A Semantic Foundation for Gradual Set-theoretic Types

GPL Award Accessit

Victor Lanvin

June 7th, 2022
Gradual Typing

– Goal: have both static and dynamic typing in the same language.

– How: by adding a dynamic type, denoted “?”. 

The transition is gradual: 

\[ \text{Int} \rightarrow \text{?} \rightarrow \text{Int} \rightarrow \text{Bool} \]
Gradual Typing

- 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

– 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:

\[ ? \preceq ? \rightarrow ? \preceq \text{Int} \rightarrow ? \preceq \text{Int} \rightarrow \text{Bool} \]
Set-Theoretic Types

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

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

$(x = e_{\in \text{Int}}) ? \text{true} : x : \text{Bool} \lor \neg \text{Int}$

- Powerful but often syntactically heavy.
- In **Semantic subtyping**: 

  $\text{Types} \simeq \text{Sets of values}$
  
  $\text{Subtyping} \simeq \text{Set-containment}$
Set-Theoretic Types

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

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

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

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

  if $x$ then 3 else true : Int $\lor$ Bool
- **Types** with connectives ($\lor$, $\land$, $\neg$).

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

if $x$ then 3 else true : Int $\lor$ Bool

$(x = e \in \text{Int})? \text{true}:x : \text{Bool} \lor \neg \text{Int}$
Set-Theoretic Types

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

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

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

  \[(x = e \in \text{Int})? \text{true: x : Bool } \lor \neg \text{ Int}\]

- Powerful but often syntactically heavy.
Set-Theoretic Types

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

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

  if $x$ then 3 else true : Int $\lor$ Bool

  $(x = e \in \text{Int})? \text{true}: x : \text{Bool} \lor \neg \text{Int}$

- Powerful but often syntactically heavy.

- In **Semantic subtyping**:

  Types $\simeq$ Sets of values
  Subtyping $\simeq$ Set-containment
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. This is however very unsafe, as it accepts a string for example.
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 : ) : =
  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:

``` OCaml
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:

```ocaml
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:

```haskell
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.
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 ⟨α list⟩
  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:

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

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:

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

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

This is however very **unsafe**, as it accepts a **string** for example.
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 \lor α array) \land ?) =
  if condition then
    List.map f data
  else
    Array.map f data

- By subtyping, (α list \lor α array) \land ? \leq ?.
Motivating Example (2/2)

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

- By **subtyping**, $(\alpha \text{ list } \lor \alpha \text{ array}) \land ? \leq ?$.
- Can only be used with **lists or arrays**.
- No need for **manual type checks**.
Motivating Example (2/2)

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

- By subtyping, \((α \text{ list } \lor α \text{ array}) \land ? \leq ?\).
- 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 : Bool) f (data : (α list ∨ α array) ∧ ?) =
  if condition then
    List.map f data
  else
    Array.map f data

- By subtyping, $(\alpha \text{ list } \lor \alpha \text{ array}) \land ? \leq ?$.
- 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).
Motivating Example (2/2)

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 ∨ α array) ∧ ? ≤ ?. 
- 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 list ∨ α array) ∧ ? ≤ ?.  
- 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 subtype-consistency relation \( \leq \).
1. Define a \textit{subtype-consistency} relation $\tilde{\leq}$. 

This relation is not transitive! $? \tilde{\leq} \tau \tilde{\leq} ?$ for all $\tau$
How it is Usually Done

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

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

2. Embed this relation into typing rules.

   $\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_1'$ \hspace{1cm} $\Gamma \vdash e_2 : \tau_2$ \hspace{1cm} $\tau_2 \sim \leq \tau_1$

   \[ \Gamma \vdash e_1 \ e_2 : \tau_1' \]
How it is 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.

   \[
   \frac{
   \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \lesssim \operatorname{dom}(\tau_1)
   }{
   \Gamma \vdash e_1 \ e_2 : \tau_1 \circ \tau_2
   }
   \]

   This gets even more complicated with set-theoretic types!
1. Translate gradual types to **static types** (types without ?) **with variables**.
Our (First) Approach

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

2. Define **transitive** relations on gradual types, and in particular "**precision**" which contains the **essence of gradual typing**.
Our (First) Approach

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

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

3. Embed precision into more and more complex systems (Hindley-Milner, with subtyping, and with semantic subtyping).
Our (First) Approach

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

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

3. Embed precision 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**.
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*:

\[ ? \leq ? \lor \text{Int} \quad \text{Int} \land ? \leq ? \]
**Precision** is what allows us to **cross the barrier** from the dynamic world into the static world (*and only this way*).
**Precision** is what allows us to **cross the barrier** from the dynamic world into the static world (and only this way).

\[ ? \preceq \tau \text{ for every } \tau \]

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

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

And it is transitive:

\[ ? \preceq ? \rightarrow ? \preceq ? \rightarrow \text{Int} \preceq \text{Int} \rightarrow \text{Int} \]
**Precision** 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-like** rule: **materialization**.
Declarative Type Systems

\[
\begin{align*}
\Gamma, x : \tau & \vdash x : \tau \\
\Gamma & \vdash \lambda x . e : \tau_1 \rightarrow \tau_2 \\
\Gamma & \vdash e_1 : \tau_1 \rightarrow \tau_2 \\
\Gamma & \vdash e_2 : \tau_1 \\
\Gamma & \vdash e_1 \; e_2 : \tau_2
\end{align*}
\]
Declarative Type Systems

\[
\begin{align*}
\begin{array}{c}
\Gamma, x : \tau \vdash x : \tau \\
\hline
\Gamma, x : \tau \vdash x : \tau \\
\end{array} \\
\begin{array}{c}
\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \\
\hline
\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \\
\hline
\Gamma \vdash e_2 : \tau_1 \\
\hline
\Gamma \vdash e_1 \ e_2 : \tau_2 \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\Gamma \vdash e : \tau_1 \\
\hline
\tau_1 \preceq \tau_2 \\
\hline
\Gamma \vdash e : \tau_2 \\
\end{array} \\
\begin{array}{c}
\Gamma \vdash e : \tau_1 \\
\hline
\tau_1 \leq \tau_2 \\
\hline
\Gamma \vdash e : \tau_2 \\
\end{array}
\end{align*}
\]
Declarative Type Systems

\[
\frac{\Gamma, x : \forall \vec{\alpha}. \tau \vdash x : \tau \{\vec{\alpha} := \vec{t}\}}{\Gamma, x : \tau_1 \vdash e : \tau_2} \quad \frac{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2}{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \text{Gen}_\Gamma(\tau_1) \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}
\]
Declarative Type Systems

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

\[ \Gamma \vdash e : \tau_1 \]

\[ \tau_1 \preceq \tau_2 \quad \tau_1 \leq \tau_2 \]

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

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

\[
\begin{align*}
&\Gamma, x : \forall \alpha. \tau \vdash x : \tau \{\alpha := \bar{t}\} \\
&\Gamma, x : \tau_1 \vdash e : \tau_2 \\
&\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2
\end{align*}
\]

\[
\begin{align*}
&\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1 \\
&\Gamma \vdash e_1 \; e_2 : \tau_2
\end{align*}
\]

\[
\begin{align*}
&\Gamma \vdash e_1 : \tau_1 \\
&\Gamma, x : \text{Gen} \Gamma(\tau_1) \vdash e_2 : \tau \\
&\Gamma \vdash \text{let} \; x = e_1 \; \text{in} \; e_2 : \tau
\end{align*}
\]

\[
\begin{align*}
&\Gamma \vdash e : \tau_1 \quad \tau_1 \preceq \tau_2 \\
&\Gamma \vdash e : \tau_2
\end{align*}
\]

\[
\begin{align*}
&\Gamma \vdash e : \tau_1 \quad \tau_1 \leq \tau_2 \\
&\Gamma \vdash e : \tau_2
\end{align*}
\]

And as a bonus, we get the **static gradual guarantee** for free!
Theorem
For every type $\tau \in \text{GTypes}$, there exists $t_1, t_2 \in \text{STypes}$ such that:

$\tau \preceq t_1$ and $\tau \preceq t_2$

$\forall \tau' \in \text{GTypes}. \, \tau \preceq \tau' \implies t_1 \leq \tau' \leq t_2$
### Theorem

For every type $\tau \in \text{GTypes}$, there exists $t_1, t_2 \in \text{STypes}$ such that:

$$\tau \preceq t_1 \text{ and } \tau \preceq t_2$$

$$\forall \tau' \in \text{GTypes}. \tau \preceq \tau' \implies t_1 \leq \tau' \leq t_2$$

We write $t_1 = \tau\downarrow$ and $t_2 = \tau\uparrow$. 
Theorem

For every type $\tau \in \text{GTypes}$, there exists $t_1, t_2 \in \text{STypes}$ such that:

$\tau \prec t_1$ and $\tau \prec t_2$

$\forall \tau' \in \text{GTypes}. \; \tau \prec \tau' \implies t_1 \leq \tau' \leq t_2$

We write $t_1 = \tau \downarrow$ and $t_2 = \tau \uparrow$.

$ (? \rightarrow ?) \uparrow = 0 \rightarrow 1 \quad ( ? \rightarrow ?) \downarrow = 1 \rightarrow 0$
Theorem

For every type $\tau \in \text{GTypes}$, there exists $t_1, t_2 \in \text{STypes}$ such that:

$\tau \preceq t_1$ and $\tau \preceq t_2$

$\forall \tau' \in \text{GTypes}. \; \tau \preceq \tau' \implies t_1 \leq \tau' \leq t_2$

We write $t_1 = \tau \downarrow$ and $t_2 = \tau \uparrow$.

$(\_ \rightarrow \_) \uparrow = 0 \rightarrow 1 \quad (\_ \rightarrow \_) \downarrow = 1 \rightarrow 0$

These types are computed in \textit{linear time}!
We show the following:

$$\tau_1 \leq \tau_2 \iff \begin{cases} \tau_1 \downarrow \leq \tau_2 \downarrow \\ \tau_1 \uparrow \leq \tau_2 \uparrow \end{cases}$$

Moreover, we have that for every gradual type $\tau$:

$$\tau \simeq \tau \downarrow \lor (\bot \land \tau \uparrow)$$

We can use this representation to lift operators to gradual types!
An Equivalent Representation of Gradual Types

We show the following:

\[ \tau_1 \leq \tau_2 \iff \begin{cases} \tau_1 \downarrow \leq \tau_2 \downarrow \\ \tau_1 \uparrow \leq \tau_2 \uparrow \end{cases} \]

\[ \tau_1 \not\leq \tau_2 \iff \begin{cases} \tau_1 \downarrow \leq \tau_2 \downarrow \\ \tau_2 \uparrow \leq \tau_1 \uparrow \end{cases} \]

Moreover, we have that for every gradual type \( \tau \),

\[ \tau \simeq \tau \downarrow \lor (\bot \land \tau \uparrow) \]

We can use this representation to lift operators to gradual types!
An Equivalent Representation of Gradual Types

We show the following:

\[ \tau_1 \leq \tau_2 \iff \begin{cases} \tau_1 \Downarrow \leq \tau_2 \Downarrow \\ \tau_1 \Uparrow \leq \tau_2 \Uparrow \end{cases} \quad \tau_1 \preceq \tau_2 \iff \begin{cases} \tau_1 \Downarrow \leq \tau_2 \Downarrow \\ \tau_2 \Uparrow \leq \tau_1 \Uparrow \end{cases} \]

Moreover, we have that for every gradual type \( \tau \),

\[ \tau \simeq \tau \Downarrow \lor (\top \land \tau \Uparrow) \]
An Equivalent Representation of Gradual Types

We show the following:

$$\tau_1 \leq \tau_2 \iff \begin{cases} \tau_1 \downarrow \leq \tau_2 \downarrow \\ \tau_1 \uparrow \leq \tau_2 \uparrow \end{cases} \quad \tau_1 \preceq \tau_2 \iff \begin{cases} \tau_1 \downarrow \leq \tau_2 \downarrow \\ \tau_2 \uparrow \leq \tau_1 \uparrow \end{cases}$$

Moreover, we have that for every gradual type $\tau$,

$$\tau \simeq \tau \downarrow \lor (\mathbf{?} \land \tau \uparrow)$$

We can use this representation to lift operators to gradual types!

$$\text{dom}(\tau) \overset{\text{def}}{=} \text{dom}(\tau \uparrow) \lor (\mathbf{?} \land \text{dom}(\tau \downarrow))$$
Conclusion

Your favorite typing rules + Materialization + Subsumption = Your gradual type system
Conclusion

Your favorite typing rules + Materialization + Subsumption = Your gradual type system

We present:

1. A simple method of declaratively adding gradual typing to any existing type system.
Conclusion

Your favorite typing rules + Materialization + Subsumption = Your gradual type system

We present:

1. A simple method of declaratively adding gradual typing to any existing type system.

2. A set-theoretic interpretation of gradual types that has considerable consequences.
Conclusion

*Your favorite typing rules + Materialization + Subsumption = Your gradual type system*

We present:

1. A simple method of **declaratively adding** gradual typing to any existing type system.
2. A **set-theoretic interpretation** of gradual types that has considerable consequences.
3. The **algorithmic systems** for our GTLC with set-theoretic types.
Conclusion

*Your favorite typing rules + Materialization + Subsumption = Your gradual type system*

We present:

1. A simple method of *declaratively adding* gradual typing to any existing type system.
2. A *set-theoretic interpretation* of gradual types that has considerable consequences.
3. The *algorithmic systems* for our GTLC with set-theoretic types.
4. *Denotational semantics* for several calculi, including CDuce, and a GTLC without set-theoretic types.
Future work

- Fully unify our logical approach and our denotational semantics.
Future work

- Fully unify our logical approach and our denotational semantics.

- Sound and complete type inference for gradual set-theoretic types.
Future work

- Fully **unify** our logical approach and our denotational semantics.

- Sound and complete **type inference** for gradual set-theoretic types.

- Add **more features** to our calculus, such as intersection types for functions.
Future work

- Fully **unify** our logical approach and our denotational semantics.

- Sound and complete **type inference** for gradual set-theoretic types.

- Add **more features** to our calculus, such as intersection types for functions.

- A **denotational semantics** for a cast calculus with set-theoretic types.