Gradual Typing: A New Perspective

With polymorphism, unions, intersections, and much more

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

18 February 2019
Let's write a map, that can work on both arrays and lists depending on a condition:

```plaintext
let map (condition : Bool) (f : α → β) (data : ) : =
```

Runtime checks or casts are then inserted automatically by the compiler.
Let's write a map, that can work on both arrays and lists depending on a condition:

```ocaml
let map (condition : Bool) (f : \alpha \to \beta) (data : ) : =
  if condition then
    List.map f data
  else
    Array.map f data
```

Runtime checks or casts are then inserted automatically by the compiler.
Let’s write a map, that can work on both arrays and lists depending on a condition:

```ml
let map (condition : Bool) (f : α -> β) (data : ?) : =
  if condition then
    List.map f data
  else
    Array.map f data
```
Let’s write a map, that can work on both arrays and lists depending on a condition:

```ml
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:

```ml
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 “?”. 

\[
\text{Int} \rightarrow ? \rightarrow ? \rightarrow \text{Int} \rightarrow ? \rightarrow \text{Bool}
\]
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:

\[
? \ll ? \rightarrow ? \ll \text{Int} \rightarrow ? \ll \text{Int} \rightarrow \text{Bool}
\]
Sometimes this **gradualization** is too **coarse**

```ocaml
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"
```
Sometimes this **gradualization** is too **coarse**

```ocaml
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!
Sometimes this **gradualization** is too **coarse**

```ocaml
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**.
let map (condition : Bool) (f : α -> β) (data : ) : =
  if condition then
    List.map f data
  else
    Array.map f data
let map (condition : Bool) (f : α -> β) (data : (α array ∨ α 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 ∨ α list ≠ α array.
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
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 \textbf{not well-typed} without additional checks, since \(α\ array \lor α\ list \not\subseteq α\ array\).
We need to explicitly deconstruct the union:

```ocaml
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.
We need to explicitly **deconstruct the union**:

```plaintext
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**:

```haskell
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**.
− **Types with connectives** \((\lor, \land, \neg)\)
Set-Theoretic Types Summarized

- **Types with connectives** ($\lor$, $\land$, $\neg$)

- Useful for overloading, branching, but often syntactically heavy.
Set-Theoretic Types Summarized

- **Types with connectives** ($\lor$, $\land$, $\neg$)

- Useful for overloading, branching, but often syntactically heavy.

$$(\text{Int} \to \text{Int}) \land (\text{Bool} \to \text{Bool}) = \text{overloaded function}$$
Set-Theoretic Types Summarized

- **Types with connectives** ($\lor, \land, \neg$)

- Useful for overloading, branching, but often syntactically heavy.

$$(\text{Int} \rightarrow \text{Int}) \land (\text{Bool} \rightarrow \text{Bool}) = \text{overloaded function}$$

if $x$ then 3 else true : $\text{Int} \lor \text{Bool}$
Set-Theoretic Types Summarized

- **Types with connectives** \((\lor, \land, \neg)\)

- Useful for overloading, branching, but often syntactically heavy.

\[(\text{Int} \to \text{Int}) \land (\text{Bool} \to \text{Bool}) = \text{overloaded function}\]

\[\text{if } x \text{ then } 3 \text{ else true : Int } \lor \text{ Bool}\]

- In **Semantic subtyping**, 
  
  Types \(\simeq\) Sets of values
  
  Subtyping \(\simeq\) Set-containment
## Pros and Cons

<table>
<thead>
<tr>
<th>Set-theoretic types</th>
<th>Gradual types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Safe</strong></td>
<td>Unsafe</td>
</tr>
<tr>
<td><strong>Expressive</strong></td>
<td>Too coarse</td>
</tr>
<tr>
<td>Verbose</td>
<td><strong>Light</strong></td>
</tr>
<tr>
<td>Restrictive</td>
<td><strong>Permissive</strong></td>
</tr>
<tr>
<td>Set-theoretic types</td>
<td>Gradual types</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td><strong>Safe</strong></td>
<td>Unsafe</td>
</tr>
<tr>
<td><strong>Expressive</strong></td>
<td>Too coarse</td>
</tr>
<tr>
<td>Verbose</td>
<td><strong>Light</strong></td>
</tr>
<tr>
<td>Restrictive</td>
<td>Permissive</td>
</tr>
</tbody>
</table>

Can we get the **best of both worlds?**
let map condition f
    (data : (α list ∨ α array)) =
    if condition then
        List.map f data
    else
        Array.map f data
let map condition f
    (data : (α list ∨ α array) ∧ ?) =
    if condition then
        List.map f data
    else
        Array.map f data
let map condition f
  (data : (α list ∨ α array) ∧ ?) =
  if condition then
    List.map f data
  else
    Array.map f data

– By subtyping, (α list ∨ α array) ∧ ? ≤ ?.
Mixing the Two

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

- By **subtyping**, 
  \[(α list ∨ α array) ∧ ? ≤ ?\].

- Can only be used with **lists or arrays**

- No need for **manual type checks**
Mixing the Two

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

- By **subtyping**, \((α \text{ list } ∨ α \text{ array}) \land ? \leq ?\).
- 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

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

- By **subtyping**, \((α \text{ list} ∨ α \text{ array}) ∧ ? \leq ?\).
- 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

```plaintext
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, \((α \text{ list} \lor α \text{ array}) \land ? \leq ?\).
- 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!)
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**, \((α \text{ list} \lor α \text{ array}) \land \text{?} \leq \text{?}\).
- 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!)
1. Define a \textit{subtype-consistency} relation $\lesssim$.
How is it Usually Done?

1. Define a **subtype-consistency** relation $\sim$.

   This relation is not transitive! $\sim \tau \sim$ for all $\tau$.
How is it Usually Done?

1. Define a \textit{subtype-consistency} relation $\sim \leq$.

   \textbf{This relation is not transitive!} $\sim \leq \tau \sim \leq \tau$ for all $\tau$

2. Embed this relation into typing rules.

   \[
   \frac{
   \Gamma \vdash e_1 : \tau_1 \rightarrow \tau'_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \sim \leq \tau_1
   }{
   \Gamma \vdash e_1 \ e_2 : \tau'_1
   }\]
How is it Usually Done?

1. Define a **subtype-consistency** relation $\lesssim$.

   **This relation is not transitive!** $\lesssim \tau \lesssim ?$ for all $\tau$

2. Embed this relation into typing rules.

   \[
   \begin{array}{c}
   \Gamma \vdash e_1 : \tau_1 \\
   \Gamma \vdash e_2 : \tau_2 \\
   \tau_2 \lesssim \text{dom}(\tau_1)
   \end{array}
   \Rightarrow
   \begin{array}{c}
   \Gamma \vdash e_1 \ e_2 : \tau_1 \circ \tau_2
   \end{array}
   \]

   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.

```haskell
let map (condition : Bool) (f : α -> β) (data : ?) : ? =
  if condition then
    List.map f data
  else
    Array.map f data
```
Every occurrence of ? behaves like a distinct, existentially quantified type variable.

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

```ocaml
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 ?) \textbf{with variables}.

2. Define \textbf{transitive} relations on gradual types, and in particular “materialization” which contains the \textbf{essence of gradual typing}.
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.
We first define the **discrimination** of a gradual type:

\[ D(?) = \left\{ X_1; X_2; \ldots \right\} \]
Discrimination and Materialization

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

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

\[ \mathcal{D}((\text{Int} \rightarrow ?) \land ?) = \{(\text{Int} \rightarrow X_1) \land X_1; (\text{Int} \rightarrow X_1) \land X_2; \ldots\} \]
We first define the **discrimination** of a gradual type:

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

\[
\mathcal{D}((\text{Int} \rightarrow ?) \land ?) = \{(\text{Int} \rightarrow X_1) \land X_1;
(\text{Int} \rightarrow X_1) \land X_2;
\ldots\}\n\]

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

\[
\tau_1 \ll \tau_2 \iff \exists T_1 \in \mathcal{D}(\tau_1), \sigma : \text{Vars} \rightarrow \text{GTypes}, T_1\sigma = \tau_2
\]
We first define the **discrimination** of a gradual type:

\[ D(?) = \{ X_1; X_2; \ldots \} \]

\[ D((\text{Int} \rightarrow ?) \land ?) = \{(\text{Int} \rightarrow X_1) \land X_1; \]
\[ (\text{Int} \rightarrow X_1) \land X_2; \]
\[ \ldots \} \]

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

\[ \tau_1 \ll \tau_2 \iff \exists T_1 \in D(\tau_1), \sigma : \text{Vars} \rightarrow \text{GTypes}, T_1\sigma = \tau_2 \]

As well as **gradual subtyping**:

\[ \tau_1 \leq \tau_2 \iff \exists (T_1, T_2) \in D(\tau_1) \times D(\tau_2), T_1 \leq_T T_2 \]
Subtyping only allows us to move inside the dynamic world, or inside the static world. It does not allow crossing the barrier.
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:

\[
? \leq ? \quad ? \not\leq \text{Int} \quad \text{Int} \not\leq ?
\]
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:

\[
? \leq ? \quad ? \not\leq \text{Int} \quad \text{Int} \not\leq ?
\]

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

\[
? \leq ? \lor \text{Int} \quad \text{Int} \land ? \leq ?
\]
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!)

\[ ? \sqsubseteq \tau \quad \text{for every } \tau \]

\[ ? \rightarrow ? \sqsubseteq \tau_1 \rightarrow \tau_2 \quad \text{for every } \tau_1, \tau_2 \]
Materialization is what allows us to cross the barrier from the dynamic world into the static world (and only this way!)

? $\ll \tau$ for every $\tau$

? $\rightarrow$ ? $\ll \tau_1 \rightarrow \tau_2$ for every $\tau_1, \tau_2$

And it is transitive:

? $\ll$ ? $\rightarrow$ ? $\ll$ ? $\rightarrow$ Int $\ll$ Int $\rightarrow$ Int
**Materialization** is what allows us to **cross the barrier** from the dynamic world into the static world (and only this way!)

\[ ? \preceq \tau \quad \text{for every } \tau \]
\[ ? \rightarrow ? \preceq \tau_1 \rightarrow \tau_2 \quad \text{for every } \tau_1, \tau_2 \]

And it is **transitive**:

\[ ? \preceq ? \rightarrow ? \preceq ? \rightarrow \text{Int} \preceq \text{Int} \rightarrow \text{Int} \]

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

\[
\frac{\Gamma, x : \tau \vdash x : \tau}{\Gamma, \tau_1 \vdash e : \tau_2}
\]

\[
\frac{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2}{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}
\]

\[
\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2}
\]

And as a bonus, we get the static gradual guarantee for free!
Declarative Type Systems

\[\Gamma, x : \tau \vdash x : \tau\]

\[\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2\]

\[\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1\]

\[\Gamma \vdash e_1 \; e_2 : \tau_2\]

\[\Gamma \vdash e : \tau_1 \quad \tau_1 \preceq \tau_2\]

\[\Gamma \vdash e : \tau_2\]

\[\Gamma \vdash e : \tau_1 \quad \tau_1 \leq \tau_2\]

\[\Gamma \vdash e : \tau_2\]
Declarative Type Systems

\[ \Gamma, x : \forall \vec{\alpha}.\tau \vdash x : \tau\{\vec{\alpha} := \vec{t}\} \]

\[ \Gamma, x : \tau_1 \vdash e : \tau_2 \]

\[ \Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \]

\[ \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1 \]

\[ \Gamma \vdash e_1 \ e_2 : \tau_2 \]

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \text{Gen}_\Gamma(\tau_1) \vdash e_2 : \tau \]

\[ \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau \]
Declarative Type Systems

\[
\Gamma, x : \forall \vec{a}. \tau \vdash x : \tau \{ \vec{a} := \vec{t} \}
\]

\[
\Gamma, x : \tau_1 \vdash e : \tau_2
\]

\[
\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2
\]

\[
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1
\]

\[
\Gamma \vdash e_1 \ e_2 : \tau_2
\]

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \text{Gen}_\Gamma(\tau_1) \vdash e_2 : \tau
\]

\[
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau
\]

\[
\Gamma \vdash e : \tau_1 \quad \tau_1 \preceq \tau_2
\]

\[
\Gamma \vdash e : \tau_2
\]

And as a bonus, we get the static gradual guarantee for free!
Declarative Type Systems

\[
\Gamma, x : \forall \vec{\alpha} \tau \vdash x : \tau \{ \vec{\alpha} := \bar{t} \}
\]

\[
\Gamma, x : \tau_1 \vdash e : \tau_2
\]

\[
\Gamma \vdash \lambda x . e : \tau_1 \rightarrow \tau_2
\]

\[
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1
\]

\[
\Gamma \vdash e_1 \ e_2 : \tau_2
\]

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \text{Gen}_\Gamma(\tau_1) \vdash e_2 : \tau
\]

\[
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau
\]

\[
\Gamma \vdash e : \tau_1 \quad \tau_1 \preceq \tau_2
\]

\[
\Gamma \vdash e : \tau_2
\]

\[
\Gamma \vdash e : \tau_1 \quad \tau_1 \leq \tau_2
\]

\[
\Gamma \vdash e : \tau_2
\]
Declarative Type Systems

\[
\frac{\Gamma, x : \forall \vec{\alpha} . \tau \vdash x : \tau \{ \vec{\alpha} := \vec{t} \}}{\Gamma, x : \forall \vec{\alpha} . \tau \vdash x : \tau}
\]

\[
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x . e : \tau_1 \to \tau_2}
\]

\[
\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2}
\]

\[
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \text{Gen}_\Gamma(\tau_1) \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}
\]

\[
\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \preceq \tau_2}{\Gamma \vdash e : \tau_2}
\]

\[
\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2}
\]

And as a bonus, we get the **static gradual guarantee** for free!
In the body of the function,

\[ \Gamma \vdash \text{data} : (\alpha \text{ array} \lor \alpha \text{ list}) \land ? \]
In the body of the function,

\[ \Gamma \vdash \text{data} : (\alpha \text{ array} \lor \alpha \text{ list}) \land ? \]

By subtyping:

\[ (\alpha \text{ array} \lor \alpha \text{ list}) \land ? \leq ? \]
In the body of the function,

\[ \Gamma \vdash \text{data} : (\alpha \text{ array} \lor \alpha \text{ list}) \land ? \]

By subtyping:

\[ (\alpha \text{ array} \lor \alpha \text{ list}) \land ? \leq ? \]

And by materialization:

\[ ? \preceq \alpha \text{ array} \]
In the body of the function,

\[ \Gamma \vdash \text{data} : (\alpha \text{ array} \lor \alpha \text{ list}) \land ? \]

By subtyping:

\[ (\alpha \text{ array} \lor \alpha \text{ list}) \land ? \leq ? \]

And by materialization:

\[ ? \preceq \alpha \text{ array} \]

Hence \( \Gamma \vdash \text{data} : \alpha \text{ array} \)
In the body of the function,

\[ \Gamma \vdash \text{data} : (\alpha \text{ array} \lor \alpha \text{ list}) \land ? \]

By subtyping:

\[ (\alpha \text{ array} \lor \alpha \text{ list}) \land ? \leq ? \]

And by materialization:

\[ ? \preceq \alpha \text{ array} \]

Hence \( \Gamma \vdash \text{data} : \alpha \text{ array} \)

\[ \implies \text{Array.map f data is well-typed.} \]
Now **from the outside**, consider a partial application $f$:

$$
\Gamma \vdash f : ((\alpha \text{ array} \lor \alpha \text{ list}) \land ?) \rightarrow t
$$
Now from the outside, consider a partial application \( f \):

\[
\Gamma \vdash f : ((\alpha \text{ array} \lor \alpha \text{ list}) \land ?) \rightarrow t
\]

Let's say we want to apply it to a string. We need to materialize the type of \( f \) to string \( \rightarrow t \).
Now from the outside, consider a partial application \( f : \) 

\[ \Gamma \vdash f : ((\alpha \text{ array} \lor \alpha \text{ list}) \land ?) \rightarrow t \]

Let's say we want to apply it to a string. We need to materialize the type of \( f \) to \( \text{string} \rightarrow t \).

Simply materializing \( ? \) does not work: 

\[ ((\alpha \text{ array} \lor \alpha \text{ list}) \land \text{string}) = \emptyset \]
Now from the outside, consider a partial application $f$:

$$\Gamma \vdash f : ((\alpha \text{ array} \lor \alpha \text{ list}) \land ?) \rightarrow t$$

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

Simply materializing $?$ does not work:

$$((\alpha \text{ array} \lor \alpha \text{ list}) \land \text{string}) = \emptyset$$

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

$$((\alpha \text{ array} \lor \alpha \text{ list}) \land ?) \rightarrow t \not\subseteq ? \rightarrow t$$
We do not have consistency anymore, and materialization only allows us to go one way.
We do not have consistency anymore, and materialization only allows us to go one way.

\[
\begin{array}{c}
\tau_1 \\
\sim \\
\tau \\
\sim \\
\tau_2
\end{array}
\]
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.

\[
\begin{array}{c}
\tau_1 \sim \tau_2 \\
\tau_1 \lessdot \tau \\
\tau \lessdot \tau_2
\end{array}
\]

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

\[
\tau_1 \sim \tau_2
\]

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

\[
\lambda x : \text{?}. x \langle \text{?} \Rightarrow \text{Int} \rangle + 1 \rangle (\text{true} \langle \text{Bool} \Rightarrow \text{?} \rangle)
\]

\[\rightarrow \text{true} \langle \text{Bool} \Rightarrow \text{?} \rangle \langle \text{?} \Rightarrow \text{Int} \rangle + 1 \]

\[\rightarrow \text{blame } l_1 \]
Towards a Cast Language

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

\[
(\lambda x : ?. x\langle ? \xrightarrow{l_1} \text{Int} \rangle + 1) \ (\text{true}\langle \text{Bool} \xrightarrow{l_2} ? \rangle)
\]

\[
\hookrightarrow \text{true}\langle \text{Bool} \xrightarrow{l_2} ? \rangle \langle ? \xrightarrow{l_1} \text{Int} \rangle + 1
\]

\[
\hookrightarrow \text{blame} \ l_1
\]

\[
(\lambda x : \text{Int}. x + 1) \ \langle \text{Int} \rightarrow \text{Int} \xrightarrow{l_1} ? \rightarrow ? \rangle (\text{true}\langle \text{Bool} \xrightarrow{l_2} ? \rangle)
\]

\[
\hookrightarrow (\text{true}\langle \text{Bool} \xrightarrow{l_2} ? \rangle \langle ? \xrightarrow{\overline{l}_1} \text{Int} \rangle + 1) \langle \text{Int} \xrightarrow{l_1} ? \rangle
\]

\[
\hookrightarrow \text{blame} \ \overline{l}_1
\]
Towards a Cast Language

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

\[
(\lambda x : ?. x \langle ? \xrightarrow{l_1} \text{Int} \rangle + 1) \ (\text{true}\langle \text{Bool} \xrightarrow{l_2} ? \rangle) \\
\quad \leftarrow \text{true}\langle \text{Bool} \xrightarrow{l_2} ? \rangle\langle ? \xrightarrow{l_1} \text{Int} \rangle + 1 \\
\quad \leftarrow \text{blame } l_1
\]

\[
(\lambda x : \text{Int}. x + 1) \ \langle \text{Int} \rightarrow \text{Int} \xrightarrow{l_1} ? \rightarrow ? \rangle (\text{true}\langle \text{Bool} \xrightarrow{l_2} ? \rangle) \\
\quad \leftarrow (\text{true}\langle \text{Bool} \xrightarrow{l_2} ? \rangle\langle ? \xrightarrow{l_1} \text{Int} \rangle + 1)\langle \text{Int} \xrightarrow{l_1} ? \rangle \\
\quad \leftarrow \text{blame } \bar{l}_1
\]

**Blame** tells us where an error occurred, and in which way the boundary was crossed.
Principle: to every use of the materialization rule corresponds a cast.
Principle: to every use of the materialization rule corresponds a cast.

\[
\Gamma \vdash e : \tau_1 \quad \tau_1 \preceq \tau_2 \\
\hline \\
\Gamma \vdash e : \tau_2
\]
Principle: to every use of the materialization rule corresponds a cast.

\[ \Gamma \vdash e : \tau_1 \rightarrow e' \quad \tau_1 \preceq \tau_2 \]

\[ \Gamma \vdash e : \tau_2 \rightarrow e' \langle \tau_1 \Rightarrow \tau_2 \rangle \]
Principle: to every use of the materialization rule corresponds a cast.

\[
\begin{align*}
\Gamma \vdash e : \tau_1 & \iff e' \quad \tau_1 \preceq \tau_2 \\
\Gamma \vdash e : \tau_2 & \iff e' \langle \tau_1 \Rightarrow \tau_2 \rangle
\end{align*}
\]

Casts of the form \( \langle \text{Int} \rightarrow ? \Rightarrow ? \rightarrow \text{Int} \rangle \) are forbidden.
Declarative Compilation

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

\[
\begin{align*}
\Gamma \vdash e : \tau_1 & \quad \rightarrow e' \quad \tau_1 \preceq \tau_2 \\
\Gamma \vdash e : \tau_2 & \quad \rightarrow e' \quad \langle \tau_1 \Rightarrow \tau_2 \rangle
\end{align*}
\]

Casts of the form \langle \text{Int} \rightarrow ? \Rightarrow ? \rightarrow \text{Int} \rangle are forbidden.

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

\[
\begin{align*}
\Gamma \vdash e : \tau_1 & \quad \left\{ \begin{array}{l}
p = l \\
p = \bar{l}
\end{array} \right. \quad \Rightarrow \tau_1 \preceq \tau_2 \\
\Gamma \vdash e\langle \tau_1 \Rightarrow \tau_2 \rangle : \tau_2
\end{align*}
\]
Type preservation for the declarative compilation is immediate.
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 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.
– 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.
– 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.
– 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
Part 1: Hindley-Milner

static types \( \mathcal{T}_t \ni t \ ::= \alpha \mid b \mid t \times t \mid t \rightarrow t \)

gradual types \( \mathcal{T}_\tau \ni \tau \ ::= \ ? \mid \alpha \mid b \mid \tau \times \tau \mid \tau \rightarrow \tau \)

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

cast language \( E \ ::= \lambda^{\tau \rightarrow \tau} x.\ E \mid \text{let } x = E \text{ in } E \mid \Lambda\vec{\alpha}.\ E \mid E[\vec{t}] \mid E(\tau \Rightarrow \tau) \mid \ldots \)
Inference: Main Ideas

– Based on the works of Pottier and Rémy [2005], and of Castagna et al. [2016].
– 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.
We first **generate constraints** of the form\(^1\):

\[
C ::= (t \leq t) \mid (\tau \gtrless \alpha) \mid (x \gtrless \alpha) \mid \text{def } x : \tau \text{ in } C \mid \exists \alpha. C \mid C \land C
\]

\(^1\)Let constraints are omitted for the sake of simplicity
We first generate constraints of the form\(^1\):

\[
C ::= (t \leq t) \mid (\tau \triangleleft \alpha) \mid (x \not\lesssim \alpha) \mid \text{def } x : \tau \text{ in } C \mid \exists \vec{\alpha}. \ C \mid C \land C
\]

\[
\llangle x : t \rrangle = \exists \alpha. (x \not\lesssim \alpha) \land (\alpha \leq t)
\]

\[
\llangle (\lambda x. e) : t \rrangle = \exists \alpha_1, \alpha_2. (\text{def } x : \alpha_1 \text{ in } \llangle e : \alpha_2 \rrangle) \land (\alpha_1 \not\lesssim \alpha_1) \land (\alpha_1 \rightarrow \alpha_2 \leq t)
\]

\[
\llangle (\lambda x : \tau. e) : t \rrangle = \exists \alpha_1, \alpha_2. (\text{def } x : \tau \text{ in } \llangle e : \alpha_2 \rrangle) \land (\tau \not\lesssim \alpha_1) \land (\alpha_1 \rightarrow \alpha_2 \leq t)
\]

\(^1\)Let constraints are omitted for the sake of simplicity
We first generate constraints of the form:\n\n\[ C ::= (t \leq t) \mid (\tau \preceq \alpha) \mid (x \preceq \alpha) \mid \text{def } x : \tau \text{ in } C \mid \exists \vec{\alpha}. \ C \mid C \land C \]

\[ \langle\langle x : t \rangle\rangle = \exists \alpha. (x \preceq \alpha) \land (\alpha \leq t) \]
\[ \langle\langle (\lambda x. e) : t \rangle\rangle = \exists \alpha_1, \alpha_2. (\text{def } x : \alpha_1 \text{ in } \langle\langle e : \alpha_2 \rangle\rangle) \land (\alpha_1 \preceq \alpha_1) \land (\alpha_1 \rightarrow \alpha_2 \leq t) \]
\[ \langle\langle (\lambda x : \tau. e) : t \rangle\rangle = \exists \alpha_1, \alpha_2. (\text{def } x : \tau \text{ in } \langle\langle e : \alpha_2 \rangle\rangle) \land (\tau \preceq \alpha_1) \land (\alpha_1 \rightarrow \alpha_2 \leq t) \]

Note that \( \langle\langle (\lambda x : \? . x) : \text{Int} \rightarrow \text{Int} \rangle\rangle \) can be solved, whereas \( \langle\langle (\lambda x . x) : \? \rightarrow \? \rangle\rangle \) cannot.

\(^1\)Let constraints are omitted for the sake of simplicity
We then **rewrite the structured constraints** to obtain a set containing **type constraints**:

\[ D ::= (t_1 \leq t_2) \mid (\tau \preceq \alpha) \]
We then rewrite the structured constraints to obtain a set containing type constraints:

\[ D ::= (t_1 \preceq t_2) \mid (\tau \preceq \alpha) \]

\[
\Gamma; \Delta \vdash (x \preceq \alpha) \rightsquigarrow \{ \tau\{\vec{\alpha} := \vec{\beta}\} \preceq \alpha \}
\]

\[
\Gamma(x) = \forall \vec{\alpha}. \tau
\]

\[ \vec{\beta} \text{ fresh} \]
We then rewrite the structured constraints to obtain a set containing type constraints:

\[
D ::= (t_1 \leq t_2) \mid (\tau \leq \alpha)
\]

\[
\Gamma; \Delta \vdash (x \leq \alpha) \leadsto \{\tau\{\bar{\alpha} := \bar{\beta}\} \leq \alpha\}
\]

\[\Gamma(x) = \forall \bar{\alpha}. \tau\]
\[\bar{\beta}\text{ fresh}\]

\[
(\Gamma, x : \tau); \Delta \vdash C \leadsto D
\]

\[
\Gamma; \Delta \vdash \text{def } x : \tau \text{ in } C \leadsto D
\]
Solving constraints

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

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

For example, the constraint

\[ ? \rightarrow ? \rightarrow ? \preceq \text{Bool} \rightarrow \alpha \]
Solving constraints

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

For example, the constraint

\[ ? \rightarrow ? \rightarrow ? \preceq \text{Bool} \rightarrow \alpha \]

will become

\[ X_1 \rightarrow X_2 \rightarrow X_3 \preceq \text{Bool} \rightarrow \alpha \]
Solving constraints

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

For example, the constraint

\[ ? \to ? \to ? \preceq \text{Bool} \to \alpha \]

will become

\[ X_1 \to X_2 \to X_3 \preceq \text{Bool} \to \alpha \]

and solving it will return the unifier

\[ \theta : X_1 \leftrightarrow \text{Bool}; X_2 \leftrightarrow \beta; X_3 \leftrightarrow \gamma; \alpha \leftrightarrow (\beta \to \gamma) \]
To summarize, given an expression $e$, and a constraint derivation $D$ of $\Gamma; \Delta \vdash \langle \langle e : t \rangle \rangle \leadsto D$, we can **compute a unifier** $\theta$ satisfying $D$. This derivation and the associated unifier can be used to compile $e$ in a straightforward way: to every materialization constraint introduced in $D$ corresponds a cast.

Inference (and compilation) for this system is **sound**, **type-preserving** and **complete** w.r.t. the declarative system.
To summarize, given an expression $e$, and a constraint derivation $\mathcal{D}$ of $\Gamma; \Delta \vdash \langle \langle e: t \rangle \rangle \rightsquigarrow D$, we can **compute a unifier** $\theta$ 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.**
To summarize, given an expression $e$, and a constraint derivation $D$ of $\Gamma; \Delta \vdash \langle\langle e : t \rangle\rangle \leadsto D$, we can compute a unifier $\theta$ satisfying $D$.

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

$$\langle x \rangle^D_\theta = x\langle \tau \theta \Rightarrow \alpha \theta \rangle \quad \text{if} \quad D = \Gamma; \Delta \vdash \langle\langle x : t \rangle\rangle \leadsto \{(\tau \preceq \alpha), (\alpha \preceq t)\}$$
Compilation and Results

To summarize, given an expression $e$, and a constraint derivation $\mathcal{D}$ of $\Gamma; \Delta \vdash \langle\langle e : t \rangle\rangle \leadsto D$, we can compute a unifier $\theta$ 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.

\[
\langle x \rangle^D_\theta = x \langle \tau \theta \Rightarrow \alpha \theta \rangle \quad \text{if} \quad \mathcal{D} = \Gamma; \Delta \vdash \langle\langle x : t \rangle\rangle \leadsto \{(\tau \preceq \alpha), (\alpha \preceq t)\}
\]

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

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

For example,

\[
\text{fun } x \rightarrow \text{if } (\text{fst } x) \text{ then } (1 + \text{snd } x) \text{ else } x
\]

should be of type \((\text{Bool} \times \text{Int}) \rightarrow (\text{Int} \mid (\text{Bool} \times \text{Int}))\)
The types become:

static types $t ::= \alpha \mid b \mid t \times t \mid t \to t \mid t \lor t \mid \neg t \mid \emptyset$

gradual types $\tau ::= ? \mid \alpha \mid b \mid \tau \times \tau \mid \tau \to \tau \mid \tau \lor \tau \mid \neg \tau \mid \emptyset$
The types become:

static types \[ t ::= \alpha \mid b \mid t \times t \mid t \rightarrow t \mid t \lor t \mid \neg t \mid \emptyset \]

gradual types \[ \tau ::= ? \mid \alpha \mid b \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \tau \lor \tau \mid \neg \tau \mid \emptyset \]

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).

\[ \{ (\alpha \preceq t_1), (\alpha \preceq t_2) \} \simeq \{ (\alpha \preceq t_1 \land t_2) \} \]
Part 3: Adding Set-Theoretic Types

The types become:

**static types** \( t ::= \alpha \mid b \mid t \times t \mid t \rightarrow t \mid t \vee t \mid \neg t \mid \emptyset \)

**gradual types** \( \tau ::= ? \mid \alpha \mid b \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \tau \vee \tau \mid \neg \tau \mid \emptyset \)

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).

\[ \{(\alpha \preceq t_1), (\alpha \preceq t_2)\} \simeq \{(\alpha \preceq t_1 \land t_2)\} \]

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

\((? \rightarrow ?) \lor ? \iff (X_0 \rightarrow X_1) \lor X_1\)
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.

\[
(\ ? \rightarrow \ ? \ ) \lor \ ? \quad \mapsto \quad (X_0 \rightarrow X_1) \lor X_1
\]

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

[ExpandL] \[ V⟨τ \Rightarrow ?⟩ \leftrightarrow V⟨τ \Rightarrow gnd(τ)⟩⟨gnd(τ) \Rightarrow ?⟩ \]

[Collapse] \[ V⟨ρ \Rightarrow ?⟩⟨? \Rightarrow q ρ'⟩ \leftrightarrow V \quad \text{if } ρ = ρ' \]

[Blame] \[ V⟨ρ \Rightarrow ?⟩⟨? \Rightarrow q ρ'⟩ \leftrightarrow \text{blame } q \quad \text{if } ρ \neq ρ' \]

\[
gnd(τ_1 \rightarrow τ_2) = ? \rightarrow ? \quad gnd(τ_1 \times τ_2) = ? \times ? \quad gnd(b) = b
\]
The semantics of the cast calculus for **HM without subtyping** are basically the same as those presented by Siek et al. [2015].

**[ExpandL]**
\[ V\langle \tau \xrightarrow{R} ? \rangle \leftrightarrow V\langle \tau \xrightarrow{R} \text{gnd}(\tau) \rangle \langle \text{gnd}(\tau) \xrightarrow{R} ? \rangle \]

**[Collapse]**
\[ V\langle \rho \xrightarrow{R} ? \rangle \langle ? \xrightarrow{a} \rho' \rangle \leftrightarrow V \quad \text{if } \rho = \rho' \]

**[Blame]**
\[ V\langle \rho \xrightarrow{R} ? \rangle \langle ? \xrightarrow{a} \rho' \rangle \leftrightarrow \text{blame } q \quad \text{if } \rho \neq \rho' \]

\[ \text{gnd}(\tau_1 \rightarrow \tau_2) = ? \rightarrow ? \quad \text{gnd}(\tau_1 \times \tau_2) = ? \times ? \quad \text{gnd}(b) = b \]

**Adding subtyping** is just a matter of allowing \( \rho \leq \rho' \).
Adding set-theoretic types is more complicated, mostly because we need to take into account unions and intersections containing ?.
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.

$$(\text{Int} \to \text{Int}) \lor (\text{Bool} \to \text{Bool})/(\text{Int} \to \text{Int}) \lor ? = (\text{Int} \to \text{Int}) \lor (? \to ?)$$
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.

\[
(\text{Int} \to \text{Int}) \lor (\text{Bool} \to \text{Bool})/(\text{Int} \to \text{Int}) \lor ? = (\text{Int} \to \text{Int}) \lor (? \to ?)
\]

[ExpandL] \[
V\langle \tau_1 \Rightarrow \tau_2 \rangle \leftrightarrow V\langle \tau_1 \Rightarrow \tau_1/\tau_2 \rangle \langle \tau_1/\tau_2 \Rightarrow \tau_2 \rangle
\]
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.

\[
\text{(Int } \rightarrow \text{Int) } \lor \text{(Bool } \rightarrow \text{Bool)} / \text{(Int } \rightarrow \text{Int) } \lor \ ? = \text{(Int } \rightarrow \text{Int) } \lor \ ? \rightarrow ?
\]

\[
[\text{ExpandL}] \quad V(\tau_1 \xrightarrow{P} \tau_2) \leftrightarrow V(\tau_1 \xrightarrow{P} \tau_1 / \tau_2) \langle \tau_1 / \tau_2 \xrightarrow{P} \tau_2 \rangle
\]

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.
– 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 aforementioned inference algorithm **reuses existing unification algorithms**.
– 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.
Thanks for listening!

Comments, questions, suggestions?