Consistent Subtyping for All

Ningning Xie Xuan Bi Bruno C. d. S. Oliveira

11 May, 2018

The University of Hong Kong

Background

There has been ongoing debate about which language paradigm, static typing or dynamic typing, is better

Background [1]

Some people are in favor of static typing:

- Communication
- Reliability
- Efficiency
- Productivity

[1] Adapted from POPL2017 tutorial.

Background [1]

Some people are in favor of static typing:

- Communication
- Reliability
- Efficiency
- Productivity

the other prefer dynamic typing:

- ~~Don't have to write type annotations~~
- Expressiveness
- Cognitive load
- Learning curve

[1] Adapted from POPL2017 tutorial.

Gradual Typing From a Programmer's View

Gradual typing enables the evolution of programs from untyped to typed, and provides *fine-grained* control over which parts are statically checked.[2]

[2] Examples courtesy of Garcia's slides at POPL'16

Gradual Typing From a Programmer's View

Gradual typing enables the evolution of programs from untyped to typed, and provides *fine-grained* control over which parts are statically checked.[2]

- Program with no type information (dynamic checking)

```
def f(x) = x + 2
def h(g) = g(1)
h f
```


[2]Examples courtesy of Garcia's slides at POPL'16

Gradual Typing From a Programmer's View

Gradual typing enables the evolution of programs from untyped to typed, and provides *fine-grained* control over which parts are statically checked.[2]

- Program with no type information (dynamic checking)

```
def f(x) = x + 2
def h(g) = g(1)
h f
```

- Program with full type information (static checking)

```
def f(x : Int) = x + 2
def h(g : Int → Int) = g(1)
h f
```

[2] Examples courtesy of Garcia's slides at POPL'16

Gradual Typing From a Programmer's View

Gradual typing enables the evolution of programs from untyped to typed, and provides *fine-grained* control over which parts are statically checked.[2]

- Program with no type information (dynamic checking)

```
def f(x) = x + 2
def h(g) = g(1)
h f
```

- Program with full type information (static checking)

```
def f(x : Int) = x + 2
def h(g : Int → Int) = g(1)
h f
```

- Program with some type information (mixed checking)

```
def f(x : Int) = x + 2
def h(g) = g(1)
h f
```

[2] Examples courtesy of Garcia's slides at POPL'16

Gradual Typing 101

- A gradual type system enforces static type discipline whenever possible:

```
def f(x : Bool) = x + 2    -- static error
def h (g) = g(1)
h f
```

Gradual Typing 101

- A gradual type system enforces static type discipline whenever possible:

```
def f(x : Bool) = x + 2    -- static error
def h (g) = g(1)
h f
```

> "Inside every gradual language is a small static language struggling to get out..."
>
> Anonymous

Gradual Typing 101

- A gradual type system enforces static type discipline whenever possible:

```
def f(x : Bool) = x + 2    -- static error
def h (g) = g(1)
h f
```

> "Inside every gradual language is a small static language struggling to get out..."
>
> — Anonymous

- When the type information is not available, it delegates to dynamic checking at runtime:

```
def f(x : Int) = x + 2
def h(g) = g(True)
h f                          -- runtime error
```

Gradual Typing 101

- The key external feature of every gradual type system is the *unknown type* $\star$.

```
f (x : Int) = x + 2    -- static checking
h (g : ⋆) = g 1        -- dynamic checking
h f
```

- Central to gradual typing is type consistency $\sim$, which relaxes type equality: $\star \sim \mathsf{Int}$, $\star \to \mathsf{Int} \sim \mathsf{Int} \to \star, \ldots$

$$\begin{array}{c} \mathsf{Int} = \mathsf{Int} \\ \mathsf{Bool} = \mathsf{Bool} \\ \mathsf{Int} \neq \mathsf{Bool} \end{array} \quad \xrightarrow{\textit{extend}} \quad \begin{array}{c} \mathsf{Int} \sim \mathsf{Int} \\ \mathsf{Bool} \sim \mathsf{Bool} \\ \mathsf{Int} \not\sim \mathsf{Bool} \\ \star \sim \mathsf{Int} \\ \mathsf{Int} \to \star \sim \star \to \mathsf{Int} \\ \ldots \end{array}$$

Gradual Typing 101

- The key external feature of every gradual type system is the *unknown type* $\star$.

```
f (x : Int) = x + 2    -- static checking
h (g : ⋆) = g 1         -- dynamic checking
h f
```

- Central to gradual typing is type consistency $\sim$, which relaxes type equality: $\star \sim \mathsf{Int}$, $\star \to \mathsf{Int} \sim \mathsf{Int} \to \star, \ldots$
- Dynamic semantics is defined by type-directed translation to an internal language with runtime casts:

$$(\langle \star \hookrightarrow \star \to \star \rangle g)\ (\langle \mathsf{Int} \hookrightarrow \star \rangle 1)$$

Many Successes

Gradual typing has seen great popularity both in academia and industry. Over the years, there emerge many gradual type disciplines:

- Subtyping
- Parametric Polymorphism
- Type inference
- Security Typing
- Effects
- ...

Many Successes, But...

Gradual typing has seen great popularity both in academia and industry. Over the years, there emerge many gradual type disciplines:

- Subtyping
- Parametric Polymorphism
- Type inference
- Security Typing
- Effects
- ...

☞ *As type systems get more complex, it becomes more difficult to adapt notions of gradual typing.* [Garcia et al., 2016]

Problem

- Can we design a gradual type system with *implicit higher-rank polymorphism*?

Problem

- Can we design a gradual type system with *implicit higher-rank polymorphism*?
- State-of-art techniques are inadequate.

Why It Is interesting

- Haskell supports implicit higher-rank polymorphism, but some "safe" programs are rejected:

```
foo :: ([Int], [Char])
foo = let f x = (x [1, 2], x ['a', 'b'])
      in f reverse  -- GHC rejects
```

Why It Is interesting

- Haskell supports implicit higher-rank polymorphism, but some "safe" programs are rejected:

```
foo :: ([Int], [Char])
foo = let f x = (x [1, 2], x ['a', 'b'])
      in f reverse   -- GHC rejects
```

- If we had gradual typing...

```
let f (x : ⋆) = (x [1, 2], x ['a', 'b'])
in f reverse
```

Why It Is interesting

- Haskell supports implicit higher-rank polymorphism, but some "safe" programs are rejected:

```
foo :: ([Int], [Char])
foo = let f x = (x [1, 2], x ['a', 'b'])
      in f reverse  -- GHC rejects
```

- If we had gradual typing...

```
let f (x : ⋆) = (x [1, 2], x ['a', 'b'])
in f reverse
```

- Moving to more precised version still type checks, but with more static safety guarantee:

```
let f (x : ∀a. [a] → [a]) = (x [1, 2], x ['a', 'b'])
in f reverse
```

Contributions

- A new specification of consistent subtyping that works for implicit higher-rank polymorphism
- An easy-to-follow recipe for turning subtyping into consistent subtyping
- A gradually typed calculus with implicit higher-rank polymorphism
    - Satisfies correctness criteria (formalized in Coq)
    - A sound and complete algorithm

What Is Consistent Subtyping

- Consistent subtyping ($\lesssim$) is the extension of subtyping to gradual types. [Siek and Taha, 2007]

What Is Consistent Subtyping

- Consistent subtyping ($\lesssim$) is the extension of subtyping to gradual types. [Siek and Taha, 2007]
- A static subtyping relation ($<:$) over gradual types, with the key insight that $\star$ is *neutral* to subtyping ($\star <: \star$)

What Is Consistent Subtyping

- Consistent subtyping ($\lesssim$) is the extension of subtyping to gradual types. [Siek and Taha, 2007]
- A static subtyping relation ($<:$) over gradual types, with the key insight that $\star$ is *neutral* to subtyping ($\star <: \star$)

Definition (Consistent Subtyping à la Siek and Taha)

The following two are *equivalent*:

1. $A \lesssim B$ if and only if $A \sim C$ and $C <: B$ for some C.
2. $A \lesssim B$ if and only if $A <: C$ and $C \sim B$ for some C.

Design Principle

☞ *Gradual typing and subtyping are orthogonal and can be combined in a principled fashion.* – *Siek and Taha*

Challenge

- Polymorphic types induce a subtyping relation:
  $\forall a.\, a \rightarrow a <: \mathsf{Int} \rightarrow \mathsf{Int}$
- Design consistent subtyping that combines 1) consistency 2) subtyping 3) polymorphism.

Challenge

- Polymorphic types induce a subtyping relation:
  $\forall a.\, a \to a <: \mathsf{Int} \to \mathsf{Int}$
- Design consistent subtyping that combines 1) consistency 2) subtyping 3) polymorphism.

☞ *Gradual typing and polymorphism are orthogonal and can be combined in a principled fashion.*[3]

[3] Note that here we are mostly concerned with static semantics.

Problem with Existing Definition

Odersky-Läufer Type System

- The underlying static language is the well-established type system for higher-rank types. [Odersky and Läufer, 1996]

Types	A, B	$::=$	$\mathsf{Int} \mid a \mid A \to B \mid \forall a.\, A$
Monotypes	τ, σ	$::=$	$\mathsf{Int} \mid a \mid \tau \to \sigma$
Terms	e	$::=$	$x \mid \mathsf{n} \mid \lambda x : A.\, e \mid \lambda x.\, e \mid e_1\, e_2$
Contexts	Ψ	$::=$	$\bullet \mid \Psi, x : A \mid \Psi, a$

Subtyping

$\boxed{\Psi \vdash A <: B}$ *(Subtyping)*

$$\frac{a \in \Psi}{\Psi \vdash a <: a} \qquad \frac{}{\Psi \vdash \mathsf{Int} <: \mathsf{Int}} \qquad \frac{\Psi \vdash B_1 <: A_1 \qquad \Psi \vdash A_2 <: B_2}{\Psi \vdash A_1 \to A_2 <: B_1 \to B_2}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] <: B}{\Psi \vdash \forall a.\, A <: B} \qquad \frac{\Psi, a \vdash A <: B}{\Psi \vdash A <: \forall a.\, B}$$

Subtyping with Unknown Types

$\boxed{\Psi \vdash A <: B}$ *(Subtyping)*

$$\frac{a \in \Psi}{\Psi \vdash a <: a} \qquad \frac{}{\Psi \vdash \mathsf{Int} <: \mathsf{Int}} \qquad \frac{\Psi \vdash B_1 <: A_1 \qquad \Psi \vdash A_2 <: B_2}{\Psi \vdash A_1 \to A_2 <: B_1 \to B_2}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] <: B}{\Psi \vdash \forall a.\, A <: B} \qquad \frac{\Psi, a \vdash A <: B}{\Psi \vdash A <: \forall a.\, B}$$

$$\frac{}{\Psi \vdash \star <: \star}$$

Type Consistency

$\boxed{A \sim B}$ *(Type Consistency)*

$$\frac{}{A \sim A} \qquad \frac{}{A \sim \star} \qquad \frac{}{\star \sim A} \qquad \frac{A_1 \sim B_1 \qquad A_2 \sim B_2}{A_1 \to A_2 \sim B_1 \to B_2}$$

Type Consistency with Polymorphic Types

$\boxed{A \sim B}$ *(Type Consistency)*

$$\frac{}{A \sim A} \qquad \frac{}{A \sim \star} \qquad \frac{}{\star \sim A} \qquad \frac{A_1 \sim B_1 \qquad A_2 \sim B_2}{A_1 \to A_2 \sim B_1 \to B_2}$$

$$\frac{A \sim B}{\forall a.\, A \sim \forall a.\, B}$$

Type Consistency with Polymorphic Types

$\boxed{A \sim B}$ *(Type Consistency)*

$$\frac{}{A \sim A} \qquad \frac{}{A \sim \star} \qquad \frac{}{\star \sim A} \qquad \frac{A_1 \sim B_1 \qquad A_2 \sim B_2}{A_1 \to A_2 \sim B_1 \to B_2}$$

$$\frac{A \sim B}{\forall a.\, A \sim \forall a.\, B}$$

☞ *The simplicity comes from the orthogonality between consistency and subtyping!*

Bad News

Definition (Consistent Subtyping à la Siek and Taha)

The following two are equivalent:

1. $A \lesssim B$ if and only if $A \sim C$ and $C <: B$ for some C.
2. $A \lesssim B$ if and only if $A <: C$ and $C \sim B$ for some C.

☞ *Equivalence is broken in the polymorphic setting!*

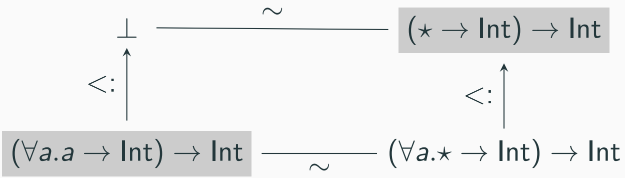
Bad News

Definition (Consistent Subtyping à la Siek and Taha)

The following two are equivalent:

1. $A \lesssim B$ if and only if $A \sim C$ and $C <: B$ for some C. ✓
2. $A \lesssim B$ if and only if $A <: C$ and $C \sim B$ for some C. ✗

☞ *Equivalence is broken in the polymorphic setting!*



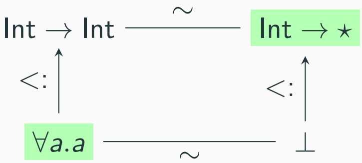
Bad News

Definition (Consistent Subtyping à la Siek and Taha)

The following two are equivalent:

1. $A \lesssim B$ if and only if $A \sim C$ and $C <: B$ for some C. ✗
2. $A \lesssim B$ if and only if $A <: C$ and $C \sim B$ for some C. ✓

☞ *Equivalence is broken in the polymorphic setting!*



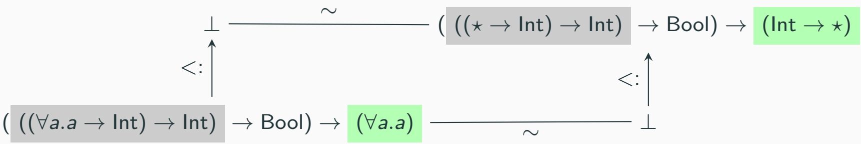
Bad News

Definition (Consistent Subtyping à la Siek and Taha)

The following two are equivalent:

1. $A \lesssim B$ if and only if $A \sim C$ and $C <: B$ for some C. ✗
2. $A \lesssim B$ if and only if $A <: C$ and $C \sim B$ for some C. ✗

☞ *Equivalence is broken in the polymorphic setting!*



Revisiting Consistent Subtyping

Consistent Subtyping vs. Subtyping

- Subtyping validates the *subsumption principle*

$$\frac{\Psi \vdash e : A \qquad A <: B}{\Psi \vdash e : B}$$

Consistent Subtyping vs. Subtyping

- Subtyping validates the *subsumption principle*, so should consistent subtyping

$$\frac{\Psi \vdash e : A \qquad A \lesssim B}{\Psi \vdash e : B}$$

Consistent Subtyping vs. Subtyping

- Subtyping validates the *subsumption principle*, so should consistent subtyping

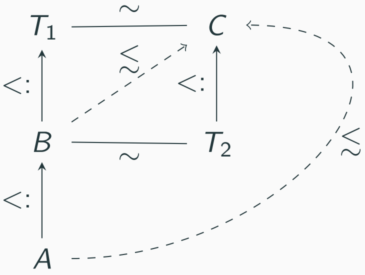
$$\frac{\Psi \vdash e : A \qquad A \lesssim B}{\Psi \vdash e : B}$$

- Subtyping is transitive, but consistent subtyping *is not*

Observations

Observation (I)

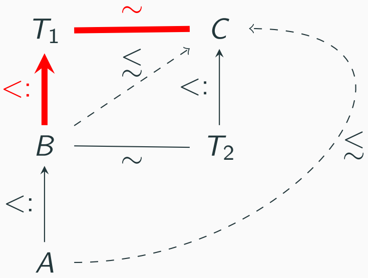
If $A <: B$ *and* $B \lesssim C$*, then* $A \lesssim C$*.*



Observations

Observation (I)

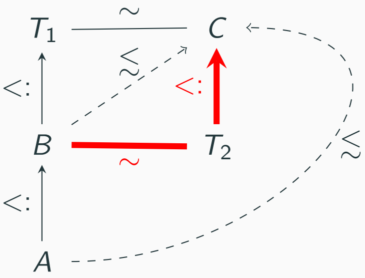
If $A <: B$ and $B \lesssim C$, then $A \lesssim C$.



Observations

Observation (I)

If $A <: B$ *and* $B \lesssim C$*, then* $A \lesssim C$.



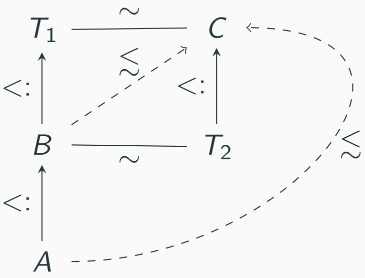
Observations

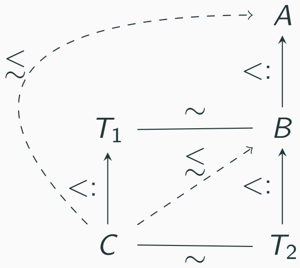
Observation (I)

If $A <: B$ *and* $B \lesssim C$*, then* $A \lesssim C$*.*

Observation (II)

If $C \lesssim B$ *and* $B <: A$*, then* $C \lesssim A$*.*





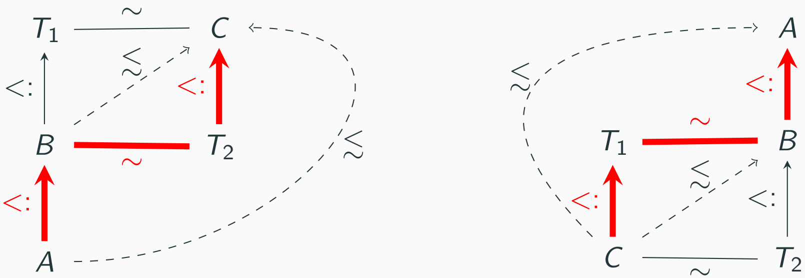
Observations

Observation (I)

If $A <: B$ *and* $B \lesssim C$*, then* $A \lesssim C$.

Observation (II)

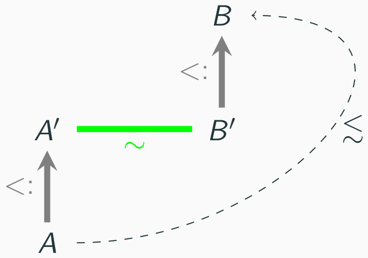
If $C \lesssim B$ *and* $B <: A$*, then* $C \lesssim A$.



Consistent Subtyping, the Specification

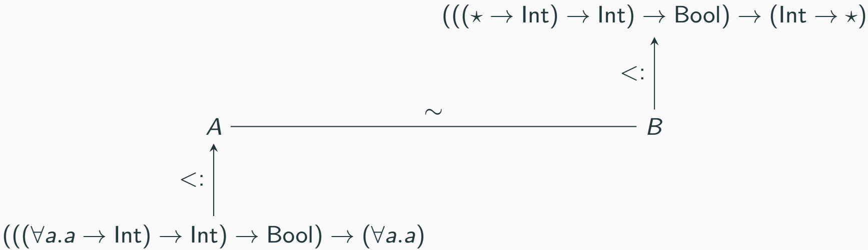
Definition (Generalized Consistent Subtyping)

$\Psi \vdash A \lesssim B \overset{def}{=} \Psi \vdash A <: A'$, $A' \sim B'$ and $\Psi \vdash B' <: B$ for some A' and B'.



Definition (Generalized Consistent Subtyping)

$\Psi \vdash A \lesssim B \overset{def}{=} \Psi \vdash A <: A'$, $A' \sim B'$ and $\Psi \vdash B' <: B$ for some A' and B'.



$$A = ((\forall a.a \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \mathsf{Int})$$

$$B = ((\forall a.\star \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \star)$$

Non-Determinism

Definition (Generalized Consistent Subtyping)

$\Psi \vdash A \lesssim B \stackrel{def}{=} \Psi \vdash A <: A'$, $A' \sim B'$ and $\Psi \vdash B' <: B$ for some A' and B'.

Two sources of non-determinism:

1. Two intermediate types A' and B'

Non-Determinism

Definition (Generalized Consistent Subtyping)

$\Psi \vdash A \lesssim B \overset{def}{=}$ $\Psi \vdash A <: A'$, $A' \sim B'$ and $\Psi \vdash B' <: B$ for some A' and B'.

Two sources of non-determinism:

1. Two intermediate types A' and B'

2. Guessing monotypes $\dfrac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] <: B}{\Psi \vdash \forall a.\, A <: B}$

Non-Determinism

Definition (Generalized Consistent Subtyping)

$\Psi \vdash A \lesssim B \overset{def}{=} \Psi \vdash A <: A'$, $A' \sim B'$ and $\Psi \vdash B' <: B$ for some A' and B'.

Two sources of non-determinism:

1. Two intermediate types A' and B'

☞ *We can derive a syntax-directed inductive definition without resorting to subtyping or consistency at all!*

Consistent Subtyping Without Existentials

Notice $\Psi \vdash \star \lesssim A$ always holds ($\star <: \star \sim A <: A$), and vise versa ($\Psi \vdash A \lesssim \star$)

Consistent Subtyping Without Existentials: First Step

1. Replace $<:$ with $\lesssim$

$\boxed{\Psi \vdash A <: B}$ *(Subtyping)*

$$\frac{a \in \Psi}{\Psi \vdash a <: a} \qquad \frac{}{\Psi \vdash \mathsf{Int} <: \mathsf{Int}} \qquad \frac{\Psi \vdash B_1 <: A_1 \qquad \Psi \vdash A_2 <: B_2}{\Psi \vdash A_1 \to A_2 <: B_1 \to B_2}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] <: B}{\Psi \vdash \forall a.\, A <: B} \qquad \frac{\Psi, a \vdash A <: B}{\Psi \vdash A <: \forall a.\, B}$$

$$\frac{}{\Psi \vdash \star <: \star}$$

Consistent Subtyping Without Existentials: First Step

1. Replace $<:$ with $\lesssim$

$\boxed{\Psi \vdash A \lesssim B}$ *(Consistent Subtyping, not yet)*

$$\frac{a \in \Psi}{\Psi \vdash a \lesssim a} \qquad \frac{}{\Psi \vdash \mathsf{Int} \lesssim \mathsf{Int}} \qquad \frac{\Psi \vdash B_1 \lesssim A_1 \qquad \Psi \vdash A_2 \lesssim B_2}{\Psi \vdash A_1 \to A_2 \lesssim B_1 \to B_2}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] \lesssim B}{\Psi \vdash \forall a.\, A \lesssim B} \qquad \frac{\Psi, a \vdash A \lesssim B}{\Psi \vdash A \lesssim \forall a.\, B}$$

$$\frac{}{\Psi \vdash \star \lesssim \star}$$

Consistent Subtyping Without Existentials: Second Step

1. Replace $<:$ with $\lesssim$
2. Replace $\Psi \vdash \star \lesssim \star$ with $\Psi \vdash \star \lesssim A$ and $\Psi \vdash A \lesssim \star$

$\boxed{\Psi \vdash A \lesssim B}$ *(Consistent Subtyping, not yet)*

$$\frac{a \in \Psi}{\Psi \vdash a \lesssim a} \qquad \frac{}{\Psi \vdash \mathsf{Int} \lesssim \mathsf{Int}} \qquad \frac{\Psi \vdash B_1 \lesssim A_1 \qquad \Psi \vdash A_2 \lesssim B_2}{\Psi \vdash A_1 \to A_2 \lesssim B_1 \to B_2}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] \lesssim B}{\Psi \vdash \forall a.\, A \lesssim B} \qquad \frac{\Psi, a \vdash A \lesssim B}{\Psi \vdash A \lesssim \forall a.\, B}$$

$$\frac{}{\Psi \vdash \star \lesssim \star}$$

Consistent Subtyping Without Existentials: Second Step

1. Replace $<:$ with $\lesssim$
2. Replace $\Psi \vdash \star \lesssim \star$ with $\Psi \vdash \star \lesssim A$ and $\Psi \vdash A \lesssim \star$

$\boxed{\Psi \vdash A \lesssim B}$ *(Consistent Subtyping)*

$$\frac{a \in \Psi}{\Psi \vdash a \lesssim a} \qquad \frac{}{\Psi \vdash \mathsf{Int} \lesssim \mathsf{Int}} \qquad \frac{\Psi \vdash B_1 \lesssim A_1 \qquad \Psi \vdash A_2 \lesssim B_2}{\Psi \vdash A_1 \to A_2 \lesssim B_1 \to B_2}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] \lesssim B}{\Psi \vdash \forall a.\, A \lesssim B} \qquad \frac{\Psi, a \vdash A \lesssim B}{\Psi \vdash A \lesssim \forall a.\, B}$$

$$\frac{}{\Psi \vdash \star \lesssim A} \qquad \frac{}{\Psi \vdash A \lesssim \star}$$

Definition Meets Specification

Theorem

$\Psi \vdash A \lesssim B$ *iff* $\Psi \vdash A <: A'$, $A' \sim B'$ *and* $\Psi \vdash B' <: B$ *for some* A' *and* B'.

Declarative Type System

Type System

$\boxed{\Psi \vdash e : A}$ *(Typing, selected rules)*

$$\frac{\Psi, a \vdash e : A}{\Psi \vdash e : \forall a.\, A}\ \textsc{u-gen}$$

$$\frac{\Psi, x : A \vdash e : B}{\Psi \vdash \lambda x : A.\, e : A \to B}\ \textsc{u-lamann}$$

$$\frac{\Psi, x : \tau \vdash e : B}{\Psi \vdash \lambda x.\, e : \tau \to B}\ \textsc{u-lam}$$

$$\frac{\Psi \vdash e_1 : A \qquad \Psi \vdash A \rhd A_1 \to A_2 \qquad \Psi \vdash e_2 : A_3 \qquad \Psi \vdash A_3 \lesssim A_1}{\Psi \vdash e_1\ e_2 : A_2}\ \textsc{u-app}$$

Type System

$$\frac{\Psi \vdash e_1 : A \qquad \Psi \vdash A \rhd A_1 \to A_2 \qquad \Psi \vdash e_2 : A_3 \qquad \Psi \vdash A_3 \lesssim A_1}{\Psi \vdash e_1\ e_2 : A_2}\ \textsc{U-App}$$

$\boxed{\Psi \vdash A \rhd A_1 \to A_2}$ *(Matching)*

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash A[a \mapsto \tau] \rhd A_1 \to A_2}{\Psi \vdash \forall a.\, A \rhd A_1 \to A_2}\ \textsc{M-Forall}$$

$$\frac{}{\Psi \vdash A_1 \to A_2 \rhd A_1 \to A_2}\ \textsc{M-Arr} \qquad \frac{}{\Psi \vdash \star \rhd \star \to \star}\ \textsc{M-Unknown}$$

Dynamic Semantics

- Type-directed translation into an intermediate language with runtime casts ($\Psi \vdash e : A \rightsquigarrow s$)
- We translate to the Polymorphic Blame Calculus (PBC) [Ahmed et al., 2011]

PBC terms[4] $\quad s ::= x \mid \mathsf{n} \mid \lambda x : A.\, s \mid \Lambda a.\, s \mid s_1\, s_2 \mid \langle A \hookrightarrow B \rangle s$

[4] Only a subst of PBC terms are used

Correctness Criteria

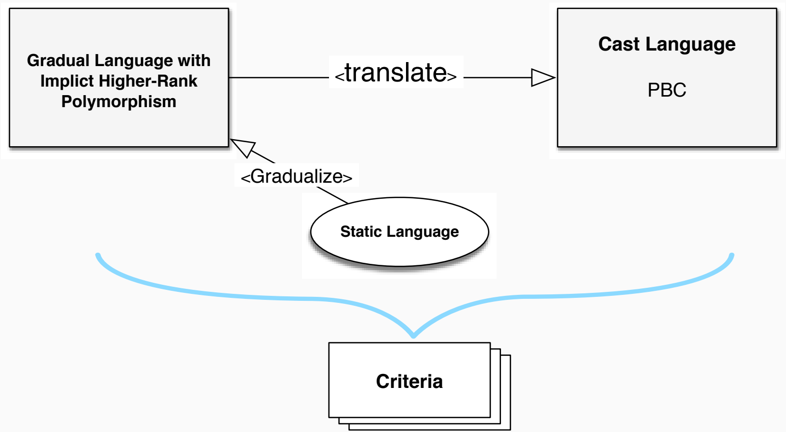
- **Conservative extension:** for all static Ψ, e, and A,
  - if $\Psi \vdash^{OL} e : A$, then there exists B, such that $\Psi \vdash e : B$, and $\Psi \vdash B <: A$.
  - if $\Psi \vdash e : A$, then $\Psi \vdash^{OL} e : A$
- **Monotonicity w.r.t. precision:** for all Ψ, e, e', A, if $\Psi \vdash e : A$, and $e' \sqsubseteq e$, then $\Psi \vdash e' : B$, and $B \sqsubseteq A$ for some B.
- **Type Preservation of cast insertion:** for all Ψ, e, A, if $\Psi \vdash e : A$, then $\Psi \vdash e : A \rightsquigarrow s$, and $\Psi \vdash^{B} s : A$ for some s.
- **Monotonicity of cast insertion:** for all $\Psi, e_1, e_2, s_1, s_2, A$, if $\Psi \vdash e_1 : A \rightsquigarrow s_1$, and $\Psi \vdash e_2 : A \rightsquigarrow s_2$, and $e_1 \sqsubseteq e_2$, then $\Psi \mid \Psi \vdash s_1 \sqsubseteq^{B} s_2$.

Correctness Criteria

- **Conservative extension:** for all static Ψ, e, and A,
  - if $\Psi \vdash^{OL} e : A$, then there exists B, such that $\Psi \vdash e : B$, and $\Psi \vdash B <: A$.
  - if $\Psi \vdash e : A$, then $\Psi \vdash^{OL} e : A$
- **Monotonicity w.r.t. precision:** for all Ψ, e, e', A, if $\Psi \vdash e : A$, and $e' \sqsubseteq e$, then $\Psi \vdash e' : B$, and $B \sqsubseteq A$ for some B.
- **Type Preservation of cast insertion:** for all Ψ, e, A, if $\Psi \vdash e : A$, then $\Psi \vdash e : A \rightsquigarrow s$, and $\Psi \vdash^{B} s : A$ for some s.
- **Monotonicity of cast insertion:** for all $\Psi, e_1, e_2, s_1, s_2, A$, if $\Psi \vdash e_1 : A \rightsquigarrow s_1$, and $\Psi \vdash e_2 : A \rightsquigarrow s_2$, and $e_1 \sqsubseteq e_2$, then $\Psi \mid \Psi \vdash s_1 \sqsubseteq^{B} s_2$.

☞ *Proved in Coq!*

Recap


Gradual Language with Implict Higher-Rank Polymorphism
<translate>
Cast Language
PBC
<Gradualize>
Static Language
Criteria

More in the Paper

- A bidirectional account of the algorithmic type system (inspired by [Dunfield and Krishnaswami, 2013])
- Extension to top types
- Discussion and comparison with other approaches (AGT [Garcia et al., 2016], Directed Consistency [Jafery and Dunfield, 2017])
- Discussion of dynamic guarantee

Future Work

- Fix the issue with dynamic guarantee (partially)
- More features: fancy types, etc.

References

A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *POPL*, 2011.

J. Dunfield and N. R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, 2013.

R. Garcia, A. M. Clark, and É. Tanter. Abstracting gradual typing. In *POPL*, 2016.

K. A. Jafery and J. Dunfield. Sums of uncertainty: Refinements go gradual. In *POPL*, 2017.

M. Odersky and K. Läufer. Putting type annotations to work. In *POPL*, 1996.

J. G. Siek and W. Taha. Gradual typing for objects. In *ECOOP*, 2007.

Consistent Subtyping for All

Ningning Xie Xuan Bi Bruno C. d. S. Oliveira

11 May, 2018

The University of Hong Kong

Backup Slides

Dynamic Guarantee

- Changes to the annotations of a gradually typed program should not change the dynamic behaviour of the program.
- The declarative system breaks it...

$$(\lambda f : \forall a. a \to \mathsf{Int}. \lambda x : \mathsf{Int}. f\ x)\,(\lambda x. 1)\, 3 \Downarrow 3$$

$$(\lambda f : \forall a. a \to \mathsf{Int}. \lambda x : \star. f\ x)\,(\lambda x. 1)\, 3 \Downarrow ?$$

- A common problem in gradual type inference, see [Garcia and Cimini 2015]. Static and gradual type parameters may help.
- A more sophisticated term precision is needed in PBC. [Igarashi et al. 2017]