\sqcup : \mathcal{P}(\mathrm{GType})^2 \to \mathcal{P}(\mathrm{GType})$ *distributes an union over two intersections of arrows, that is:*

$$
T_1 \sqcup T_2 = \{ \bigwedge_{(i_1,i_2) \in I_1 \times I_2} s_{i_1} \wedge s_{i_2} \to \tau_{i_1} \vee \tau_{i_2} \mid \bigwedge_{i_k \in I_k} s_{i_k} \to \tau_{i_k} \in T_k, \quad k \in \{1,2\}\}
$$

The minimal concretisation $\mathcal{A}^{\Downarrow}$ *is defined analogously (by substituting* $\mathcal{A}^{\Uparrow}$ *by* $\mathcal{A}^{\Downarrow}$*), except for the case* $?$ *where* $\mathcal{A}^{\Downarrow}$ *is defined as* $\mathcal{A}^{\Downarrow}(\,?\,) = \{\mathbb{1} \to \,?\,\}$.

Moreover, we define the finite concretisation of a function type $\mathcal{A}(\tau)$ *as the union* $\mathcal{A}^{\Uparrow}(\tau) \cup \mathcal{A}^{\Downarrow}(\tau)$.

Informally, $\mathcal{A}(\tau)$ represents the "possible behaviours" of a value of a gradual function type τ that can be observed (at type level) when this value is applied to an argument.

There are several things to note about this definition. First of all, the set of interpretations of a gradual function type contains only intersection of arrows, each of them having as domain a static type. This can be seen as a way to "simulate" a disjunctive normal form and makes it easier to define the domain and result type for gradual functions.

Secondly, we do not directly handle the case of negations. This is because we only allow negations of static types, which are then handled like any static type (by writing them in disjunctive normal form).

Finally, this definition distinguishes between two subsets of interpretations, $\mathcal{A}^{\Uparrow}$ and $\mathcal{A}^{\Downarrow}$, which only differ on the domain they assign to the unknown type. Intuitively, $\mathcal{A}^{\Uparrow}(\tau)$ is the set of interpretations obtained by assuming that none of the unknown types are actually interpreted as (applicable) function types. On the contrary, $\mathcal{A}^{\Downarrow}(\tau)$ is the set of interpretations obtained by assuming that all the unknown types are interpreted as functions. Intuitively, for every interpretation $\sigma \in \mathcal{A}^{\Uparrow}(\tau)$, *we are sure* that a value of type τ can be applied as a function of type σ. However, for every interpretation $\sigma \in \mathcal{A}^{\Downarrow}(\tau)$, *it is possible* that a value of type τ can be applied as a function of type σ, but this application *may* fail. Therefore, in this case, we need to insert dynamic type checks.

Let us consider, for example, an application $e_1\ e_2$ where e_1 has type $?$. This application should be accepted statically because it *might* succeed, since any function type is a possible interpretation of $?$. This is reflected by the fact that $\mathcal{A}^{\Downarrow}(\,?\,)$, the set of the "most lenient" interpretations of $?$, is actually $\{\mathbb{1} \to \,?\,\}$. However, such an application should require additional dynamic checks (typically, verifying that e_1 actually evaluates to a function), because it *might* also fail, since non-function types are also possible interpretations of $?$. This is reflected by the fact that $\mathcal{A}^{\Uparrow}(\,?\,)$, the set of the "most restrictive" (in terms of domain) interpretations of $?$ is $\{\mathbb{0} \to \,?\,\}$.

Note that we only use these functions to check if an expression can be applied, and to compute the result type of an application. Thus, we only need approximations of the actual possible interpretations (as function types) of a type. In terms of domain and codomain, `Int` and $\mathbb{0} \to \mathbb{1}$ behave in the same way (since both types denote values that cannot be applied); hence the value of $\mathcal{A}^{\Uparrow}(\,?\,)$ or $\mathcal{A}(b)$.

2.3.3 Gradual definition.

Using the finite concretisation function, we can now define the domain and result of a gradual function type in a similar way as for static types.

Definition 8. *(Gradual Domain) — For every gradual function type τ, we define its domain as the "largest possible", that is,*

$$gdom(\tau) = \bigvee_{\bigwedge_{i\in I} s_i\to\tau_i\in\mathcal{A}(\tau)} (\bigvee_{i\in I} s_i)$$

As expected, a gradual function type can be applied as any of its interpretations, therefore its domain is the union of all its possible domains. However, one can remark that, for the same reasons as for the concretisation of a gradual type, there is a type in $\mathcal{A}(\tau)$ with a maximal domain and one with a minimal domain. Therefore, the outermost union in the definition of the gradual domain is simply a convenient way to denote the maximal domain of the set $\mathcal{A}(\tau)$. We also define analogously $gdom^{\Uparrow}(\tau)$ by only taking the union over $\mathcal{A}^{\Uparrow}(\tau)$. Note that $gdom^{\Uparrow}(\tau) \leq gdom(\tau)$.

Intuitively, using the previous explanations about the function $\mathcal{A}^{\Uparrow}$, one can see $gdom^{\Uparrow}(\tau)$ as the *safe* domain of τ. That is, any value that can be typed with a subtype of $gdom^{\Uparrow}(\tau)$ can be passed to a function of type τ, and the application will succeed.

As a sidenote, defining a function $gdom^{\Downarrow}$ in the same way as $gdom^{\Uparrow}$ would be redundant, as $gdom^{\Downarrow}$ would actually be equal to $gdom$.

Definition 9. *(Gradual Result Type) — For every gradual function type τ and every type σ, we define the result type of the application of τ to σ (noted $\tau\,@\,\sigma$) as:*

$$\tau\,@\,\sigma = \bigvee_{\bigwedge_{i\in I} s_i\to\tau_i\in\mathcal{A}(\tau)} \bigvee_{\substack{Q\subsetneq I \\ \sigma^{\Uparrow}\not\leq\bigvee_{q\in Q} s_q}} (\bigwedge_{p\in I\setminus Q} \tau_p)$$

Once again, this definition is really similar to its static counterpart. However, one should remark the particularity that arises from the fact that the type of the argument can be a gradual type, that is, the condition $\sigma^{\Uparrow} \not\leq \bigvee_{q\in Q} s_q$.

This condition can be explained simply by lifting its static counterpart. Indeed, in the static definition of the result type, we have the similar condition $s \not\leq \bigvee_{q\in Q} s_q$. Let $\leq_{\mathrm{G}}$ and $\not\leq_{\mathrm{G}}$ denote respectively the gradual lifting of $\leq$ and $\not\leq$. It is not true that $\tau \not\leq_{\mathrm{G}} \sigma \iff \neg(\tau \leq_{\mathrm{G}} \sigma)$, due to the existential quantification that is introduced by the lifting. However, much like $\leq_{\mathrm{G}}$ can be expressed in terms of extrema, we have that $\tau \not\leq_{\mathrm{G}} \sigma \iff \tau^{\Uparrow} \not\leq \sigma^{\Downarrow}$, hence the lifting of the condition.

Informally, this amounts to distinguishing the case where the argument can be passed to the function and the case where it has an incompatible type. For example, when applying a function of type $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$ to an argument of type `?`, the argument can either be a subtype of `Int` (the function therefore returning `Int`), a subtype of `Bool` (returning `Bool`), a subtype of both (returning $\texttt{Int} \wedge \texttt{Bool} \simeq \mathbb{0}$) or none of those, raising a cast error.

Of course, we would also like our gradual type system to be a conservative extension of the underlying static type system. To achieve this, the gradual definitions of the domain and the result type of a function need to coincide with their static counterparts. Therefore, we prove the following property about the domain and result of static function types:

Proposition 3. *For every static function types t and s,*

$$gdom(t) \simeq dom(t)$$

$$t\,@\,s \simeq t\bullet s$$

This is an important property as it will help us show that, in the absence of gradually typed terms, our lambda calculus behaves in the same way as a statically typed one (ie. it does not produce errors).

3 Gradually-Typed Language

In this section, we define the syntax of a language that uses the gradual types we defined in the previous section, and give the corresponding typing rules.

3.1 Syntax

3.1.1 Language syntax.

The gradually-typed language we consider here is closely related to the simply typed lambda-calculus. Its syntax is defined by the following grammar:

$$\begin{aligned} \textbf{Term} \quad & e ::= x \mid c \mid \lambda^{\mathbb{I}} x.\ e \mid e\ e \mid (e \in t)?e : e \\ \textbf{Value} \quad & v ::= x \mid c \mid \lambda^{\mathbb{I}} x.\ e \\ \textbf{Interface} \quad & \mathbb{I} ::= \{\sigma_i \to \tau_i \mid i \in I\} \end{aligned}$$

Although this grammar is similar to the simply typed lambda-calculus', there are a few additions that arise from our type system. Due to the dynamic nature of gradual types, we have to keep a dynamic representation of types. Therefore, we allow dynamic type tests, noted $(e \in t)?e_1 : e_2$, which branch either to e_1 or to e_2 according to whether the result of the expression e is of type t or not. Additionally, we do not allow gradual types in these dynamic tests. This is because gradual subtyping can be reduced to static subtyping, and checking the type of a value against a gradual type τ would amount to checking its type against $\tau^{\Uparrow}$.

Moreover, rather than giving an explicit type only to the arguments of a function (as in the simply typed lambda-calculus), we give an explicit *interface* to every function. An interface is a set of arrows, which stands for the set of the possible types of a function. Giving explicit function types to abstractions allows us to have more precise types: for example, although the two function types $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$ and $(\texttt{Int} \vee \texttt{Bool}) \to (\texttt{Int} \vee \texttt{Bool})$ have the same domain (and compatible codomains), the former is way more precise than the latter[2], but it cannot be expressed if we only give an explicit type to the arguments.

3.1.2 Interfaces.

When considering only static types, the type of a function is —usually— the intersection of the types in its interface, as it can be considered as a value of any of those types. For example, the type of a (well-typed) function having the interface $\{\texttt{Int} \to \texttt{Int}; \texttt{Bool} \to \texttt{Bool}\}$ is indeed the intersection $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$

However, when dealing with gradual types, there are several problems with this approach. For example, there are two possible ways to understand the interface $\{\texttt{Int} \to \texttt{Int};\ ? \to\ ?\}$. One could say that a function with this interface returns a value of type `Int` when applied to an argument of type `Int`, and returns something else when applied to an argument that is not of type `Int`. Or, one could also interpret the interface as stating that the type of this function is the intersection $(\texttt{Int} \to \texttt{Int}) \wedge (\ ? \to\ ?)$, that is to say, that it returns a value of type $\texttt{Int} \wedge\ ?$ (which is a subtype of every type) when given a parameter of type `Int` (just apply the definition of gradual type application given in the previous section).

We chose to consider the first approach, which seemed more intuitive and closer to the philosophy of gradual types. Therefore, to keep the intuition that the type of a function is the intersection of the types in its interface (and thus ease the formalization), we decided to put a restriction on the interfaces: all the domains (left part) of the arrows of an interface have to be distinct. Formally, in what follows, we will only consider the interfaces $\{\sigma_i \to \tau_i \mid i \in I\}$ such that $\forall (i,j) \in I^2, \sigma_i \wedge \sigma_j \simeq \mathbb{0}$.

As an example, the interface $\{\texttt{Int} \to \texttt{Int};\ ? \to\ ?\}$ is not a valid interface (because $\texttt{Int} \wedge\ ? \neq \mathbb{0}$), but $\{\texttt{Int} \to \texttt{Int}; (\ ?\backslash\texttt{Int}) \to\ ?\}$ is.

This new definition is not restrictive, as the transformation of an arbitrary interface to a valid interface can be done statically (although this can lead to an exponential blow-up on the size of an interface). For example, the invalid interface $\{\texttt{Nat} \to \texttt{Nat}; \texttt{Even} \to \texttt{Even};\ ? \to\ ?\}$ can be converted statically into the (intuitively) equivalent interface

$$\{(\texttt{Nat}\backslash\texttt{Even}) \to \texttt{Nat}; (\texttt{Nat} \wedge \texttt{Even}) \to (\texttt{Nat} \wedge \texttt{Even}); (\texttt{Even}\backslash\texttt{Nat}) \to \texttt{Even}; (\ ?\backslash(\texttt{Nat} \wedge \texttt{Even})) \to\ ?\}$$

[2] If you apply a function of type $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$ to a parameter of type `Int`, you know that the result will be of type `Int`. Whereas, if you apply a function of type $(\texttt{Int} \vee \texttt{Bool}) \to (\texttt{Int} \vee \texttt{Bool})$ to the same parameter, all you know is that the result will be of type $\texttt{Int} \vee \texttt{Bool}$. Thus, the former type conveys more information that the latter.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}(\text{TVar}) \qquad \frac{}{\Gamma \vdash c : B(c)}(\text{TCst}) \qquad \frac{\forall(\sigma \to \tau) \in \mathbb{I}, \quad \Gamma, x : \sigma \vdash e : \tau}{\lambda^{\mathbb{I}} x.\ e : \texttt{TypeOf}(\mathbb{I})}(\text{TAbs})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad {\tau_1}^{\Downarrow} \leq \mathbb{0} \to \mathbb{1} \quad {\tau_2}^{\Downarrow} \leq gdom(\tau_1)}{\Gamma \vdash e_1\ e_2 : \tau_1 @ \tau_2}(\text{TApp})$$

$$\frac{\Gamma \vdash e : \tau \quad \begin{cases} \tau^{\Uparrow} \not\leq \neg t & \implies \Gamma \vdash e_1 : \sigma_1 \\ \tau^{\Uparrow} \not\leq t & \implies \Gamma \vdash e_2 : \sigma_2 \end{cases}}{\Gamma \vdash ((e \in t)?e_1 : e_2) : \sigma_1 \vee \sigma_2}(\text{TCase})$$

Figure 1: Typing rules for the gradually-typed language

In the following, we will use $\texttt{TypeOf}(\mathbb{I})$ to denote the type associated to a valid interface $\mathbb{I}$, that is, the intersection of its arrow types. Formally, we define $\texttt{TypeOf}(\mathbb{I}) = \bigwedge_{(\sigma\to\tau)\in\mathbb{I}} \sigma \to \tau$.

As a sidenote, to preserve the advantages of gradual typing, it is also possible to accept empty interfaces. In this case, such interfaces will be interpreted as $\{\, \texttt{?} \to \texttt{?}\}$.

3.2 Typing

Having explained how to type abstractions, we now use the notions defined in the previous section to give the typing rules for our gradually-typed language. The rules are shown in Figure 1 and assume that we have a function B that associates to every constant c its static type.

The two rules that have not been explained yet are (Tapp) and (Tcase). The former, (Tapp), is similar to the one presented in [8]. For an application $e_1\ e_2$, this rule simply checks that e_1 effectively has a function type (i.e. that its gradual type is a gradual subtype of $\mathbb{0} \to \mathbb{1}$, the type of all functions), and that the type of e_2 is compatible with (i.e. it is a gradual subtype of) the domain of e_1.

The second rule, (Tcase), is simply defined as the lifting of its static counterpart, presented in [5]. Intuitively, when evaluating the type of the condition $(e \in t)?e_1 : e_2$, we only need to check the type of the branches that *can* be evaluated. Thus, given the type τ of the expression e, if there is an interpretation of τ that is not a subtype of t (i.e. ${\tau}^{\Uparrow} \not\leq t$), then *there is a possibility* that, dynamically, the condition fails. Therefore, in that case, we need to check the type of the second branch. The same reasoning can be done with $\neg t$ and the first branch, hence the two cases of the rule (Tcase). Of course, if the first condition (respectively the second condition) does not hold, then the type of the typecase is σ_2 (respectively σ_1), rather that the union $\sigma_1 \vee \sigma_2$.

Rationale: The language we defined in this section is an abstraction of the language the programmer is supposed to program with. Notice that we did not define the semantics of this language and *a fortiori* we did not prove any soundness or safety property of the type system defined above. The semantics of this language will be given by translating its terms into the "cast language" we define in the next section. The translation is defined in Section 5 where we also prove (cf. Theorem 3) that every well-typed term of this language is translated into a well-typed term of the cast language. The soundness of the cast language's type systems (cf. Theorem 2) implies a safety property (as expressed in Corollary 2) for the type system of Figure 1.

4 Cast Language

Having defined the syntax of a gradually-typed language, we now need to define its semantics. To that end we need a target language, that is, a lambda-calculus that contains dynamic type verifications (a.k.a. *casts*) in which the gradually-typed language will be compiled to. In this section, we define this language, its syntax, static and dynamic semantics, and prove the soundness of its type system.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}(\text{TCVar}) \qquad \frac{}{\Gamma \vdash c : B(c)}(\text{TCCst}) \qquad \frac{\forall(\sigma \to \tau') \in \mathbb{I}, \quad \Gamma, x : \sigma \vdash e : \tau'}{\lambda^{\mathbb{I}}_{\langle\tau\rangle}\ x.\ e : \tau}(\text{TCAbs})$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \langle\tau\rangle\ e : \tau}(\text{TCCast})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad {\tau_1}^{\Downarrow} \leq \mathbb{0} \to \mathbb{1} \quad {\tau_2}^{\Uparrow} \leq gdom^{\Uparrow}(\tau_1)}{\Gamma \vdash e_1\ e_2 : \tau_1 @ \tau_2}(\text{TCApp})$$

$$\frac{\Gamma \vdash e : \tau \quad \begin{cases} \tau^{\Uparrow} \not\leq \neg t & \implies \Gamma \vdash e_1 : \sigma_1 \\ \tau^{\Uparrow} \not\leq t & \implies \Gamma \vdash e_2 : \sigma_2 \end{cases}}{\Gamma \vdash ((e \in t)?e_1 : e_2) : \sigma_1 \vee \sigma_2}(\text{TCCase})$$

Figure 2: Typing rules for the cast language

4.1 Syntax

The target language we consider here is closely related to the gradually-typed lambda calculus defined in the previous section. It is defined by the following grammar:

$$\begin{aligned} \textbf{Term} \quad & e ::= x \mid c \mid \lambda^{\mathbb{I}}_{\langle\tau\rangle}\ x.\ e \mid e\ e \mid (e \in t)?e : e \mid \langle\tau\rangle\ e \\ \textbf{Value} \quad & v ::= c \mid \lambda^{\mathbb{I}}_{\langle\tau\rangle}\ x.\ e \\ \textbf{Interface} \quad & \mathbb{I} ::= \{\sigma_i \to \tau_i \mid i \in I\} \\ \textbf{Error} \quad & \mathcal{E} ::= \texttt{CastError} \end{aligned}$$

As before, every interface $\mathbb{I} = \{\sigma_i \to \tau_i \mid i \in I\}$ must satisfy the condition $\forall(i,j) \in I^2, \sigma_i \wedge \sigma_j = \mathbb{0}$.

The most important change in this grammar, compared to the gradually-typed calculus, is the addition of the *cast* construction $\langle\tau\rangle\ e$. This expression basically verifies that the value resulting from the evaluation of the expression e has type τ.

Lambda-abstractions now also contain cast annotations. The notation $\lambda^{\mathbb{I}}_{\langle\tau\rangle}\ x.\ e$ is simply a way to denote the cast of the function $\lambda^{\mathbb{I}}x.\ e$ to the type τ. However, as we will see later, we evaluate function casts in a lazy way to avoid unnecessary but costly η-expansions (as seen in [9]), and this notation is better suited to denote lazy evaluation of casts.

To simplify the language, we do not include *un-casted* lambda-abstractions of the form $\lambda^{\mathbb{I}}x.\ e$ in the grammar. However, such an expression can be defined as syntactic sugar for a lambda-abstraction with an identity cast, that is $\lambda^{\mathbb{I}}_{\langle\texttt{TypeOf}(\mathbb{I})\rangle}\ x.\ e$.

4.2 Typing and semantics

Our main objective is to prove the soundness of this cast language, which will allow us to prove that the execution resulting of the compilation of our gradually-typed language is safe. To achieve this, we start by defining a set of typing rules for the cast language. We do it before giving the semantics of the language since the definition of the semantics will use the definition of the type system (because of the presence of dynamic type-cases).

4.2.1 Typing rules.

The typing rules for the cast language are presented in Figure 2, and are mostly similar to the typing rules of our gradually-typed lambda calculus. The major difference reside in the rule (TCApp), where

$$\frac{\sigma = \texttt{TypeOf}(\mathbb{I}) \quad \sigma^{\Downarrow} \leq \tau^{\Uparrow} \quad \emptyset \vdash v : \tau'}{(\lambda^{\mathbb{I}}_{\langle \tau \rangle}\ x.\ e)v \mapsto \langle \tau @\ \tau' \rangle\ e[x := \langle gdom(\sigma) \rangle\ v]}(\text{RApp})$$

$$\frac{\emptyset \vdash v : \tau \quad \tau^{\Downarrow} \leq t}{((v \in t)?e_1 : e_2) \mapsto e_1}(\text{RCaseL}) \qquad \frac{\emptyset \vdash v : \tau \quad \tau^{\Downarrow} \not\leq t}{((v \in t)?e_1 : e_2) \mapsto e_2}(\text{RCaseR})$$

$$\frac{}{\langle \tau \rangle\ \lambda^{\mathbb{I}}_{\langle \tau' \rangle}\ x.\ e \mapsto \lambda^{\mathbb{I}}_{\langle \tau \rangle}\ x.\ e}(\text{RCstApp}) \qquad \frac{\emptyset \vdash c : s \quad s \leq \tau^{\Uparrow}}{\langle \tau \rangle\ c \mapsto c}(\text{RCstVal})$$

$$\frac{\tau^{\Downarrow} \not\leq \texttt{TypeOf}(\mathbb{I})^{\Uparrow}}{(\lambda^{\mathbb{I}}_{\langle \tau \rangle}\ x.\ e)v \mapsto \texttt{CastError}}(\text{RAppErr}) \qquad \frac{\emptyset \vdash c : s \quad s \not\leq \tau^{\Uparrow}}{\langle \tau \rangle\ c \mapsto \texttt{CastError}}(\text{RValErr})$$

$$\frac{e \mapsto e'}{E[e] \mapsto E[e']}(\text{RContext})$$

Figure 3: Small-step reduction semantics for the cast language

we added the condition $\tau_2{}^{\Uparrow} \leq gdom^{\Uparrow}(\tau_1)$. This condition is better explained in the following section on compilation, but the idea behind it is that we want to make sure that the application will succeed. That is, for an application to be well-typed, casts must have been inserted to (dynamically) verify that the function and the argument are *always* compatible (i.e. any interpretation of the type τ_2 must be a subtype of the domain of any interpretation of τ_1, that is, a subtype of $gdom^{\Uparrow}(\tau_1)$). Without this condition, $(\langle \texttt{?} \rangle\ 2)\ 3$ would be a well-typed term that reduces to 2 3, which is a ill-typed, stuck term.

The only other differences reside in the rule (TCCast) which determines the type of a casted expression, and the rule (TCAbs) where the function cast attached to a lambda-expression has to be taken into account. However, the rule (TCAbs) can be seen as a combination of the rule (TAbs) of the gradually-typed language and of the rule (TCCast).

At first sight, the rule (TCCast) may seem too lenient. According to this rule, if e is a well-typed expression, then the expression $\langle \tau \rangle\ e$ has type τ, independently of the type of e and τ. However, there are only two possible ways to evaluate $\langle \tau \rangle\ e$: either the cast succeeds and the expression reduces to a value of type τ, or the cast fails and the expression reduces to `CastError`. Therefore, whenever an expression of the form $\langle \tau \rangle\ e$ reduces to a value, this value will have type τ. Moreover, adding a premise such as $\sigma^{\Downarrow} \leq \tau^{\Uparrow}$ (where σ would be the type of e) may be too restrictive in some cases —as, it would only accept casts that can be statically verified—, and would jeopardize the very interest of dynamic typing, which aims at pushing such verifications at run-time, thus reducing the number of static type checks and giving more freedom to the programmer. Moreover, *downcasts* (such as casting a value of type `Int` to a value of type `Nat`) can be useful but do not satisfy this premise.

4.2.2 Operational Semantics

We now define the operational semantics of the cast language in a small-step style. The reduction rules are presented in Figure 3, and can be subdivided in three groups: applications (and conditional branching), typecasts, and errors. We evaluate the terms using a leftmost outermost weak reduction strategy; formally, evaluation contexts are defined using the following grammar:

$$E ::= \Box \mid Ee \mid vE \mid (E \in t)?e : e \mid \langle \tau \rangle\ E$$

The rules (RCstVal) and (RValErr), dealing with the casts of values that are not lambda-abstractions (i.e. constants), are quite straightforward. If the type of the constant is compatible with (that is, is a possible interpretation of) the type of the cast, then the cast succeeds and the constant is left unchanged. Otherwise, the cast fails and the expression reduces to a cast error.

The rules (RCaseL) and (RCaseR), dealing with type cases, are also straightforward. When evaluating the expression $((v \in t)?e_1 : e_2)$, if the value v can be typed with a subtype of t, then the first branch is chosen. Otherwise, the second branch is chosen.

The rule (RApp) is probably the most complicated one. Remember that we evaluate function casts in a lazy way. Therefore, when evaluating the expression $(\lambda^{\mathbb{I}}_{\langle\tau\rangle}\ x.\ e)v$, we need to check that it is possible to cast the lambda-expression to the type τ, hence the condition $\texttt{TypeOf}(\mathbb{I})^{\Downarrow} \leq \tau^{\Uparrow}$. Moreover, the expression e expects the variable x to be of type $gdom(\texttt{TypeOf}(\mathbb{I}))$, and not of type $gdom(\tau)$ (which is the domain of the casted function). Therefore, we do not simply substitute x by the value v but by the cast of v to the expected type.[3] Last but not least, the type of the function being cast to τ, the expected result type is $\tau @\ \tau'$ (where τ' is the type of v). However, the value returned by the expression $e[x := \langle gdom(\texttt{TypeOf}(\mathbb{I}))\rangle\ v]$ is of type $\texttt{TypeOf}(\mathbb{I}) @\ gdom(\texttt{TypeOf}(\mathbb{I}))$, hence the cast of the result.

Finally, the rule (RAppErr) simply handles the case where the lazy evaluation of a function cast fails, while the rule (RCstApp) "stores" a cast in a lambda abstraction, to evaluate it later. It is important to note that, in (RCstApp), the previous cast is simply erased. Storing all successive casts of a function would introduce a chain of casts when computing the result of an application, where the only relevant cast would be the outermost one (as it represents the type that is expected by the rest of the program). Therefore, removing those casts preserve the soundness of the execution while reducing the number of possible cast errors.[4] Moreover, function casts are only evaluated when necessary, that is, when (and only when) the function is applied. This avoids costly η-expansions that can hinder the performances of gradual typing, as shown in [12].

4.3 Soundness.

Having defined the semantics and the typing rules of our cast language, we now prove the soundness of its type system. This will allow us to prove the safety of the gradually typed language by simply proving that its compilation to the cast language *preserves* well-typedness.

To prove the soundness of the cast language, we first prove the two usual lemmas: *subject reduction* and *progress*. The subject reduction property (or type preservation) is usually stated as "if a term t_1 reduces to a term t_2 such that t_1 has type τ then t_2 also has type τ". However, in a gradual type system, the type of a term can change throughout its evaluation. For example, the term $(\lambda^{\{\ ?\rightarrow\ ?\}}x.\ x)2$ has type **?**, but reduces to 2 which has type `Int`. Moreover, stating the theorem as "if t_1 reduces to t_2, then the type of t_2 must be a subtype of the type of t_1" would not be a good solution. Due to the non-transitive behaviour of gradual subtyping, and to the fact that there is no subtitution property in gradual typing, this would make any induction-based reasoning difficult. Therefore, we prove a stronger statement, which says that the type of t_2 must *always* be a subtype of the type of t_1. That is, any interpretation of the type of t_2 must be a subtype of the type of t_1. If t_1 has type τ_1 and t_2 has type τ_2, this is equivalent to showing that ${\tau_2}^{\Uparrow} \leq {\tau_1}^{\Uparrow}$. This relation is transitive and implies gradual subtyping, which makes an induction-based proof possible.

Lemma 1. *(Subject Reduction) — For every terms t_1 and t_2, if $t_1 \mapsto t_2$ and $\emptyset \vdash t_1 : \tau_1$ then $\emptyset \vdash t_2 : \tau_2$ and ${\tau_2}^{\Uparrow} \leq {\tau_1}^{\Uparrow}$.*

Proof. The proof is done by case analysis over the rule used in the reduction $t_1 \mapsto t_2$. The term t_2 is then typed using the rules given in figure 2.

Due to the absence of substitution property in gradual typing, one should be careful to prove this property for every possible evaluation context. □

The progress lemma usually states that every well-typed term t that is not a value reduces to another term. However, in our cast system, it is possible to have well-typed terms that do not reduce to another term but to a cast error. For example, the term $\langle\texttt{Bool}\rangle\ 2$ reduces to `CastError` but has type `Bool`. Therefore, our version of the progress lemma includes the possibility that a well-typed term reduces to a cast error.

[3] As a side note, this substitution is likely to happen more than once in the expression e. Therefore, one should probably evaluate the cast of v lazily and remember the result to avoid doing the same cast multiple times.

[4] When evaluating a program, we try to succeed whenever possible, and fail only when necessary.

Lemma 2. *(Progress) — For every term t, if $\emptyset \vdash t : \tau$ then $t \in$ **Value** or $\exists t' \in$ **Term**, $t \mapsto t'$ or $t \mapsto$ `CastError`.*

Proof. The proof is done by structural induction over the term t, supposing that t is not a value. Given that t must be closed and the call-by-value semantics, the only non-trivial cases to consider are the application $f\ v$, the typecase $(v \in t)?e_1 : e_2$, and the cast $\langle\tau\rangle\ v$. All these terms reduce either to another term or to a cast error. □

Using the two previous lemmas, we can now prove the soundness of the evaluation of the cast language. The soundness theorem states that every well-typed, non-diverging[5] term of the cast language reduces either to a value or to a cast error.

Theorem 2. *(Soundness) — For every term t, if $\emptyset \vdash t : \tau$ then either t diverges or $\exists v \in$ **Value** such that $t \mapsto^* v$ or $t \mapsto^*$ `CastError`.*

Proof. This theorem is a direct consequence of the progress and the subject reduction properties. □

However, the soundness theorem in itself is not really meaningful. Indeed, if every term reduced to a cast error, the theorem would still hold, but the language would be useless. Therefore, we prove a more interesting corollary named *Static Safety* which states that, in the absence of casts and gradually-typed interfaces, the cast language behaves in the same way as the lambda calculus with set-theoretic types presented in [7] (ie. every well-typed term eventually reduces to a value).

Corollary 1. *(Static Safety) — For every term t, if t is well typed ($\emptyset \vdash t : s$), does not contain casts, and is fully annotated (ie. it does not contain gradual types) then either t diverges or $\exists v \in$ **Value** such that $t \mapsto^* v$.*

This corollary will allow us to prove that if a term of the gradually-typed lambda calculus is fully annotated, then the compiled term of the cast calculus reduces to a value. This is, in fact, the essence of gradual typing: one should be able to add or remove type annotations at its convenience while fully annotated code should retain maximal safety.

5 Compilation

In this section, we define the compilation procedure of the gradually-typed lambda calculus to the cast calculus. By proving that the compilation rules map well-typed terms of the graudally-typed calculus into well-typed terms of the cast calculus and using the soundness of the latter calculus, we can prove that the execution of well-typed gradually-typed terms is sound.

5.1 Compiling Applications

One of the most difficult problems that arises when one tries to define compilation rules for a gradually typed calculus is the compilation of applications. This part presents the problem as well as our solution, based on the approach presented in [9].

5.1.1 The cast insertion problem.

Let us consider simple (non set-thoeretic) types, and an application $e_1\ e_2$. We want to compile this application to a term of the cast calculus by adding typechecks where necessary. There are several possibilities:

- The type of e_1 is *necessarily* an arrow type, that is, e_1 is of type $\sigma \to \tau$; while every interpretation of the type of e_2 is a subtype of every interpretation of σ. In terms of extrema, if σ' is the type of e_2, we have $\sigma'^{\Uparrow} \leq \sigma^{\Downarrow}$. In this case, the application of e_1 to e_2 will *always* succeed, whatever the actual interpretations of e_1 and e_2 are. Therefore, there is no need to insert casts.

[5] Since this language is close to the simply typed lambda calculus, it is likely that the (strong) normalization property holds. However, due to time constraints, we do not have a proof of this property yet.

– Once again, the type of e_1 is known to be an arrow type, that is, e_1 has type $\sigma \to \tau$; but although the gradual type of e_2 is a (gradual) subtype of σ, there are interpretations of e_1 and e_2 such that the application fails. For exemple, consider the application *succ* x in a context where *succ* has type $\texttt{Int} \to \texttt{Int}$ and x has type $\texttt{?}$. This application is well-typed as $\texttt{?} \leq_{\text{G}} \texttt{Int}$. However, if x is replaced with `true`, then the application must raise a cast error. This is because `Bool` is a possible interpretation of $\texttt{?}$ that is not a subtype of `Int`. Therefore, in this case, we need to introduce a cast of the argument to the type expected by the function.

– Finally, there is the case where e_1 is not known to be a function. Without set-theoretic types, this only happens when e_1 has type $\texttt{?}$. In this case, we need to introduce a cast to verify that e_1 is indeed a function that can accept e_2 as parameter, that is, a function of type $\sigma \to \texttt{?}$ where σ is the type of e_2.

Those three cases describe, in essence, the compilation rules for applications presented in [9]. However, set-theoretic types make cast insertion even more difficult. Consider for example an expression f of type $(\texttt{Int} \to \texttt{Int}) \wedge \texttt{?}$. If we want to apply f to 0, which is of type `Int`, we do not need to insert a cast, as we *are sure* that f has type $\texttt{Int} \to \texttt{Int}$. However, if we want to apply f to *true*, which is of type `Bool`, we need to cast f to the type $\texttt{Bool} \to \texttt{?}$, so as to dynamically verify that f can accept values of type `Bool`.

5.1.2 Cast criteria.

To solve the problem presented in the last paragraph, we define two criteria that allow us to know whether we need to cast the argument, cast the function, or if we do not need to insert any cast. Note that it is not necessary to cast both the function and the argument. Indeed, as seen in the operational semantics of the cast language, casting a function automatically adds a cast of the argument when β-reducing the application.

The idea is to use the *safe* domain function, $gdom^{\Uparrow}$, presented in the first section. When compiling an application $e_1\ e_2$ (of the graudally-typed language) to a new application $e_1'\ e_2'$ (of the cast language), we want to ensure that e_2' can be passed to e_1'. Using $gdom^{\Uparrow}$, this means that all the possible interpretations of the type of e_2' should be a subtype of $gdom^{\Uparrow}(\tau)$, where τ is the type of e_1'.

From there, let us consider an application $e_1\ e_2$, such that e_1 compiles to e_1' of type τ_1 and e_2 compiles to e_2' of type τ_2. We also suppose that the application is well-typed (in the type system of the gradually-typed language) and that the compilation preserves types. Therefore, we know that ${\tau_2}^{\Downarrow} \leq gdom(\tau_1)$. We distinguish the following three cases:

– If we already have ${\tau_2}^{\Uparrow} \leq gdom^{\Uparrow}(\tau_1)$, then there is no need to insert casts. This is the case with all non-gradually, well-typed applications.

– If we have ${\tau_2}^{\Uparrow} \leq gdom(\tau_1)$ (but not the previous condition), this means that the application may succeed, provided e_1' has a type compatible with e_2'. Therefore, we need to insert a cast of e_1' to $\tau_2 \to (\tau_1 @\, \tau_2)$. For example, this is the case when applying a function of type $\tau = \texttt{?} \vee (\texttt{Int} \to \texttt{Int})$ to an argument of type `Bool`: in that case, we cast the function to the type $\texttt{Bool} \to (\tau @\, \texttt{Bool})$, where $\tau @\, \texttt{Bool}$ is actually equal to $\texttt{?}$.

– If none of the cases above applies, then this means that no value of type τ_1 will accept, unconditionally, a value of type τ_2. Therefore, the problem comes from the argument, which we need to cast to $gdom^{\Downarrow}(\tau_1)$. Moreover, to preserve the type of the compiled expression, we need to cast the result of the application to $\tau_1 @\, \tau_2$.

5.2 Compilation rules

Having presented how to solve the problems that arise when one tries to compile applications, we now give all the compilation rules of the gradually-typed language. These rules are presented in Figure 4. Once again, when compiling the typecases, if the first condition (resp. the second condition) does not hold, then the typecase is simply compiled to e_2' (resp. e_1').

The three compilation rules for applications simply implement the criteria described in the previous section, and add casts where necessary. As a sidenote, although the rule (CompApp2) introduces a

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : \tau}(\text{CompVar}) \qquad \frac{}{\Gamma \vdash c \rightsquigarrow c : B(c)}(\text{CompCst})$$

$$\frac{\forall \sigma_i \to \tau_i \in \mathbb{I}, \qquad \Gamma, x : \sigma_i \vdash e \rightsquigarrow e_i : \tau_i' \qquad e_i' = \begin{cases} e_i & \text{if } {\tau_i'}^{\Uparrow} \leq {\tau_i}^{\Downarrow} \\ \langle \tau_i \rangle\ e_i & \text{otherwise} \end{cases}}{\Gamma \vdash \lambda^{\mathbb{I}} x.\ e \rightsquigarrow (\lambda^{\mathbb{I}} x.(x \in {\sigma_1}^{\Uparrow})?\ e_1' : \cdots : (x \in {\sigma_i}^{\Uparrow})?\ e_i' : \cdots) : \texttt{TypeOf}(\mathbb{I})}(\text{CompLam})$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2 \qquad {\tau_2}^{\Uparrow} \leq gdom^{\Uparrow}(\tau_1)}{\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \tau_1 @ \tau_2}(\text{CompApp1})$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2 \qquad {\tau_2}^{\Uparrow} \leq gdom(\tau_1) \qquad {\tau_2}^{\Uparrow} \not\leq gdom^{\Uparrow}(\tau_1)}{\Gamma \vdash e_1\ e_2 \rightsquigarrow (\langle \tau_2 \to (\tau_1 @ \tau_2) \rangle\ e_1')\ e_2' : \tau_1 @ \tau_2}(\text{CompApp2})$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2 \qquad {\tau_2}^{\Uparrow} \not\leq gdom(\tau_1) \qquad {\tau_2}^{\Uparrow} \not\leq gdom^{\Uparrow}(\tau_1)}{\Gamma \vdash e_1\ e_2 \rightsquigarrow \langle \tau_1 @ \tau_2 \rangle\ (e_1'\ \langle gdom^{\Downarrow}(\tau_1) \rangle\ e_2') : \tau_1 @ \tau_2}(\text{CompApp3})$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : \tau \quad \begin{cases} \tau^{\Uparrow} \not\leq \neg t & \implies \Gamma \vdash e_1 \rightsquigarrow e_1' : \sigma_1 \\ \tau^{\Uparrow} \not\leq t & \implies \Gamma \vdash e_2 \rightsquigarrow e_2' : \sigma_2 \end{cases}}{\Gamma \vdash ((e \in t)?e_1 : e_2) \rightsquigarrow ((e' \in t)?e_1' : e_2') : \sigma_1 \vee \sigma_2}(\text{CompCase})$$

Figure 4: Compilation rules for the gradually-typed language

function cast, this cast will be evaluated lazily. In this rule, the expression e_1' will first be evaluated, resulting in a lambda expression which will store the cast. As such, the number of casts inserted at compile time do not hinder the performance of our calculus.

For example, consider the application $(\lambda^{\texttt{Int} \to \texttt{?}} x.\ \lambda^{\texttt{?} \to \texttt{?}} y.\ x + y)\ 3\ e$, where e has type `Int`. This application will be compiled to $\langle \texttt{Int} \to \texttt{?} \rangle\ ((\lambda^{\texttt{Int} \to \texttt{?}} x.\ \lambda^{\texttt{?} \to \texttt{?}} y.\ x + \langle \texttt{Int} \rangle\ y)\ 3)\ e$, which reduces to $(\langle \texttt{Int} \to \texttt{?} \rangle\ (\lambda^{\texttt{?} \to \texttt{?}} y.\ 3 + \langle \texttt{Int} \rangle\ y))\ e$. From there, the function cast is stored in the lambda expression to be evaluated later, after the evaluation of e: $(\lambda^{\texttt{?} \to \texttt{?}}_{\langle \texttt{Int} \to \texttt{?} \rangle}\ y.\ 3 + \langle \texttt{Int} \rangle\ y)\ e$.

What is more interesting is the compilation rule for lambda expressions. Basically, when compiling a function, we need to compile its body for every possible type of the argument found in the interface. However, for two different types, we may have two different compilations of the body, as different casts may be inserted at different places. Therefore, we need to somehow "merge" the different compilations to form the body of the compiled function. To achieve this, we make use of the fact that, in the interface of a function, all the domains of the arrows are disjoint. Every value is contained in at most one of the domains of an interface. Thus it is possible, dynamically, to check the type of the argument of the function and chose which of the compiled bodies should be executed. The rule (CompLam) simply compiles this typecase.

5.3 Safety and Soundness

Now that the compilation rules are defined, we want to prove that the underlying execution of the gradually-typed language is sound. We already proved the soundness of the execution of the cast calculus, that is, every (converging) well-typed term of this calculus reduces either to a value or to a cast error. Therefore, we just need to prove that the compilation of a well-typed term of the gradually-typed lambda calculus produces a well-typed term of the cast calculus. Formally, this can be stated as the following theorem:

Theorem 3. *(Soundness of Compilation) — For every term t of the gradually-typed calculus, if $\emptyset \vdash t : \tau$ and $t \rightsquigarrow t'$, then $\emptyset \vdash t' : \tau$.*

Proof. To prove this theorem, we first need to prove that for every term t of the GTLC, if $\Gamma \vdash t \rightsquigarrow t' : \tau$, then $\Gamma \vdash t' : \tau$. From there, the theorem is proved by structural induction over the term t and the rule used in the compilation of t. □

Using this theorem and the soundness of the cast language, the safety of the gradually-typed lambda calculus is an immediate result:

Corollary 2. *(Safety of the GTLC) — For every term t of the gradually-typed calculus, if $\emptyset \vdash t : \tau$, then either t diverges, or $\exists v \in$ **Value** such that $t \mapsto^* v$ or $t \mapsto^*$ `CastError`.*

Once again, it is likely that the strong normalization property holds in the cast language. In that case, it would also hold in the gradually-typed language, and a well-typed term of this language would never diverge.

Finally, we can also lift in the same way the *Static Safety* theorem from the cast calculus to the gradually-typed calculus. This is the most important result about our language as it shows that it actually respects the principles of gradual typing, and that it preserves the full power of static types.

Corollary 3. *(Static Safety of the GTLC) — For every term t of the gradually-typed calculus, if $\emptyset \vdash t : s$, and if t is fully annotated (ie. it does not contain gradual types), then either t diverges or $\exists v \in$ **Value** such that $t \mapsto^* v$.*

Proof. This is an immediate consequence of the static safety of the cast language, as the compilation of a fully-annotated term does not introduce casts. □

6 Conclusion and Future Work

In this report, we presented a full type system that mixes set-theoretic types with gradual typing. We also presented a language, derived from the lambda calculus, that uses this type system. Finally, we proposed a cast calculus that integrates set-theoretic types and gave a compilation system that builds and execute a term of the cast calculus from a term of the gradually typed caclulus.

We showed classic properties of our type system, such as its soundness, but more importantly, we showed that it respects the principles of gradual typing by preserving the full power of static types when required. That is, programmers can chose to fully annotate their code —without any compromise on safety or performance— or to omit certain type annotations.

As a secondary contribution, we proposed a way to evaluate function casts lazily. The reduction rules we provide are independent from our set-theoretic type system and should apply to any implementation of a gradually typed language. By removing costly function expansions, this may significantly improve the performances of gradually typed languages.

A logical continuation of this work would be to implement and test a gradually and set-theoretically typed language. There have been some really interesting approaches, such as Flow [1] or TypeScript [2], and it would be interesting to compare our approach to theirs. Given that we also proposed a system that evaluates cast in a lazy way, which may alleviate the performance problems shown in [12], it would be interesting to add lazy evaluation of casts to the languages used for the benchmarks in this paper to evaluate the performances of our system. Finally, a more ambitious goal would be to find a more general theory, such as the one presented in [8], that provides a systemic way to add gradual typing to set-theoretic type systems.

References

[1] Flow Language. `https://flowtype.org/`.

[2] Typescript Language. `https://www.typescriptlang.org/`.

[3] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: an xml-centric general-purpose language. In *ACM SIGPLAN Notices*, volume 38, pages 51–63. ACM, 2003.

[4] Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. Polymorphic functions with set-theoretic types: Part 2: Local type inference and type reconstruction. *ACM SIGPLAN Notices*, 50(1):289–302, 2015.

[5] Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. *ACM SIGPLAN Notices*, 49(1):5–17, 2014.

[6] Matteo Cimini and Jeremy G Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 443–455. ACM, 2016.

[7] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM (JACM)*, 55(4):19, 2008.

[8] Ronald Garcia, Alison M Clark, and Éric Tanter. Abstracting gradual typing. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, 2016.

[9] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[10] Jeremy G Siek and Sam Tobin-Hochstadt. The recursive union of some gradual types. In *A List of Successes That Can Change the World*, pages 388–410. Springer, 2016.

[11] Jeremy G Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 symposium on Dynamic languages*, page 7. ACM, 2008.

[12] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *ACM SIGPLAN Notices*, volume 51, pages 456–468. ACM, 2016.