université PARIS DIDEROT PARIS 7

Gradual Set-Theoretic Types

Adding stronger static guarantees to gradual type systems

Victor Lanvin

ENS Paris-Saclay, Cachan, France

école normale supérieure paris-saclay

1. Gradual typing [3]

- **Goal**: the programmer can control precisely how much type-checking is done statically versus dynamically.
- **Pros**:
  - Rapid prototyping
  - The code is more flexible and easier to maintain
- **Cons**:
  - Often reduced to an all-or-nothing approach
  - Performance issues
- **Example [4]**:

```
let succ : Int -> Int = ...
let notb : Bool -> Bool = ...
let f (condition : Bool) (x : ?) : ? =
  if condition then
    succ x
  else
    notb x
```

2. Set-theoretic types [1]

- **Goal**: interprete types as sets of values and use set-theoretic operations (union, intersection, etc.) on types.
- **Pros**:
  - Powerful type system (stronger than simple types)
  - Subtyping is reduced to set containment
  - Concepts such as overloading can be precisely typed
- **Cons**:
  - Type reconstruction is undecidable
  - The syntactic overhead hinders development speed
- **Example**:

```
let f (condition : Bool) (x : (Int | Bool))
      : (Int | Bool) =
  if condition then
    succ ⟨Int⟩ x    (* Typecast *)
  else
    notb ⟨Bool⟩ x
```

3. Motivation

Advantages of mixing gradual types and set-theoretic types

- Finer grained transition between static typing and dynamic typing
- Reduce the syntactic overhead of set-theoretic types
- Stronger static guarantees on gradually-typed programs

Reject ill-typed applications, without the need for explicit casts:

```
let f (condition : Bool) (x : (Int | Bool) & ?)
      : (Int | Bool) =
  if condition then
    succ x
  else
    notb x
```

More precise return types, depending on the type of the previous parameters:

```
let f : (Bool -> (Int & ?) -> Int)
      & (Bool -> (Bool & ?) -> Bool) =
  fun condition x ->
  if condition then
    succ x
  else
    notb x
```

Remark. The previous type is equivalent to the following type:

```
Bool ->
   (((Int & ?) -> Int) & ((Bool & ?) -> Bool))
```

4. Semantic interpretation and subtyping

Syntax.

$$t \in \text{StaticTypes} ::= t \vee t \mid t \wedge t \mid \neg t \mid t \to t \mid b \mid \texttt{Empty} \mid \texttt{Any}$$
$$\tau \in \text{GradualTypes} ::= \textbf{?} \mid \tau \vee \tau \mid \tau \wedge \tau \mid \neg t \mid \tau \to \tau \mid b \mid \texttt{Empty} \mid \texttt{Any}$$
$$b ::= \texttt{Int} \mid \texttt{Bool} \mid \ldots$$

Abstract interpretation [2]. A gradual type is interpreted as a set of static types. Functions can then be *lifted* from static types to gradual types:

$$\begin{tikzcd} \text{GType} \arrow[r, "\widetilde{f}"] \arrow[d, "\gamma"'] & \text{GType} \\ \mathcal{P}(\text{SType}) \arrow[r, "f"'] & \mathcal{P}(\text{SType}) \arrow[u, "\alpha"'] \end{tikzcd}$$

$$\gamma(?) = \text{SType}$$
$$\gamma(? \vee \texttt{Int}) = \{t \vee \mathit{Int} \mid t \in \text{SType}\}$$

Minimal and maximal concretisations are obtained by substituting all occurences of ? by `Empty` or `Any`, depending on their position (covariant or contravariant). For example,

$$\mathit{max}((? \to ?) \vee ?) = (\texttt{Empty} \to \texttt{Any}) \vee \texttt{Any}$$

Gradual subtyping for set-theoretic types reduces to static (semantic) subtyping:

$$\begin{aligned} \sigma \,\widetilde{\leq}\, \tau &\iff \exists (s,t) \in \gamma(\sigma) \times \gamma(\tau), s \leq t \\ &\iff \mathit{min}(\sigma) \leq \mathit{max}(\tau) \end{aligned}$$

5. Typing applications

By only partially concretizing gradual types we can obtain a finite concretisation function γ_f and a set-theoretic abstraction function:

$$\alpha_f : \mathcal{P}_f(\text{GType}) \to \text{GType}$$
$$\alpha_f(S) = \bigvee_{\sigma \in S} \sigma$$

The domain and result type of an application are defined by lifting their static counterparts using the finite operations.

$\mathit{dom} : \text{GType} \to \text{SType}$	$\circ : \text{GType} \times \text{GType} \to \text{GType}$
$\mathit{dom}((\texttt{Int} \to \texttt{Int}) \wedge ?) = \texttt{Any}$	$((\texttt{Int} \to \texttt{Int}) \wedge ?) \circ \texttt{Int} = \texttt{Int} \wedge ?$
$\mathit{dom}((\texttt{Int} \to \texttt{Int}) \vee ?) = \texttt{Int}$	$((\texttt{Int} \to \texttt{Int}) \wedge ?) \circ \texttt{Bool} = ?$

6. Compiling applications

Two domains are defined for every function type: a *safe* domain $\mathit{dom}^{\mathcal{S}}$ and a *possible* domain $\mathit{dom}^{\mathcal{P}}$:

$\mathit{dom}^{\mathcal{P}}(?) = \texttt{Any}$	as ? can *possibly* represent any function.
$\mathit{dom}^{\mathcal{S}}(?) = \texttt{Empty}$	as ? is *not always* a function.
$\mathit{dom}^{\mathcal{S}}(? \to ?) = \texttt{Any}$	as ? → ? can *always* be applied to *any value*.

Casts are inserted if the type of the argument a function is not *always* in its safe domain:

- If $\Gamma \vdash \texttt{f} : (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$ and $\Gamma \vdash \texttt{x} : ?$
  Then `f x` compiles to $\texttt{f}\langle \texttt{Int} \vee \texttt{Bool}\rangle\ \texttt{x}$
- If $\Gamma \vdash \texttt{f} : \textbf{?} \wedge (\texttt{Bool} \to \texttt{Bool})$ and $\Gamma \vdash \texttt{x} : \texttt{Int}$
  Then `f x` compiles to $(\langle \texttt{Int} \to ?\rangle\ \texttt{f})\ \texttt{x}$

Results:

- Compilation is type-preserving
- Every well-typed term without gradual types reduces to a value
- Every well-typed term reduces to a value or a cast error

7. Bibliography

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

[2] R. Garcia, A. M. Clark, and É. Tanter. Abstracting gradual typing. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, 2016.

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

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

Victor Lanvin

victor.lanvin@ens-cachan.fr