Gradual Typing: A New Perspective

With polymorphism, unions, intersections, and much more

G. Castagna, **V. Lanvin**, T. Petrucciani, J. Siek

18 February 2019

Gradual Typing (1/3)

Let's write a map, that can work on both arrays and lists depending on a condition:

```
let map (condition : Bool) (f : α -> β) (data :  ) :     =
```

Gradual Typing (1/3)

Let's write a map, that can work on both arrays and lists depending on a condition:

```
let map (condition : Bool) (f : α -> β) (data :  ) :     =
  if condition then
    List.map f data
  else
    Array.map f data
```

Gradual Typing (1/3)

Let's write a map, that can work on both arrays and lists depending on a condition:

```
let map (condition : Bool) (f : α -> β) (data : ?) :     =
  if condition then
    List.map f data
  else
    Array.map f data
```

Gradual Typing (1/3)

Let's write a map, that can work on both arrays and lists depending on a condition:

```
let map (condition : Bool) (f : α -> β) (data : ?) :  ? =
  if condition then
    List.map f data
  else
    Array.map f data
```

Gradual Typing (1/3)

Let's write a map, that can work on both arrays and lists depending on a condition:

```
let map (condition : Bool) (f : α -> β) (data : ?) :  ? =
  if condition then
    List.map f data
  else
    Array.map f data
```

Runtime checks or **casts** are then inserted **automatically** by the compiler.

Gradual Typing (2/3)

- Goal: have both **static** and **dynamic** typing in the same language.
- How: by adding a **dynamic type**, denoted “?”.

Gradual Typing (2/3)

- Goal: have both **static** and **dynamic** typing in the same language.
- How: by adding a **dynamic type**, denoted “?”.
- **Allows for a trade-off between safety and programming productivity.**

Gradual Typing (2/3)

- Goal: have both **static** and **dynamic** typing in the same language.
- How: by adding a **dynamic type**, denoted "?".
- **Allows for a trade-off between safety and programming productivity.**

The transition is **gradual**:

$$\texttt{?} \preccurlyeq \texttt{?} \to \texttt{?} \preccurlyeq \texttt{Int} \to \texttt{?} \preccurlyeq \texttt{Int} \to \texttt{Bool}$$

Gradual Typing (3/3)

Sometimes this **gradualization** is too **coarse**

```
let map (condition : Bool) (f : α -> β) (data : ?) : ? =
  if condition then
    List.map f data
  else
    Array.map f data
in
map (Random.bool ()) (fun x -> x) "Hello"
```

Gradual Typing (3/3)

Sometimes this **gradualization** is too **coarse**

```
let map (condition : Bool) (f : α -> β) (data : ?) : ? =
  if condition then
    List.map f data
  else
    Array.map f data
in
map (Random.bool ()) (fun x -> x) "Hello"
```

This **always fails!**

Gradual Typing (3/3)

Sometimes this **gradualization** is too **coarse**

```
let map (condition : Bool) (f : α -> β) (data : ?) : ? =
  if condition then
    List.map f data
  else
    Array.map f data
in
map (Random.bool ()) (fun x -> x) "Hello"
```

This **always fails!**

We want to give the programmer a way to reject such cases **statically**, while still **accepting this function**.

Enter Set-Theoretic Types (1/2)

```
let map (condition : Bool) (f : α -> β)
(data :   ) :     =
  if condition then
    List.map f data
  else
    Array.map f data
```

Enter Set-Theoretic Types (1/2)

```
let map (condition : Bool) (f : α -> β)
(data : (α array ∨ α list)) :      =
  if condition then
    List.map f data
  else
    Array.map f data
```

Enter Set-Theoretic Types (1/2)

```
let map (condition : Bool) (f : α -> β)
(data : (α array ∨ α list)) : (β array ∨ β list) =
  if condition then
    List.map f data
  else
    Array.map f data
```

Enter Set-Theoretic Types (1/2)

```
let map (condition : Bool) (f : α -> β)
(data : (α array ∨ α list)) : (β array ∨ β list) =
  if condition then
    List.map f data
  else
    Array.map f data
```

Unfortunately, this is **not well-typed** without additional checks, since α `array` $\vee$ α `list` $\not\leq$ α `array`.

Enter Set-Theoretic Types (2/2)

We need to explicitly **deconstruct the union**:

Enter Set-Theoretic Types (2/2)

We need to explicitly **deconstruct the union**:

```
let map (condition : Bool) (f : α -> β)
(data : (α array ∨ α list)) : (β array ∨ β list) =
  if condition then
    if typeOf(data) = α list then
      List.map f data
    else
      raise Runtime_type_error
  else
    (* Same for arrays *)
```

Enter Set-Theoretic Types (2/2)

We need to explicitly **deconstruct the union**:

```
let map (condition : Bool) (f : α -> β)
(data : (α array ∨ α list)) : (β array ∨ β list) =
  if condition then
    if typeOf(data) = α list then
      List.map f data
    else
      raise Runtime_type_error
  else
    (* Same for arrays *)
```

This is **safer**, but **extremely verbose**.

Set-Theoretic Types Summarized

- **Types** with **connectives** ($\vee$, $\wedge$, $\neg$)

Set-Theoretic Types Summarized

- **Types** with **connectives** ($\vee$, $\wedge$, $\neg$)
- **Useful for overloading, branching, but often syntactically heavy.**

Set-Theoretic Types Summarized

- **Types** with **connectives** ($\vee$, $\wedge$, $\neg$)
- **Useful for overloading, branching, but often syntactically heavy.**

  `(Int -> Int)` $\wedge$ `(Bool -> Bool)` = overloaded function

Set-Theoretic Types Summarized

- **Types** with **connectives** ($\vee$, $\wedge$, $\neg$)
- **Useful for overloading, branching, but often syntactically heavy.**

  `(Int -> Int)` $\wedge$ `(Bool -> Bool)` = overloaded function

  `if x then 3 else true :` `Int` $\vee$ `Bool`

Set-Theoretic Types Summarized

- **Types** with **connectives** ($\vee$, $\wedge$, $\neg$)
- **Useful for overloading, branching, but often syntactically heavy.**

  `(Int -> Int)` $\wedge$ `(Bool -> Bool)` = overloaded function

  `if x then 3 else true :` `Int` $\vee$ `Bool`

- In **Semantic subtyping**,

  Types $\simeq$ Sets of values

  Subtyping $\simeq$ Set-containment

Pros and Cons

Set-theoretic types	Gradual types
Safe	Unsafe
Expressive	Too coarse
Verbose	**Light**
Restrictive	**Permissive**

Pros and Cons

Set-theoretic types	Gradual types
Safe	Unsafe
Expressive	Too coarse
Verbose	**Light**
Restrictive	**Permissive**

Can we get the **best of both worlds?**

Mixing the Two

```
let map condition f
  (data : (α list ∨ α array) )  =
  if condition then
    List.map f data
  else
    Array.map f data
```

Mixing the Two

```
let map condition f
  (data : (α list ∨ α array) ∧ ?)  =
  if condition then
    List.map f data
  else
    Array.map f data
```

Mixing the Two

```
let map condition f
  (data : (α list ∨ α array) ∧ ?)  =
  if condition then
    List.map f data
  else
    Array.map f data
```

– By **subtyping**, $(\alpha\ \texttt{list} \vee \alpha\ \texttt{array}) \wedge ? \leq ?$.

Mixing the Two

```
let map condition f
  (data : (α list ∨ α array) ∧ ?)  =
  if condition then
    List.map f data
  else
    Array.map f data
```

- By **subtyping**, $(\alpha$ `list` $\vee$ α `array`$)$ $\wedge$ $? \leq ?$.
- Can only be used with **lists or arrays**
- No need for **manual type checks**

Mixing the Two

```
let map condition f
  (data : (α list ∨ α array) ∧ ?) =
  if condition then
    List.map f data
  else
    Array.map f data
```

- By **subtyping**, $(\alpha\ \texttt{list} \vee \alpha\ \texttt{array}) \wedge ? \leq ?$.
- Can only be used with **lists or arrays**
- No need for **manual type checks**
- We want to infer **all non-gradual types** (including the return type!)

Mixing the Two

```
let map (condition : Bool) f
  (data : (α list ∨ α array) ∧ ?)  =
  if condition then
    List.map f data
  else
    Array.map f data
```

- By **subtyping**, $(\alpha\ \texttt{list} \vee \alpha\ \texttt{array}) \wedge\ ? \leq\ ?$.
- Can only be used with **lists or arrays**
- No need for **manual type checks**
- We want to infer **all non-gradual types** (including the return type!)

Mixing the Two

```
let map condition (f : α -> β)
  (data : (α list ∨ α array) ∧ ?)  =
  if condition then
    List.map f data
  else
    Array.map f data
```

- By **subtyping**, (α `list` $\vee$ α `array`) $\wedge$? $\leq$?.
- Can only be used with **lists or arrays**
- No need for **manual type checks**
- We want to infer **all non-gradual types** (including the return type!)

Mixing the Two

```
let map condition f
  (data : (α list ∨ α array) ∧ ?) : β list ∨ β array =
  if condition then
    List.map f data
  else
    Array.map f data
```

- By **subtyping**, (α `list` $\vee$ α `array`) $\wedge$? $\leq$?.
- Can only be used with **lists or arrays**
- No need for **manual type checks**
- We want to infer **all non-gradual types** (including the return type!)

How is it Usually Done?

1. Define a **subtype-consistency** relation $\widetilde{\leq}$.

How is it Usually Done?

1. Define a **subtype-consistency** relation $\widetilde{\leq}$.

   This relation is not transitive! $? \;\widetilde{\leq}\; \tau \;\widetilde{\leq}\; ?$ for all τ

How is it Usually Done?

1. Define a **subtype-consistency** relation $\widetilde{\leq}$.

   This relation is not transitive! $? \mathrel{\widetilde{\leq}} \tau \mathrel{\widetilde{\leq}} ?$ for all τ

2. Embed this relation into typing rules.

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_1' \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_2 \mathrel{\widetilde{\leq}} \tau_1}{\Gamma \vdash e_1 \; e_2 : \tau_1'}$$

How is it Usually Done?

1. Define a **subtype-consistency** relation $\tilde{\leq}$.

   This relation is not transitive! $? \tilde{\leq} \tau \tilde{\leq} ?$ for all τ

2. Embed this relation into typing rules.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_2 \tilde{\leq} \mathtt{dom}(\tau_1)}{\Gamma \vdash e_1\ e_2 : \tau_1 \circ \tau_2}$$

This gets even more complicated with set-theoretic types!

Declarative Systems

What is the Dynamic Type?

Every occurrence of `?` behaves like a **distinct, existentially quantified** type variable.

What is the Dynamic Type?

Every occurrence of `?` behaves like a **distinct, existentially quantified** type variable.

```
let map (condition : Bool) (f : α -> β) (data : ?) : ? =
  if condition then
    List.map f data
  else
    Array.map f data
```

What is the Dynamic Type?

Every occurrence of ? behaves like a **distinct, existentially quantified** type variable.

```
let map (condition : Bool) (f : α -> β) (data : ?) : ? =
  if condition then
    List.map f data (* Here ? unifies to α list *)
  else
    Array.map f data
```

What is the Dynamic Type?

Every occurrence of `?` behaves like a **distinct, existentially quantified** type variable.

```
let map (condition : Bool) (f : α -> β) (data : ?) : ? =
  if condition then
    List.map f data (* Here ? unifies to α list *)
  else
    Array.map f data (* Here ? unifies to α array *)
```

What is the Dynamic Type?

Every occurrence of `?` behaves like a **distinct, existentially quantified** type variable.

```
let map (condition : Bool) (f : α -> β) (data : ?) : ? =
  if condition then
    List.map f data (* Here ? unifies to α list *)
  else
    Array.map f data (* Here ? unifies to α array *)
```

Main idea: interpret occurrences of `?` as arbitrary **type variables**.

Our Approach

1. Translate gradual types to **static types** (types without ?) **with variables**.

Our Approach

1. Translate gradual types to **static types** (types without ?) **with variables**.

2. Define **transitive** relations on gradual types, and in particular **"materialization"** which contains the **essence of gradual typing.**

Our Approach

1. Translate gradual types to **static types** (types without ?) **with variables**.

2. Define **transitive** relations on gradual types, and in particular **“materialization”** which contains the **essence of gradual typing.**

3. Embed materialization into **more and more complex systems** (Hindley-Milner, with subtyping, and with semantic subtyping).

Our Approach

1. Translate gradual types to **static types** (types without ?) **with variables**.

2. Define **transitive** relations on gradual types, and in particular **"materialization"** which contains the **essence of gradual typing.**

3. Embed materialization into **more and more complex systems** (Hindley-Milner, with subtyping, and with semantic subtyping).

Important remark: this translation is **only used** to define and compute relations, and **is not done in the source program**.

Discrimination and Materialization

We first define the **discrimination** of a gradual type:

$$\mathcal{D}(?) = \{X_1; X_2; \ldots\}$$

Discrimination and Materialization

We first define the **discrimination** of a gradual type:

$$\mathcal{D}(?) = \{X_1; X_2; \ldots\}$$

$$\begin{aligned}\mathcal{D}((\mathsf{Int} \to ?) \wedge ?) = \{&(\mathsf{Int} \to X_1) \wedge X_1;\\ &(\mathsf{Int} \to X_1) \wedge X_2;\\ &\ldots\}\end{aligned}$$

Discrimination and Materialization

We first define the **discrimination** of a gradual type:

$$\mathcal{D}(?) = \{X_1; X_2; \ldots\}$$

$$\mathcal{D}((\mathsf{Int} \to ?) \wedge ?) = \{(\mathsf{Int} \to X_1) \wedge X_1; \\ (\mathsf{Int} \to X_1) \wedge X_2; \\ \ldots\}$$

And we define **materialization** (which is the inverse of **precision**, as defined in Garcia [2013]):

$$\tau_1 \preccurlyeq \tau_2 \overset{\text{def}}{\iff} \exists T_1 \in \mathcal{D}(\tau_1), \sigma : \mathtt{Vars} \to \mathtt{GTypes},\ T_1\sigma = \tau_2$$

We first define the **discrimination** of a gradual type:

$$\mathcal{D}(?) = \{X_1; X_2; \ldots\}$$

$$\begin{aligned}\mathcal{D}((\mathsf{Int} \to ?) \wedge ?) = \{&(\mathsf{Int} \to X_1) \wedge X_1;\\ &(\mathsf{Int} \to X_1) \wedge X_2;\\ &\ldots\}\end{aligned}$$

And we define **materialization** (which is the inverse of **precision**, as defined in Garcia [2013]):

$$\tau_1 \preccurlyeq \tau_2 \overset{\text{def}}{\iff} \exists T_1 \in \mathcal{D}(\tau_1), \sigma : \texttt{Vars} \to \texttt{GTypes}, T_1\sigma = \tau_2$$

As well as **gradual subtyping**:

$$\tau_1 \leq \tau_2 \overset{\text{def}}{\iff} \exists (T_1, T_2) \in \mathcal{D}(\tau_1) \times \mathcal{D}(\tau_2), T_1 \leq_T T_2$$

Subtyping

Subtyping only allows us to move **inside** the dynamic world, or **inside** the static world. It **does not** allow crossing the barrier.

Subtyping

Subtyping only allows us to move **inside** the dynamic world, or **inside** the static world. It **does not** allow crossing the barrier.

As opposed to consistent subtyping, it is **transitive**:

$$? \leq ? \qquad ? \nleq \mathsf{Int} \qquad \mathsf{Int} \nleq ?$$

Subtyping

Subtyping only allows us to move **inside** the dynamic world, or **inside** the static world. It **does not** allow crossing the barrier.

As opposed to consistent subtyping, it is **transitive**:

$$? \leq ? \qquad ? \not\leq \mathsf{Int} \qquad \mathsf{Int} \not\leq ?$$

It can be used to handle unions and intersections, by **simply plugging-in** the static version of **semantic subtyping**:

$$? \leq ? \vee \mathsf{Int} \qquad \mathsf{Int} \wedge ? \leq ?$$

Materialization

Materialization is what allows us to **cross the barrier** from the dynamic world into the static world (**and only this way!**)

Materialization is what allows us to **cross the barrier** from the dynamic world into the static world (**and only this way!**)

$$? \precsim \tau \quad \text{for every } \tau$$

$$? \to ? \precsim \tau_1 \to \tau_2 \quad \text{for every } \tau_1, \tau_2$$

Materialization is what allows us to **cross the barrier** from the dynamic world into the static world (**and only this way!**)

$? \precsim \tau$ for every τ

$? \to ? \precsim \tau_1 \to \tau_2$ for every τ_1, τ_2

And it is **transitive**:

$$? \precsim ? \to ? \precsim ? \to \mathtt{Int} \precsim \mathtt{Int} \to \mathtt{Int}$$

Materialization

Materialization is what allows us to **cross the barrier** from the dynamic world into the static world (**and only this way!**)

$? \precsim \tau$ for every τ

$? \to ? \precsim \tau_1 \to \tau_2$ for every τ_1, τ_2

And it is **transitive**:

$$? \precsim ? \to ? \precsim ? \to \texttt{Int} \precsim \texttt{Int} \to \texttt{Int}$$

Therefore it can be embedded into a type system as a **subsumption rule**.

Declarative Type Systems

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

Declarative Type Systems

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \preccurlyeq \tau_2}{\Gamma \vdash e : \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2}$$

Declarative Type Systems

$$\frac{}{\Gamma, x : \forall\vec{\alpha}.\tau \vdash x : \tau\{\vec{\alpha} := \vec{t}\}} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \mathtt{Gen}_\Gamma(\tau_1) \vdash e_2 : \tau}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau}$$

Declarative Type Systems

$$\frac{}{\Gamma, x : \forall\vec{\alpha}.\tau \vdash x : \tau\{\vec{\alpha} := \vec{t}\}} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \texttt{Gen}_\Gamma(\tau_1) \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \preccurlyeq \tau_2}{\Gamma \vdash e : \tau_2}$$

Declarative Type Systems

$$\frac{}{\Gamma, x : \forall\vec{\alpha}.\tau \vdash x : \tau\{\vec{\alpha} := \vec{t}\}} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \mathtt{Gen}_\Gamma(\tau_1) \vdash e_2 : \tau}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \preccurlyeq \tau_2}{\Gamma \vdash e : \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2}$$

Declarative Type Systems

$$\frac{}{\Gamma, x : \forall\vec{\alpha}.\tau \vdash x : \tau\{\vec{\alpha} := \vec{t}\}} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \texttt{Gen}_\Gamma(\tau_1) \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \preccurlyeq \tau_2}{\Gamma \vdash e : \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2}$$

And as a bonus, we get the **static gradual guarantee** for free!

Back to the Example (1/2)

In the body of the function,

$$\Gamma \vdash \texttt{data} \;:(\alpha\ \texttt{array} \vee \alpha\ \texttt{list}) \wedge ?$$

Back to the Example (1/2)

In the body of the function,

$$\Gamma \vdash \texttt{data} \ : (\alpha \ \texttt{array} \vee \alpha \ \texttt{list}) \wedge \texttt{?}$$

By **subtyping**:

$$(\alpha \ \texttt{array} \vee \alpha \ \texttt{list}) \wedge \texttt{?} \leq \texttt{?}$$

Back to the Example (1/2)

In the body of the function,

$$\Gamma \vdash \texttt{data} \ : (\alpha \ \texttt{array} \vee \alpha \ \texttt{list}) \wedge \texttt{?}$$

By **subtyping**:

$$(\alpha \ \texttt{array} \vee \alpha \ \texttt{list}) \wedge \texttt{?} \leq \texttt{?}$$

And by **materialization**:

$$\texttt{?} \preccurlyeq \alpha \ \texttt{array}$$

Back to the Example (1/2)

In the body of the function,

$$\Gamma \vdash \texttt{data} \ : (\alpha \ \texttt{array} \vee \alpha \ \texttt{list}) \wedge \texttt{?}$$

By **subtyping**:

$$(\alpha \ \texttt{array} \vee \alpha \ \texttt{list}) \wedge \texttt{?} \leq \texttt{?}$$

And by **materialization**:

$$\texttt{?} \preccurlyeq \alpha \ \texttt{array}$$

Hence $\Gamma \vdash \texttt{data} \ : \alpha \ \texttt{array}$

Back to the Example (1/2)

In the body of the function,

$$\Gamma \vdash \texttt{data} \ : (\alpha \ \texttt{array} \vee \alpha \ \texttt{list}) \wedge ?$$

By **subtyping**:

$$(\alpha \ \texttt{array} \vee \alpha \ \texttt{list}) \wedge ? \leq ?$$

And by **materialization**:

$$? \preccurlyeq \alpha \ \texttt{array}$$

Hence $\Gamma \vdash \texttt{data} \ : \alpha \ \texttt{array}$

$\Longrightarrow$ `Array.map f data` is **well-typed.**

Back to the Example (2/2)

Now **from the outside**, consider a partial application `f`:

$$\Gamma \vdash \texttt{f} \;: ((\alpha\ \texttt{array} \vee \alpha\ \texttt{list}) \wedge ?) \to \texttt{t}$$

Back to the Example (2/2)

Now **from the outside**, consider a partial application f:

$$\Gamma \vdash \texttt{f} \; : ((\alpha\ \texttt{array} \vee \alpha\ \texttt{list}) \wedge ?) \to \texttt{t}$$

Let's say we want to apply it to a **string**. We need to **materialize** the type of f to $\texttt{string} \to \texttt{t}$.

Back to the Example (2/2)

Now **from the outside**, consider a partial application $\texttt{f}$:

$$\Gamma \vdash \texttt{f} \; : ((\alpha\ \texttt{array} \vee \alpha\ \texttt{list}) \wedge \texttt{?}) \to \texttt{t}$$

Let's say we want to apply it to a **string**. We need to **materialize** the type of $\texttt{f}$ to $\texttt{string} \to \texttt{t}$.

Simply materializing $\texttt{?}$ **does not work**:

$$((\alpha\ \texttt{array} \vee \alpha\ \texttt{list}) \wedge \texttt{string}) = \varnothing$$

Back to the Example (2/2)

Now **from the outside**, consider a partial application `f`:

$$\Gamma \vdash \texttt{f} \;\; : ((\alpha \; \texttt{array} \vee \alpha \; \texttt{list}) \wedge \texttt{?}) \to \texttt{t}$$

Let's say we want to apply it to a **string**. We need to **materialize** the type of `f` to $\texttt{string} \to \texttt{t}$.

Simply materializing `?` **does not work**:

$$((\alpha \; \texttt{array} \vee \alpha \; \texttt{list}) \wedge \texttt{string}) = \varnothing$$

Subtyping cannot be used either as it is **contravariant in the domain**:

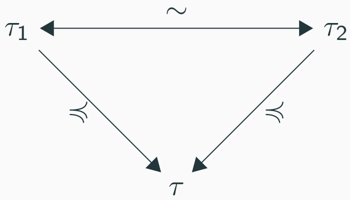
$$((\alpha \; \texttt{array} \vee \alpha \; \texttt{list}) \wedge \texttt{?}) \to \texttt{t} \not\leq \texttt{?} \to \texttt{t}$$

Is This Still Gradual Typing?

We **do not have consistency** anymore, and materialization only allows us to go **one way**.

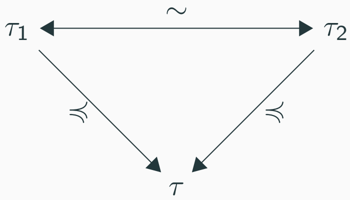
Is This Still Gradual Typing?

We **do not have consistency** anymore, and materialization only allows us to go **one way**.



Is This Still Gradual Typing?

We **do not have consistency** anymore, and materialization only allows us to go **one way**.

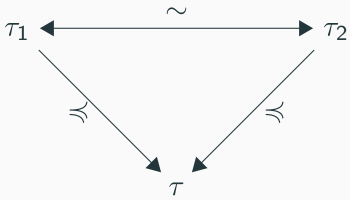


Propositions.

1- Every typable term in the system of Siek & Taha [2006] **can be given the same type** in our system.

Is This Still Gradual Typing?

We **do not have consistency** anymore, and materialization only allows us to go **one way**.



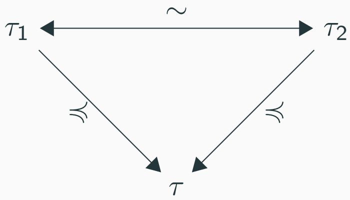
Propositions.

1- Every typable term in the system of Siek & Taha [2006] **can be given the same type** in our system.

2- Conversely, every typable term in our system **can be given a less-precise type** in the system of Siek & Taha [2006].

Is This Still Gradual Typing?

We **do not have consistency** anymore, and materialization only allows us to go **one way**.



Propositions.

1- Every typable term in the system of Siek & Taha [2006] **can be given the same type** in our system.

2- Conversely, every typable term in our system **can be given a less-precise type** in the system of Siek & Taha [2006].

3- Same results for the polymorphic system of Garcia & Cimini [2015].

Towards a Cast Language

We now want to compile our source language to a **cast language** that incorporates **casts** and **blame tracking**.

Towards a Cast Language

We now want to compile our source language to a **cast language** that incorporates **casts** and **blame tracking**.

$$(\lambda x : \texttt{?}.x\langle \texttt{?} \overset{l_1}{\Rightarrow} \mathsf{Int}\rangle + 1)\ (\texttt{true}\langle \mathsf{Bool} \overset{l_2}{\Rightarrow} \texttt{?}\rangle)$$

$$\hookrightarrow \texttt{true}\langle \mathsf{Bool} \overset{l_2}{\Rightarrow} \texttt{?}\rangle\langle \texttt{?} \overset{l_1}{\Rightarrow} \mathsf{Int}\rangle + 1$$

$$\hookrightarrow \texttt{blame}\ \ l_1$$

Towards a Cast Language

We now want to compile our source language to a **cast language** that incorporates **casts** and **blame tracking**.

$$(\lambda x : ?.x\langle ? \stackrel{l_1}{\Rightarrow} \mathsf{Int}\rangle + 1)\ (\mathtt{true}\langle \mathsf{Bool} \stackrel{l_2}{\Rightarrow} ?\rangle)$$

$$\hookrightarrow \mathtt{true}\langle \mathsf{Bool} \stackrel{l_2}{\Rightarrow} ?\rangle\langle ? \stackrel{l_1}{\Rightarrow} \mathsf{Int}\rangle + 1$$

$$\hookrightarrow \mathtt{blame}\ \ l_1$$

$$(\lambda x : \mathsf{Int}.x + 1)\ \langle \mathsf{Int} \to \mathsf{Int} \stackrel{l_1}{\Rightarrow} ? \to ?\rangle(\mathtt{true}\langle \mathsf{Bool} \stackrel{l_2}{\Rightarrow} ?\rangle)$$

$$\hookrightarrow (\mathtt{true}\langle \mathsf{Bool} \stackrel{l_2}{\Rightarrow} ?\rangle\langle ? \stackrel{\bar{l}_1}{\Rightarrow} \mathsf{Int}\rangle + 1)\langle \mathsf{Int} \stackrel{l_1}{\Rightarrow} ?\rangle$$

$$\hookrightarrow \mathtt{blame}\ \ \bar{l}_1$$

Towards a Cast Language

We now want to compile our source language to a **cast language** that incorporates **casts** and **blame tracking**.

$$(\lambda x : ?.x\langle ? \stackrel{l_1}{\Rightarrow} \mathsf{Int}\rangle + 1)\ (\mathtt{true}\langle \mathsf{Bool} \stackrel{l_2}{\Rightarrow} ?\rangle)$$

$$\hookrightarrow \mathtt{true}\langle \mathsf{Bool} \stackrel{l_2}{\Rightarrow} ?\rangle\langle ? \stackrel{l_1}{\Rightarrow} \mathsf{Int}\rangle + 1$$

$$\hookrightarrow \mathtt{blame}\ \ l_1$$

$$(\lambda x : \mathsf{Int}.x + 1)\ \langle \mathsf{Int} \to \mathsf{Int} \stackrel{l_1}{\Rightarrow} ? \to ?\rangle(\mathtt{true}\langle \mathsf{Bool} \stackrel{l_2}{\Rightarrow} ?\rangle)$$

$$\hookrightarrow (\mathtt{true}\langle \mathsf{Bool} \stackrel{l_2}{\Rightarrow} ?\rangle\langle ? \stackrel{\bar{l}_1}{\Rightarrow} \mathsf{Int}\rangle + 1)\langle \mathsf{Int} \stackrel{l_1}{\Rightarrow} ?\rangle$$

$$\hookrightarrow \mathtt{blame}\ \ \bar{l}_1$$

Blame tells us where an error occurred, and **in which way** the boundary was crossed.

Declarative Compilation

Principle: to every use of the materialization rule corresponds a cast.

Declarative Compilation

Principle: to every use of the materialization rule corresponds a cast.

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \preccurlyeq \tau_2}{\Gamma \vdash e : \tau_2}$$

Declarative Compilation

Principle: to every use of the materialization rule corresponds a cast.

$$\frac{\Gamma \vdash e : \tau_1 \;\; \mapsto e' \qquad \tau_1 \preccurlyeq \tau_2}{\Gamma \vdash e : \tau_2 \;\; \mapsto e' \; \langle \tau_1 \overset{l}{\Rightarrow} \tau_2 \rangle}$$

Declarative Compilation

Principle: to every use of the materialization rule corresponds a cast.

$$\frac{\Gamma \vdash e : \tau_1 \ \mapsto e' \qquad \tau_1 \preccurlyeq \tau_2}{\Gamma \vdash e : \tau_2 \ \mapsto e' \ \langle \tau_1 \overset{l}{\Rightarrow} \tau_2 \rangle}$$

Casts of the form $\langle \mathsf{Int} \to ? \overset{l}{\Rightarrow} ? \to \mathsf{Int} \rangle$ **are forbidden**.

Declarative Compilation

Principle: to every use of the materialization rule corresponds a cast.

$$\frac{\Gamma \vdash e : \tau_1 \;\; \mapsto e' \qquad \tau_1 \preccurlyeq \tau_2}{\Gamma \vdash e : \tau_2 \;\; \mapsto e' \; \langle \tau_1 \overset{I}{\Rightarrow} \tau_2 \rangle}$$

Casts of the form $\langle \text{Int} \to ? \overset{I}{\Rightarrow} ? \to \text{Int} \rangle$ **are forbidden**.

Moreover, the direction of the cast **can be enforced** in the typing rules:

$$\frac{\Gamma \vdash e : \tau_1 \qquad \begin{cases} p = I & \implies \tau_1 \preccurlyeq \tau_2 \\ p = \bar{I} & \implies \tau_2 \preccurlyeq \tau_1 \end{cases}}{\Gamma \vdash e \langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle : \tau_2}$$

Type Preservation and Blame Safety

Type preservation for the declarative compilation is immediate.

Type Preservation and Blame Safety

Type preservation for the declarative compilation is immediate.

Blame safety is an important result of the cast language that states that **only the dynamically-typed part** of the code can cause errors.

Type Preservation and Blame Safety

Type preservation for the declarative compilation is immediate.

Blame safety is an important result of the cast language that states that **only the dynamically-typed part** of the code can cause errors.

We only insert casts when crossing from **dynamic** to **static** code, and **precisely control** the direction of each cast throughout the execution. **This makes proving blame safety straightforward.**

Summary

– By interpreting ? as a type variable, we can define relations on gradual types **using existing definitions** on static types.

Summary

– By interpreting ? as a type variable, we can define relations on gradual types **using existing definitions** on static types.

– We presented a simple, straightforward way of **declaratively adding gradual typing** to existing type systems and compilation systems.

Summary

– By interpreting ? as a type variable, we can define relations on gradual types **using existing definitions** on static types.

– We presented a simple, straightforward way of **declaratively adding gradual typing** to existing type systems and compilation systems.

– We highlight a **direct correspondence** between compilation and type derivations.

Summary

– By interpreting ? as a type variable, we can define relations on gradual types **using existing definitions** on static types.

– We presented a simple, straightforward way of **declaratively adding gradual typing** to existing type systems and compilation systems.

– We highlight a **direct correspondence** between compilation and type derivations.

– The declarative systems enjoy **many** (almost) **free theorems** (blame safety, type preservation, static gradual guarantee).

Algorithmic Systems

static types $\mathcal{T}_t \ni t ::= \alpha \mid b \mid t \times t \mid t \to t$

gradual types $\mathcal{T}_\tau \ni \tau ::= ? \mid \alpha \mid b \mid \tau \times \tau \mid \tau \to \tau$

source language $e ::= x \mid c \mid \lambda x.\, e \mid \lambda x : \tau.\, e \mid e\ e \mid (e, e) \mid \pi_i\ e$
$\mid \text{let}\ \vec{\alpha}\ x = e\ \text{in}\ e$

cast language $E ::= \lambda^{\tau \to \tau} x.\, E \mid \text{let}\ x = E\ \text{in}\ E \mid \Lambda \vec{\alpha}.\, E \mid E\, [\vec{t}]$
$\mid E\langle \tau \overset{p}{\Rightarrow} \tau \rangle \mid \ldots$

Inference: Main Ideas

– Based on the works of **Pottier and Rémy** [2005], and of **Castagna et al.** [2016].

Inference: Main Ideas

– Based on the works of **Pottier and Rémy** [2005], and of **Castagna et al.** [2016].

– Our inference algorithm **only uses unification**, which differs from Garcia and Cimini [2015].

Inference: Main Ideas

– Based on the works of **Pottier and Rémy** [2005], and of **Castagna et al.** [2016].

– Our inference algorithm **only uses unification**, which differs from Garcia and Cimini [2015].

– We generate **structured constraints**, rewrite them to obtain a set of **unification and materialization constraints**, and solve them by **unification**.

Inference: Main Ideas

– Based on the works of **Pottier and Rémy** [2005], and of **Castagna et al.** [2016].

– Our inference algorithm **only uses unification**, which differs from Garcia and Cimini [2015].

– We generate **structured constraints**, rewrite them to obtain a set of **unification and materialization constraints**, and solve them by **unification**.

Note: we **never infer gradual types**, they can only be introduced by **explicit annotations**.

Inference: Structured Constraints

We first **generate constraints** of the form[1]:

$$C ::= (t \dot{\leq} t) \mid (\tau \dot{\preccurlyeq} \alpha) \mid (x \dot{\preccurlyeq} \alpha) \mid \text{def } x : \tau \text{ in } C \mid \exists \vec{\alpha}.\, C \mid C \wedge C$$

[1] Let constraints are omitted for the sake of simplicity

Inference: Structured Constraints

We first generate constraints of the form[1]:

$$C ::= (t \mathrel{\dot{\leq}} t) \mid (\tau \mathrel{\dot{\preccurlyeq}} \alpha) \mid (x \mathrel{\dot{\preccurlyeq}} \alpha) \mid \text{def } x \colon \tau \text{ in } C \mid \exists \vec{\alpha}.\, C \mid C \wedge C$$

$$\begin{aligned}
\langle\!\langle x \colon t \rangle\!\rangle &= \exists \alpha.\, (x \mathrel{\dot{\preccurlyeq}} \alpha) \wedge (\alpha \mathrel{\dot{\leq}} t) \\
\langle\!\langle (\lambda x.\, e) \colon t \rangle\!\rangle &= \exists \alpha_1, \alpha_2.\, (\text{def } x \colon \alpha_1 \text{ in } \langle\!\langle e \colon \alpha_2 \rangle\!\rangle) \wedge (\alpha_1 \mathrel{\dot{\preccurlyeq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t) \\
\langle\!\langle (\lambda x \colon \tau.\, e) \colon t \rangle\!\rangle &= \exists \alpha_1, \alpha_2.\, (\text{def } x \colon \tau \text{ in } \langle\!\langle e \colon \alpha_2 \rangle\!\rangle) \wedge (\tau \mathrel{\dot{\preccurlyeq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)
\end{aligned}$$

[1] Let constraints are omitted for the sake of simplicity

Inference: Structured Constraints

We first **generate constraints** of the form[1]:

$$C ::= (t \mathrel{\dot{\leq}} t) \mid (\tau \mathrel{\dot{\preccurlyeq}} \alpha) \mid (x \mathrel{\dot{\preccurlyeq}} \alpha) \mid \mathsf{def}\ x\colon \tau\ \mathsf{in}\ C \mid \exists \vec{\alpha}.\, C \mid C \wedge C$$

$$\begin{aligned}
\langle\!\langle x\colon t \rangle\!\rangle &= \exists \alpha.\, (x \mathrel{\dot{\preccurlyeq}} \alpha) \wedge (\alpha \mathrel{\dot{\leq}} t) \\
\langle\!\langle (\lambda x.\, e)\colon t \rangle\!\rangle &= \exists \alpha_1, \alpha_2.\, (\mathsf{def}\ x\colon \alpha_1\ \mathsf{in}\ \langle\!\langle e\colon \alpha_2 \rangle\!\rangle) \wedge (\alpha_1 \mathrel{\dot{\preccurlyeq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t) \\
\langle\!\langle (\lambda x\colon \tau.\, e)\colon t \rangle\!\rangle &= \exists \alpha_1, \alpha_2.\, (\mathsf{def}\ x\colon \tau\ \mathsf{in}\ \langle\!\langle e\colon \alpha_2 \rangle\!\rangle) \wedge (\tau \mathrel{\dot{\preccurlyeq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)
\end{aligned}$$

Note that $\langle\!\langle (\lambda x\colon ?.\, x)\colon \mathsf{Int} \to \mathsf{Int} \rangle\!\rangle$ **can be solved**, whereas $\langle\!\langle (\lambda x.\, x)\colon ? \to ? \rangle\!\rangle$ **cannot**.

[1] Let constraints are omitted for the sake of simplicity

Rewriting constraints

We then **rewrite the structured constraints** to obtain a set containing **type constraints**:

$$D ::= (t_1 \mathrel{\dot{\leq}} t_2) \mid (\tau \mathrel{\dot{\preccurlyeq}} \alpha)$$

Rewriting constraints

We then **rewrite the structured constraints** to obtain a set containing **type constraints**:

$$D ::= (t_1 \mathrel{\dot{\leq}} t_2) \mid (\tau \mathrel{\dot{\preccurlyeq}} \alpha)$$

$$\frac{}{\Gamma; \Delta \vdash (x \mathrel{\dot{\preccurlyeq}} \alpha) \rightsquigarrow \{\tau\{\vec{\alpha} := \vec{\beta}\} \mathrel{\dot{\preccurlyeq}} \alpha\}} \quad \begin{array}{l} \Gamma(x) = \forall \vec{\alpha}.\, \tau \\ \vec{\beta} \text{ fresh} \end{array}$$

Rewriting constraints

We then **rewrite the structured constraints** to obtain a set containing **type constraints**:

$$D ::= (t_1 \mathrel{\dot{\leq}} t_2) \mid (\tau \mathrel{\dot{\preccurlyeq}} \alpha)$$

$$\frac{}{\Gamma; \Delta \vdash (x \mathrel{\dot{\preccurlyeq}} \alpha) \rightsquigarrow \{\tau\{\vec{\alpha} := \vec{\beta}\} \mathrel{\dot{\preccurlyeq}} \alpha\}} \quad \begin{array}{l} \Gamma(x) = \forall \vec{\alpha}.\, \tau \\ \vec{\beta} \text{ fresh} \end{array}$$

$$\frac{(\Gamma, x\colon \tau); \Delta \vdash C \rightsquigarrow D}{\Gamma; \Delta \vdash \text{def } x\colon \tau \text{ in } C \rightsquigarrow D}$$

Solving constraints

Everything is finally solved using **unification**, by **replacing every occurence** of `?` in materialization constraints by a **distinct type variable**.

Solving constraints

Everything is finally solved using **unification**, by **replacing every occurence** of `?` in materialization constraints by a **distinct type variable**.

For example, the constraint

$$\texttt{?} \to \texttt{?} \to \texttt{?} \mathrel{\dot{\preccurlyeq}} \mathsf{Bool} \to \alpha$$

Solving constraints

Everything is finally solved using **unification**, by **replacing every occurence** of `?` in materialization constraints by a **distinct type variable**.

For example, the constraint

$$? \to ? \to ? \mathrel{\dot{\preccurlyeq}} \mathsf{Bool} \to \alpha$$

will become

$$X_1 \to X_2 \to X_3 \mathrel{\dot{\preccurlyeq}} \mathsf{Bool} \to \alpha$$

Solving constraints

Everything is finally solved using **unification**, by **replacing every occurence** of ? in materialization constraints by a **distinct type variable**.

For example, the constraint

$$? \to ? \to ? \mathrel{\dot{\preccurlyeq}} \mathsf{Bool} \to \alpha$$

will become

$$X_1 \to X_2 \to X_3 \mathrel{\dot{\preccurlyeq}} \mathsf{Bool} \to \alpha$$

and solving it will return the unifier

$$\theta : X_1 \mapsto \mathsf{Bool}; X_2 \mapsto \beta; X_3 \mapsto \gamma; \alpha \mapsto (\beta \to \gamma)$$

Compilation and Results

To summarize, given an expression e, and a constraint derivation $\mathcal{D}$ of $\Gamma; \Delta \vdash \langle\!\langle e \colon t \rangle\!\rangle \rightsquigarrow D$, we can **compute a unifier** θ satisfying $\mathcal{D}$.

Compilation and Results

To summarize, given an expression e, and a constraint derivation $\mathcal{D}$ of $\Gamma; \Delta \vdash \langle\!\langle e \colon t \rangle\!\rangle \rightsquigarrow D$, we can **compute a unifier** θ satisfying $\mathcal{D}$.

This derivation and the associated unifier **can be used to compile** e in a straightforward way: **to every materialization constraint introduced in $\mathcal{D}$ corresponds a cast.**

Compilation and Results

To summarize, given an expression e, and a constraint derivation $\mathcal{D}$ of $\Gamma; \Delta \vdash \langle\!\langle e\colon t \rangle\!\rangle \rightsquigarrow D$, we can **compute a unifier** θ satisfying $\mathcal{D}$.

This derivation and the associated unifier **can be used to compile** e in a straightforward way: **to every materialization constraint introduced in $\mathcal{D}$ corresponds a cast.**

$$(\!|x|\!)^{\mathcal{D}}_{\theta} = x\langle \tau\theta \overset{I}{\Rightarrow} \alpha\theta \rangle \quad \text{if} \quad \mathcal{D} = \Gamma; \Delta \vdash \langle\!\langle x\colon t \rangle\!\rangle \rightsquigarrow \{(\tau \mathrel{\dot{\preccurlyeq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}$$

Compilation and Results

To summarize, given an expression e, and a constraint derivation $\mathcal{D}$ of $\Gamma; \Delta \vdash \langle\!\langle e \colon t \rangle\!\rangle \rightsquigarrow D$, we can **compute a unifier** θ satisfying $\mathcal{D}$.

This derivation and the associated unifier **can be used to compile** e in a straightforward way: **to every materialization constraint introduced in $\mathcal{D}$ corresponds a cast.**

$$(\!|x|\!)^{\mathcal{D}}_{\theta} = x\langle \tau\theta \stackrel{l}{\Rightarrow} \alpha\theta \rangle \quad \text{if} \quad \mathcal{D} = \Gamma; \Delta \vdash \langle\!\langle x \colon t \rangle\!\rangle \rightsquigarrow \{(\tau \mathrel{\dot{\preccurlyeq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}$$

Inference (and compilation) for this system is **sound**, **type-preserving** and **complete** w.r.t. the declarative system.

Part 2: Adding subtyping

We saw that, declaratively, **adding subtyping** is just a matter of adding **one subsumption rule**.

Part 2: Adding subtyping

We saw that, declaratively, **adding subtyping** is just a matter of adding **one subsumption rule**.

Constraint generation is also unchanged, unification constraints just become **subtyping constraints.**

We saw that, declaratively, **adding subtyping** is just a matter of adding **one subsumption rule**.

Constraint generation is also unchanged, unification constraints just become **subtyping constraints.**

However, to **solve constraints** such as $\{(\alpha \dot{\leq} t_1), (\alpha \dot{\leq} t_2)\}$ we have to compute **greatest lower bounds**.

We saw that, declaratively, **adding subtyping** is just a matter of adding **one subsumption rule**.

Constraint generation is also unchanged, unification constraints just become **subtyping constraints.**

However, to **solve constraints** such as $\{(\alpha \dot{\leq} t_1), (\alpha \dot{\leq} t_2)\}$ we have to compute **greatest lower bounds**.

For example,

```
fun x -> if (fst x) then (1 + snd x) else x
```

should be of type `(Bool`$\times$`Int)` $\rightarrow$ `( Int | (Bool`$\times$`Int) )`

The types become:

static types	$t ::= \alpha \mid b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0}$
gradual types	$\tau ::= ? \mid \alpha \mid b \mid \tau \times \tau \mid \tau \to \tau \mid \tau \vee \tau \mid \neg \tau \mid \mathbb{0}$

Part 3: Adding Set-Theoretic Types

The types become:

$$\text{static types} \quad t ::= \alpha \mid b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0}$$

$$\text{gradual types} \quad \tau ::= ? \mid \alpha \mid b \mid \tau \times \tau \mid \tau \to \tau \mid \tau \vee \tau \mid \neg \tau \mid \mathbb{0}$$

Constraints are **unchanged**. However, the inference algorithm is now based on the **tallying algorithm** of Castagna et al. [2015], rather than unification (but the principle is the same).

$$\{(\alpha \mathrel{\dot{\leq}} t_1), (\alpha \mathrel{\dot{\leq}} t_2)\} \simeq \{(\alpha \mathrel{\dot{\leq}} t_1 \wedge t_2)\}$$

Part 3: Adding Set-Theoretic Types

The types become:

$$\begin{array}{rl} \text{static types} & t ::= \alpha \mid b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0} \\ \text{gradual types} & \tau ::= ? \mid \alpha \mid b \mid \tau \times \tau \mid \tau \to \tau \mid \tau \vee \tau \mid \neg \tau \mid \mathbb{0} \end{array}$$

Constraints are **unchanged**. However, the inference algorithm is now based on the **tallying algorithm** of Castagna et al. [2015], rather than unification (but the principle is the same).

$$\{(\alpha \mathrel{\dot{\leq}} t_1), (\alpha \mathrel{\dot{\leq}} t_2)\} \simeq \{(\alpha \mathrel{\dot{\leq}} t_1 \wedge t_2)\}$$

Soundness still holds for the inference algorithm, but **completeness no longer holds**.

A Remark About The Decidability of Subtyping

Recall that subtyping is defined as an **existential quantification.**

A Remark About The Decidability of Subtyping

Recall that subtyping is defined as an **existential quantification.**

However, we show that it reduces **in linear time** to subtyping **on static types.**

A Remark About The Decidability of Subtyping

Recall that subtyping is defined as an **existential quantification.**

However, we show that it reduces **in linear time** to subtyping **on static types.**

We can replace all the occurrences of ? of the same polarity by the same variable.

$$(? \to ?) \vee ? \quad \mapsto \quad (X_0 \to X_1) \vee X_1$$

A Remark About The Decidability of Subtyping

Recall that subtyping is defined as an **existential quantification.**

However, we show that it reduces **in linear time** to subtyping **on static types.**

We can replace all the occurrences of ? of the same polarity by the same variable.

$$(? \to ?) \vee ? \quad \mapsto \quad (X_0 \to X_1) \vee X_1$$

This is enough to decide subtyping

Some Remarks About Semantics (1/2)

The semantics of the cast calculus for **HM without subtyping** are basically the same as those presented by Siek et al. [2015].

Some Remarks About Semantics (1/2)

The semantics of the cast calculus for **HM without subtyping** are basically the same as those presented by Siek et al. [2015].

$$
\begin{array}{rrcll}
\text{[ExpandL]} & V\langle \tau \overset{p}{\Rightarrow} ?\rangle & \hookrightarrow & V\langle \tau \overset{p}{\Rightarrow} \mathit{gnd}(\tau)\rangle\langle \mathit{gnd}(\tau) \overset{p}{\Rightarrow} ?\rangle & \\
\text{[Collapse]} & V\langle \rho \overset{p}{\Rightarrow} ?\rangle\langle ? \overset{q}{\Rightarrow} \rho'\rangle & \hookrightarrow & V & \text{if } \rho = \rho' \\
\text{[Blame]} & V\langle \rho \overset{p}{\Rightarrow} ?\rangle\langle ? \overset{q}{\Rightarrow} \rho'\rangle & \hookrightarrow & \texttt{blame}\ q & \text{if } \rho \neq \rho'
\end{array}
$$

$$
\mathit{gnd}(\tau_1 \to \tau_2) = ? \to ? \qquad \mathit{gnd}(\tau_1 \times \tau_2) = ? \times ? \qquad \mathit{gnd}(b) = b
$$

Some Remarks About Semantics (1/2)

The semantics of the cast calculus for **HM without subtyping** are basically the same as those presented by Siek et al. [2015].

$$
\begin{array}{rrcll}
\text{[ExpandL]} & V\langle \tau \overset{p}{\Rightarrow} ?\rangle & \hookrightarrow & V\langle \tau \overset{p}{\Rightarrow} gnd(\tau)\rangle\langle gnd(\tau) \overset{p}{\Rightarrow} ?\rangle & \\
\text{[Collapse]} & V\langle \rho \overset{p}{\Rightarrow} ?\rangle\langle ? \overset{q}{\Rightarrow} \rho'\rangle & \hookrightarrow & V & \text{if } \rho = \rho' \\
\text{[Blame]} & V\langle \rho \overset{p}{\Rightarrow} ?\rangle\langle ? \overset{q}{\Rightarrow} \rho'\rangle & \hookrightarrow & \texttt{blame}\ q & \text{if } \rho \neq \rho'
\end{array}
$$

$$gnd(\tau_1 \to \tau_2) = ? \to ? \qquad gnd(\tau_1 \times \tau_2) = ? \times ? \qquad gnd(b) = b$$

Adding subtyping is just a matter of allowing $\rho \leq \rho'$.

Some Remarks About Semantics (2/2)

Adding set-theoretic types is more complicated, mostly because we need to take into account unions and intersections containing ?.

Some Remarks About Semantics (2/2)

Adding set-theoretic types is more complicated, mostly because we need to take into account unions and intersections containing ?.

We defined a **grounding operator** $^{\tau_1}/_{\tau_2}$ to compute the intermediate type of a cast.

$$^{(\mathsf{Int} \to \mathsf{Int}) \vee (\mathsf{Bool} \to \mathsf{Bool})}/_{(\mathsf{Int} \to \mathsf{Int}) \vee ?} = (\mathsf{Int} \to \mathsf{Int}) \vee (? \to ?)$$

Some Remarks About Semantics (2/2)

Adding set-theoretic types is more complicated, mostly because we need to take into account unions and intersections containing ?.

We defined a **grounding operator** ${}^{\tau_1}/_{\tau_2}$ to compute the intermediate type of a cast.

$$^{(\mathsf{Int} \to \mathsf{Int}) \vee (\mathsf{Bool} \to \mathsf{Bool})}/_{(\mathsf{Int} \to \mathsf{Int}) \vee ?} = (\mathsf{Int} \to \mathsf{Int}) \vee (? \to ?)$$

$$[\mathsf{ExpandL}] \quad V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \quad \hookrightarrow \quad V\langle \tau_1 \overset{p}{\Rightarrow} {}^{\tau_1}/_{\tau_2} \rangle \langle {}^{\tau_1}/_{\tau_2} \overset{p}{\Rightarrow} \tau_2 \rangle$$

Some Remarks About Semantics (2/2)

Adding set-theoretic types is more complicated, mostly because we need to take into account unions and intersections containing ?.

We defined a **grounding operator** ${}^{\tau_1}/_{\tau_2}$ to compute the intermediate type of a cast.

$$(\mathsf{Int} \to \mathsf{Int}) \vee (\mathsf{Bool} \to \mathsf{Bool})/_{(\mathsf{Int} \to \mathsf{Int}) \vee ?} = (\mathsf{Int} \to \mathsf{Int}) \vee (? \to ?)$$

$$[\text{ExpandL}] \quad V\langle \tau_1 \overset{p}{\Rightarrow} \tau_2 \rangle \quad \hookrightarrow \quad V\langle \tau_1 \overset{p}{\Rightarrow} {}^{\tau_1}/_{\tau_2} \rangle \langle {}^{\tau_1}/_{\tau_2} \overset{p}{\Rightarrow} \tau_2 \rangle$$

The full semantics are **conservative**, but complicated and contain **six additional rules to handle corner cases**.

Summary

– We defined a **sound and complete** type inference algorithm for a gradually-typed version of ML.

Summary

– We defined a **sound and complete** type inference algorithm for a gradually-typed version of ML.

– By interpreting once again ? as a type variable, the aforementionned inference algorithm **reuses existing unification algorithms**.

Summary

– We defined a **sound and complete** type inference algorithm for a gradually-typed version of ML.

– By interpreting once again ? as a type variable, the aforementionned inference algorithm **reuses existing unification algorithms**.

– We also gave a **sound** inference algorithm for an extension of this language with set-theoretic types, which **reuses the tallying algorithm**.

Summary

– We defined a **sound and complete** type inference algorithm for a gradually-typed version of ML.

– By interpreting once again ? as a type variable, the aforementionned inference algorithm **reuses existing unification algorithms**.

– We also gave a **sound** inference algorithm for an extension of this language with set-theoretic types, which **reuses the tallying algorithm**.

– We provided **sound semantics** for a cast calculus with set-theoretic gradual types and polymorphism.

The End

Thanks for listening!

Comments, questions, suggestions?