Gradual Set-Theoretic Types

Reconcile static and dynamic typing (and polymorphism)

Victor Lanvin
September 27, 2017
Introduction and Example
Both approaches have pros and cons:

<table>
<thead>
<tr>
<th>Static typing</th>
<th>Dynamic typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safe</td>
<td>Unsafe</td>
</tr>
<tr>
<td>Restrictive</td>
<td>Lenient</td>
</tr>
<tr>
<td>Syntactically heavy</td>
<td>Fast to write</td>
</tr>
</tbody>
</table>
Both approaches have pros and cons:

<table>
<thead>
<tr>
<th>Static typing</th>
<th>Dynamic typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safe</td>
<td>Unsafe</td>
</tr>
<tr>
<td>Restrictive</td>
<td>Lenient</td>
</tr>
<tr>
<td>Syntactically heavy</td>
<td>Fast to write</td>
</tr>
</tbody>
</table>

→ Gradual typing tries to get the best of both worlds.
A first problematic example

let succ : Int -> Int = ...  
let not : Bool -> Bool = ...  

let f (condition : Bool) (x : ...) =  
  if condition then  
    succ x  
  else  
    not x
let succ : Int -> Int = ...  
let not : Bool -> Bool = ... 

let f (condition : Bool) (x : ?) : ? = 
  if condition then 
    succ x 
  else 
    not x 

→ Cannot be typed with simple types, but valid with gradual types.
A first problematic example

let succ : Int -> Int = ...
let not : Bool -> Bool = ...

let f (condition : Bool) (x : ?) : ? =
  if condition then
    succ x
  else
    not x

→ Cannot be typed with simple types, but valid with gradual types.
→ What if \(x\) is a string?
Set-theoretic version:

```ocaml
define f (condition : Bool) (x : (Int | Bool)) : (Int | Bool) =
  if condition then
    if \ x \in \ Int then succ x else assert false
  else
    if \ x \in \ Bool then not x else assert false
```

Syntactically heavy, but safe (x cannot be a string)
Set-theoretic version:

```haskell
let f (condition : Bool) (x : (Int | Bool)) : (Int | Bool) =
  if condition then
    if x ∈ Int then succ x else assert false
  else
    if x ∈ Bool then not x else assert false
```

→ Syntactically heavy, but safe (x cannot be a string)
Even better solution: gradual set-theoretic types

Getting the best of both worlds:

```ocaml
let f (condition : Bool) (x : (Int | Bool) & ?) : (Int | Bool) =
  if condition then
    succ x
  else
    not x
```
Even better solution: gradual set-theoretic types

Getting the best of both worlds:

```ocaml
let f (condition : Bool) (x : (Int | Bool) & ?) : (Int | Bool) =
    if condition then
        succ x
    else
        not x

→ Cannot be applied to something else than an integer or a boolean, and has a precise return type
→ Syntactically straightforward
Visualizing the difference

\[ \Gamma = \{ \text{succ} : \text{Int} \rightarrow \text{Int}; \ x : \text{Int} \lor \text{Bool} \} \]
\[ \Gamma = \{ \text{succ} : \text{Int} \rightarrow \text{Int}; \ x : \text{Int} \lor \text{Bool} \} \]

\[ \Gamma \vdash \text{succ } x : \text{Int} \]
Visualizing the difference

\[ \Gamma = \{ \text{succ} : \text{Int} \to \text{Int}; \ x : \text{Int} \lor \text{Bool} \} \]
\[ \Gamma = \{ \text{succ} : \text{Int} \to \text{Int}; \; x : \text{Int} \lor \text{Bool} \} \]
Visualizing the difference

$$\Gamma = \{\text{succ} : \text{Int} \rightarrow \text{Int}; \ x : (\text{Int} \lor \text{Bool}) \land ?\}$$
Visualizing the difference

\[ \Gamma = \{\text{succ} : \text{Int} \to \text{Int}; \, x : (\text{Int} \lor \text{Bool}) \land ?\} \]

\[ \Gamma \vdash \text{succ} \, x : \text{Int} \]
Visualizing the difference

\[ \Gamma = \{ \text{succ} : \text{Int} \rightarrow \text{Int}; \ x : (\text{Int} \lor \text{Bool}) \land ? \} \]
Visualizing the difference

\[ \Gamma = \{ \text{succ} : \text{Int} \to \text{Int}; \ x : (\text{Int} \lor \text{Bool}) \land ? \} \]
\[ \Gamma = \{ \text{succ : Int} \rightarrow \text{Int}; \ x : (\text{Int} \lor \text{Bool}) \land ? \} \]
Action plan

The plan contains four steps. We define:

– the semantics of polymorphic gradual set-theoretic types
– a lambda-calculus (GTLC) and its type system
– a cast language (CL) and its reduction rules
– a compilation procedure from the GTLC to the CL
Types, Semantics and Typing
Syntax: types and language

- Static types:

$$t \in T_t ::= \alpha \mid b \mid t \times t \mid t \to t \mid t \lor t \mid t \land t \mid \neg t \mid \text{Empty} \mid \text{Any}$$

- Gradual types:

$$\tau \in T_\tau ::= \? \mid \alpha \mid b \mid t \times t \mid t \to t \mid t \lor t \mid t \land t \mid \neg t \mid \text{Empty} \mid \text{Any}$$

- Language:

$$e \in \text{Terms} ::= x \mid c \mid \lambda x: \tau. \ e \mid \lambda x. \ e \mid e \ e \mid (e, e) \mid \pi_i \ e \mid \text{let } x = e \text{ in } e$$
How it is (usually) done

Usual gradual type systems:

\[ \Gamma \vdash e_1 : \sigma_1 \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma_2 \quad \sigma_2 \lesssim \sigma_1 \]

\[ \Gamma \vdash e_1 \; e_2 : \tau \]
Usual gradual type systems:

\[
\begin{aligned}
&\Gamma \vdash e_1 : \sigma_1 \to \tau \quad \Gamma \vdash e_2 : \sigma_2 \quad \sigma_2 \simleq \sigma_1 \\
&\Gamma \vdash e_1 \ e_2 : \tau
\end{aligned}
\]

Several problems:

- Ad-hoc, inlined subtyping check
How it is (usually) done

Usual gradual type systems:

\[
\frac{\Gamma \vdash e_1 : \sigma_1 \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma_2 \quad \sigma_2 \lessdot \sigma_1}{\Gamma \vdash e_1 \; e_2 : \tau}
\]

Several problems:

- Ad-hoc, inlined subtyping check
- \( \lessdot \) is not transitive: \( ? \lessdot \tau \) and \( \tau \lessdot ? \) for every type \( \tau \)
Question: can we deduce the semantics of gradual types from the semantics of static types, while preserving the transitivity of subtyping?
Question: can we deduce the semantics of gradual types from the semantics of static types, while preserving the transitivity of subtyping?

Observations:

– ? is not a base type ($\exists \land \text{Int} \neq \text{Empty}$ for example)
– Two occurrences of ? in the same type can behave differently
Question: can we deduce the semantics of gradual types from the semantics of static types, while preserving the transitivity of subtyping?

Observations:
- ? is not a base type (\(\text{?} \land \text{Int} \neq \text{Empty}\) for example)
- Two occurrences of ? in the same type can behave differently

Solution: interpret every ? by a different, particular type variable
Gradual subtyping is now defined using static subtyping and substitutions of \texttt{?} by variables
Gradual subtyping is now defined using static subtyping and substitutions of $\_\_\_\_$ by variables.

Examples:

$$\_\_\_\_\_\_\_\_ \leq \_\_\_\_\_\_\_ \lor \text{Int} \quad \text{since} \quad X \leq_T X \lor \text{Int}$$
Gradual subtyping is now defined using *static* subtyping and substitutions of $\_\_\_\_\_$ by variables

Examples:

\[
\begin{align*}
\_\_\_\_\_\_\_ \leq \_\_\_\_\_\_ \lor \text{Int} & \quad \text{since } X \leq_T X \lor \text{Int} \\
\_\_\_\_\_\_ \lor \_\_\_\_\_\_ \leq \_\_\_\_\_\_ & \quad \text{since } X \lor X \simeq X
\end{align*}
\]
Gradual subtyping is now defined using static subtyping and substitutions of $\pi$ by variables.

Examples:

\[
\pi \leq \pi \lor \text{Int} \quad \text{since} \quad X \leq_T X \lor \text{Int} \\
\pi \lor \pi \leq \pi \quad \text{since} \quad X \lor X \simeq X \leq_T X
\]
Gradual subtyping is now defined using static subtyping and substitutions of ? by variables

Examples:

\[ \, ? \leq ? \lor \text{Int} \quad \text{since} \quad X \leq_T X \lor \text{Int} \]

\[ ? \lor ? \leq ? \quad \text{since} \quad X \lor X \simeq X \leq_T X \]

This relation is transitive!
Subtyping allows us to decrease the precision of a type without crossing the static/dynamic border.
Subtyping allows us to *decrease* the precision of a type *without* crossing the static/dynamic border.

However, the point of gradual typing is to *allow an increase in precision by crossing this border*.

→ *succ x* should be well typed if *x : ?*, which is less precise than *Int*. 

---

**Type semantics: materialization**

**Type semantics: materialization**
Subtyping allows us to decrease the precision of a type without crossing the static/dynamic border.

However, the point of gradual typing is to allow an increase in precision by crossing this border.

→ succ x should be well typed if x : ?, which is less precise than Int.

Hence the definition of a “materialization” relation:

\[ ? \ll \tau \quad (? \land \text{Int}) \rightarrow ? \ll (\text{Int} \land \text{Int}) \rightarrow \text{Bool} \]
Materialization is also transitive!

⇒ We can use both subtyping and materialization in subsumption rules
Materialization is also transitive!

⇒ We can use both subtyping and materialization in subsumption rules

\[
\begin{align*}
\text{[SUB]} & \quad \frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau} \quad \frac{\tau' \leq \tau}{\Gamma \vdash e : \tau} \\
\text{[MAT]} & \quad \frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau} \quad \frac{\tau' \lessdot \tau}{\Gamma \vdash e : \tau} \\
\text{[APPL]} & \quad \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 \ e_2 : \tau}
\end{align*}
\]
What we have now

Materialization is also transitive!

⇒ We can use both subtyping and materialization in subsumption rules

\[
\begin{align*}
\text{[SUB]} & \quad \frac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash e : \tau} \\
\text{[MAT]} & \quad \frac{\Gamma \vdash e : \tau' \quad \tau' \preccurlyeq \tau}{\Gamma \vdash e : \tau} \\
\text{[APPL]} & \quad \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 \ e_2 : \tau}
\end{align*}
\]

We got the modus ponens back!
Compilation
Objectives

- Compile the GTLC into a cast language to ensure that untyped values are not misused

- Have a declarative compilation system and an algorithmic one

- Make sure that compilation preserve types
Objectives

- Compile the GTLC into a cast language to ensure that untyped values are not misused

\[(\lambda x : ?.x + 1) \ False\] should fail

\[(\lambda x : ?.x + 1) \ 3\] should reduce to 4
Objectives

– Compile the GTLC into a cast language to ensure that untyped values are not misused
  
  \((\lambda x : ?.x + 1) \text{False}\) should fail
  
  \((\lambda x : ?.x + 1) 3\) should reduce to 4

– Have a declarative compilation system and an algorithmic one
Objectives

– Compile the GTLC into a cast language to ensure that untyped values are not misused

\[(\lambda x:?.x + 1)\text{ False} \quad \text{should fail}\]
\[(\lambda x:?.x + 1)\text{ 3} \quad \text{should reduce to 4}\]

– Have a declarative compilation system and an algorithmic one
– Make sure that compilation preserve types
Cast language

\[ E ::= \text{x} | \text{c} | \lambda^{\tau \to \tau} \text{x} \cdot E | E \ E | (E, E) | \pi_i \ E | \]

\text{let} \ x = E \ \text{in} \ E | E[\alpha_i := \tau_i]_{i=1}^{p} | E^{\langle \tau \rangle} \]
GTLC + Casts:

\[ E ::= x \mid c \mid \lambda \rightarrow^{\tau} \cdot x \mid E \mid (E, E) \mid \pi_i E \mid \text{let } x = E \text{ in } E \mid E[\alpha_i \leftarrow \tau_i]_{i=1}^n \mid E\langle\tau\rangle \]

Example:

\[ (\lambda \rightarrow^{\text{Int}} x. x\langle\text{Int}\rangle + 1) \text{ true } \rightarrow (\text{true}\langle\text{Int}\rangle) + 1 \]

\[ \rightarrow \text{CastError} \]
Main idea: to every application of the materialization rule corresponds a cast
Main idea: to every application of the materialization rule corresponds a cast

\[ \begin{array}{c}
\text{[MAT]} & \frac{\Gamma \vdash e \leadsto E : \tau' \quad \tau' \preceq \tau}{\Gamma \vdash e \leadsto E\langle \tau \rangle : \tau}
\end{array} \]
Main idea: to every application of the materialization rule corresponds a cast

\[
\begin{align*}
\text{[MAT]} & \quad \Gamma \vdash e \rightsquigarrow E : \tau' \quad \tau' \preccurlyeq \tau \\
& \quad \Gamma \vdash e \rightsquigarrow E\langle \tau \rangle : \tau
\end{align*}
\]

\[
\begin{align*}
\text{[APP]} & \quad \Gamma \vdash e_1 \rightsquigarrow E_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 \rightsquigarrow E_2 : \tau' \\
& \quad \Gamma \vdash e_1 \ e_2 \rightsquigarrow E_1 \ E_2 : \tau
\end{align*}
\]
Main idea: to every application of the materialization rule corresponds a cast

\[
\begin{align*}
\text{[MAT]} & \quad \Gamma \vdash e \rightsquigarrow E : \tau' \quad \tau' \preceq \tau \\
& \quad \Gamma \vdash e \rightsquigarrow E(\tau) : \tau
\end{align*}
\]

\[
\begin{align*}
\text{[APP]} & \quad \Gamma \vdash e_1 \rightsquigarrow E_1 : \tau' \rightarrow \tau \\
& \quad \Gamma \vdash e_2 \rightsquigarrow E_2 : \tau' \\
& \quad \Gamma \vdash e_1 \ e_2 \rightsquigarrow E_1 \ E_2 : \tau
\end{align*}
\]

1:1 correspondence with the typing rules
Conclusion
Some results

- Definition of a type system that is more intuitive and closer to the usual ones.

- Soundness of the compilation:
  Every well-typed term of the GTLC compiles to a well-typed term of the same type in the cast language.

- Soundness of the CL:
  Every well-typed term of the CL reduces to a value, an error, or diverges.
Future work

- Conservativity result, comparison to non set-theoretic approaches
- Can we reuse the constraint-solving algorithm from polymorphic set-theoretic types?
- Maybe an implementation?