Gradual Set-Theoretic Types

Reconcile static and dynamic typing (and polymorphism)

Victor Lanvin

September 27, 2017

Introduction and Example

Static typing vs dynamic typing

Both approaches have pros and cons:

Static typing	Dynamic typing
Safe	Unsafe
Restrictive	Lenient
Syntactically heavy	Fast to write

Static typing vs dynamic typing

Both approaches have pros and cons:

Static typing	Dynamic typing
Safe	Unsafe
Restrictive	Lenient
Syntactically heavy	Fast to write

$\Longrightarrow$ 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
```

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.

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?

Solution: set-theoretic types

Set-theoretic version:

```
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
```

Solution: set-theoretic types

Set-theoretic version:

```
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:

```
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:

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

$\rightarrow$ Cannot be applied to something else than an integer or a boolean, and has a precise return type

$\rightarrow$ Syntactically straightforward

Visualizing the difference

$$\Gamma = \{\texttt{succc} : \texttt{Int} \to \texttt{Int};\ \ \texttt{x} : \texttt{Int} \vee \texttt{Bool}\}$$

Visualizing the difference

$$\Gamma = \{\mathtt{succ} : \mathtt{Int} \to \mathtt{Int};\ \mathtt{x} : \mathtt{Int} \vee \mathtt{Bool}\}$$

$$\Gamma \vdash \mathtt{succ\ \ x} : \mathtt{Int}$$

Visualizing the difference

$$\Gamma = \{\mathtt{succ} : \mathtt{Int} \to \mathtt{Int};\ \mathtt{x} : \mathtt{Int} \vee \mathtt{Bool}\}$$

$$\text{[App]}\ \dfrac{\text{[Var]}\ \dfrac{}{\Gamma \vdash \mathtt{succ} \colon \mathtt{Int} \to \mathtt{Int}} \qquad \Gamma \vdash \mathtt{x} \colon \mathtt{Int}}{\Gamma \vdash \mathtt{succ}\ \ \mathtt{x} \colon \mathtt{Int}}$$

Visualizing the difference

$$\Gamma = \{\texttt{succ} : \texttt{Int} \to \texttt{Int};\ \texttt{x} : \texttt{Int} \vee \texttt{Bool}\}$$

$$[\textsc{App}]\ \dfrac{[\textsc{Var}]\ \dfrac{}{\Gamma \vdash \texttt{succ}\colon \texttt{Int} \to \texttt{Int}} \qquad [\textsc{Sub}]\ \dfrac{\Gamma \vdash \texttt{x}\colon \texttt{Int} \vee \texttt{Bool} \qquad \color{red}{\texttt{Int} \vee \texttt{Bool} \not\leq \texttt{Int}}}{\Gamma \vdash \texttt{x}\colon \texttt{Int}}}{\Gamma \vdash \texttt{succ}\ \ \texttt{x}\colon \texttt{Int}}$$

Visualizing the difference

$$\Gamma = \{\texttt{succ} : \texttt{Int} \to \texttt{Int};\ \ \texttt{x} : (\texttt{Int} \vee \texttt{Bool}) \wedge ?\}$$

Visualizing the difference

$$\Gamma = \{\texttt{succ} : \texttt{Int} \to \texttt{Int};\ \texttt{x} : (\texttt{Int} \vee \texttt{Bool}) \wedge ?\}$$

$$\Gamma \vdash \texttt{succ x} : \texttt{Int}$$

Visualizing the difference

$$\Gamma = \{\texttt{succ} : \texttt{Int} \to \texttt{Int};\ \texttt{x} : (\texttt{Int} \vee \texttt{Bool}) \wedge ?\}$$

$$[\textsc{App}]\ \dfrac{[\textsc{Var}]\ \dfrac{}{\Gamma \vdash \texttt{succ} \colon \texttt{Int} \to \texttt{Int}} \qquad \Gamma \vdash \texttt{x} \colon \texttt{Int}}{\Gamma \vdash \texttt{succ}\ \ \texttt{x} \colon \texttt{Int}}$$

Visualizing the difference

$$\Gamma = \{\mathtt{succ} : \mathtt{Int} \to \mathtt{Int};\ \mathtt{x} : (\mathtt{Int} \vee \mathtt{Bool}) \wedge ?\}$$

$$[\textsc{App}]\ \dfrac{[\textsc{Var}]\ \dfrac{}{\Gamma \vdash \mathtt{succ} \colon \mathtt{Int} \to \mathtt{Int}} \qquad [\textsc{Mat}]\ \dfrac{\Gamma \vdash \mathtt{x} \colon ? \qquad ? \preccurlyeq \mathtt{Int}}{\Gamma \vdash \mathtt{x} \colon \mathtt{Int}}}{\Gamma \vdash \mathtt{succ}\ \ \mathtt{x} \colon \mathtt{Int}}$$

Visualizing the difference

$$\Gamma = \{\mathtt{succ} : \mathtt{Int} \to \mathtt{Int};\ \ \mathtt{x} : (\mathtt{Int} \vee \mathtt{Bool}) \wedge ?\}$$

$$\dfrac{\dfrac{}{\Gamma \vdash \mathtt{succ} \colon \mathtt{Int} \to \mathtt{Int}}\,[\textsc{Var}] \qquad \dfrac{\dfrac{(\mathtt{Int} \vee \mathtt{Bool}) \wedge ? \leq ? \qquad \Gamma \vdash \mathtt{x} \colon (\mathtt{Int} \vee \mathtt{Bool}) \wedge ?}{\Gamma \vdash \mathtt{x} \colon ?}\,[\textsc{Sub}] \qquad ? \preccurlyeq \mathtt{Int}}{\Gamma \vdash \mathtt{x} \colon \mathtt{Int}}\,[\textsc{Mat}]}{\Gamma \vdash \mathtt{succ}\ \ \mathtt{x} \colon \mathtt{Int}}\,[\textsc{App}]$$

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:

$$t \in \mathcal{T}_t ::= \alpha \mid b \mid t \times t \mid t \to t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$$

- Gradual types:

$$\tau \in \mathcal{T}_\tau ::= \; \texttt{?} \mid \alpha \mid b \mid t{\times}t \mid t \to t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$$

- Language:

$$\begin{aligned} e \in \texttt{Terms} ::={} & x \mid c \mid \lambda x\colon \tau.\; e \mid \lambda x.\, e \mid e\; e \mid \\ & (e, e) \mid \pi_i\; e \mid \text{let } x = e \text{ in } e \end{aligned}$$

How it is (usually) done

Usual gradual type systems:

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 : \sigma_1 \to \tau \qquad \Gamma \vdash e_2 : \sigma_2 \qquad \sigma_2 \tilde{\leq} \sigma_1}{\Gamma \vdash e_1\ e_2 : \tau}$$

How it is (usually) done

Usual gradual type systems:

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 : \sigma_1 \to \tau \qquad \Gamma \vdash e_2 : \sigma_2 \qquad \sigma_2 \tilde{\leq} \sigma_1}{\Gamma \vdash e_1\ e_2 : \tau}$$

Several problems:

– Ad-hoc, inlined subtyping check

How it is (usually) done

Usual gradual type systems:

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 : \sigma_1 \to \tau \qquad \Gamma \vdash e_2 : \sigma_2 \qquad \sigma_2 \tilde{\leq} \sigma_1}{\Gamma \vdash e_1\ e_2 : \tau}$$

Several problems:

- Ad-hoc, inlined subtyping check
- $\tilde{\leq}$ is not transitive: $?\tilde{\leq}\tau$ and $\tau\tilde{\leq}?$ for every type τ

Type semantics: interpretation

Question: can we deduce the semantics of gradual types from the semantics of static types, while preserving the transitivity of subtyping?

Type semantics: interpretation

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 ($? \wedge \texttt{Int} \neq \texttt{Empty}$ for example)
- Two occurrences of `?` in the same type can behave differently

Type semantics: interpretation

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 ($\texttt{?} \wedge \texttt{Int} \neq \texttt{Empty}$ for example)
- Two occurrences of `?` in the same type can behave differently

Solution: interpret every `?` by a different, particular type variable

Type semantics: subtyping

Gradual subtyping is now defined using *static* subtyping and substitutions of $?$ by variables

Type semantics: subtyping

Gradual subtyping is now defined using *static* subtyping and substitutions of `?` by variables

Examples:

$$\texttt{?} \leq \texttt{?} \vee \texttt{Int} \qquad \text{since } X \leq_T X \vee \texttt{Int}$$

Type semantics: subtyping

Gradual subtyping is now defined using *static* subtyping and substitutions of `?` by variables

Examples:

$$\texttt{?} \leq \texttt{?} \vee \texttt{Int} \qquad \text{since } X \leq_T X \vee \texttt{Int}$$

$$\texttt{?} \vee \texttt{?} \leq \texttt{?} \qquad \text{since } X \vee X \simeq X$$

Type semantics: subtyping

Gradual subtyping is now defined using *static* subtyping and substitutions of `?` by variables

Examples:

$$\texttt{?} \leq \texttt{?} \vee \texttt{Int} \qquad \text{since } X \leq_T X \vee \texttt{Int}$$

$$\texttt{?} \vee \texttt{?} \leq \texttt{?} \qquad \text{since } X \vee X \simeq X \leq_T X$$

Type semantics: subtyping

Gradual subtyping is now defined using *static* subtyping and substitutions of `?` by variables

Examples:

$$\texttt{?} \leq \texttt{?} \vee \texttt{Int} \qquad \text{since } X \leq_T X \vee \texttt{Int}$$

$$\texttt{?} \vee \texttt{?} \leq \texttt{?} \qquad \text{since } X \vee X \simeq X \leq_T X$$

This relation is transitive!

Type semantics: materialization

Subtyping allows us to *decrease* the precision of a type *without crossing the static/dynamic border*

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*

$\rightarrow$ *succ* x should be well typed if $x : \texttt{?}$, which is less precise than `Int`.

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*

$\rightarrow$ *succ x* should be well typed if $x : \texttt{?}$, which is less precise than $\texttt{Int}$.

Hence the definition of a "materialization" relation:

$$\texttt{?} \preccurlyeq \tau \qquad\qquad (\texttt{?} \wedge \texttt{Int}) \rightarrow \texttt{?} \preccurlyeq (\texttt{Int} \wedge \texttt{Int}) \rightarrow \texttt{Bool}$$

What we have now

Materialization is also transitive!

$\Longrightarrow$ We can use both subtyping and materialization in subsumption rules

What we have now

Materialization is also transitive!

$\implies$ We can use both subtyping and materialization in subsumption rules

$$[\textsc{Sub}]\ \frac{\Gamma \vdash e : \tau' \qquad \tau' \leq \tau}{\Gamma \vdash e : \tau} \qquad [\textsc{Mat}]\ \frac{\Gamma \vdash e : \tau' \qquad \tau' \preccurlyeq \tau}{\Gamma \vdash e : \tau}$$

$$[\textsc{Appl}]\ \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

What we have now

Materialization is also transitive!

$\Longrightarrow$ We can use both subtyping and materialization in subsumption rules

$$[\textsc{Sub}]\ \frac{\Gamma \vdash e : \tau' \qquad \tau' \leq \tau}{\Gamma \vdash e : \tau} \qquad [\textsc{Mat}]\ \frac{\Gamma \vdash e : \tau' \qquad \tau' \preccurlyeq \tau}{\Gamma \vdash e : \tau}$$

$$[\textsc{Appl}]\ \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

We got the modus ponens back!

Compilation

Objectives

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

Objectives

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

$$(\lambda \mathrm{x} : \texttt{?}.\mathrm{x} + 1)\ \texttt{False} \qquad \text{should fail}$$

$$(\lambda \mathrm{x} : \texttt{?}.\mathrm{x} + 1)\ 3 \qquad \text{should reduce to } 4$$

Objectives

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

$$(\lambda \mathrm{x} : \texttt{?}.\mathrm{x} + 1)\ \texttt{False} \qquad \text{should fail}$$

$$(\lambda \mathrm{x} : \texttt{?}.\mathrm{x} + 1)\ \texttt{3} \qquad \text{should reduce to 4}$$

– 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

$$(\lambda \mathtt{x} : \mathtt{?}.\mathtt{x} + 1)\ \mathtt{False} \qquad \text{should fail}$$
$$(\lambda \mathtt{x} : \mathtt{?}.\mathtt{x} + 1)\ \mathtt{3} \qquad \text{should reduce to 4}$$

– Have a declarative compilation system and an algorithmic one

– Make sure that compilation preserve types

Cast language

GTLC + *Casts* :

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

Cast language

GTLC + *Casts* :

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

Example:

$$(\lambda^{? \to \mathtt{Int}} \mathtt{x}.\, \mathtt{x}\langle\mathtt{Int}\rangle + 1)\ \mathtt{true} \to (\mathtt{true}\langle\mathtt{Int}\rangle) + 1$$
$$\to \mathtt{CastError}$$

Declarative compilation system

Main idea: to every application of the materialization rule corresponds a cast

Declarative compilation system

Main idea: to every application of the materialization rule corresponds a cast

$$[\textsc{Mat}]\ \frac{\Gamma \vdash e \rightsquigarrow E : \tau' \qquad \tau' \preccurlyeq \tau}{\Gamma \vdash e \rightsquigarrow E\langle\tau\rangle : \tau}$$

Declarative compilation system

Main idea: to every application of the materialization rule corresponds a cast

$$[\textsc{Mat}]\ \frac{\Gamma \vdash e \rightsquigarrow E : \tau' \qquad \tau' \preccurlyeq \tau}{\Gamma \vdash e \rightsquigarrow E\langle\tau\rangle : \tau}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 \rightsquigarrow E_2 : \tau'}{\Gamma \vdash e_1\ e_2 \rightsquigarrow E_1\ E_2 : \tau}$$

Declarative compilation system

Main idea: to every application of the materialization rule corresponds a cast

$$[\textsc{Mat}]\ \frac{\Gamma \vdash e \rightsquigarrow E : \tau' \qquad \tau' \preccurlyeq \tau}{\Gamma \vdash e \rightsquigarrow E\langle\tau\rangle : \tau}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 \rightsquigarrow E_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 \rightsquigarrow E_2 : \tau'}{\Gamma \vdash e_1\ e_2 \rightsquigarrow E_1\ E_2 : \tau}$$

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?