Gradual Typing: A New Perspective

[Was: Gradual Typing + Set-Theoretic Types + Let-Polymorphism]

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

POPL 2019
Cascais/Lisbon, Portugal, 13–19 January 2019
Gradual Typing

- Embed both **static** and **dynamic** typing in the same language.

- Adds a **dynamic type**, denoted “?”. 
Gradual Typing

- Embed both static and dynamic typing in the same language.
- Adds a dynamic type, denoted “?”.
- Allows for a trade-off between safety and programming productivity.
Gradual Typing

- Embed both static and dynamic typing in the same language.
- Adds a dynamic type, denoted “?”. 
- Allows for a trade-off between safety and programming productivity.

? = arbitrary value
Gradual Typing

- Embed both static and dynamic typing in the same language.

- Adds a dynamic type, denoted “?”.

- Allows for a trade-off between safety and programming productivity.

\[ ? = \text{arbitrary value} \]
\[ (? \rightarrow ?) = \text{arbitrary function} \]
Set-Theoretic Types

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

Useful for overloading, branching, but often syntactically heavy.

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

In Semantic subtyping, Types $\simeq$ Sets of values
Subtyping $\simeq$ Set-containment
– **Types with** *connectives* \((\lor, \land, \neg)\)

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

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

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

\[(\text{Int} \to \text{Int}) \land (\text{Bool} \to \text{Bool}) = \text{overloaded function}\]
- **Types** with **connectives** ($\lor$, $\land$, $\neg$)

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

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

  if `x` then 3 else true : Int $\lor$ Bool
Set-Theoretic Types

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

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

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

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

- In **Semantic subtyping**,
  
  Types $\simeq$ Sets of values
  
  Subtyping $\simeq$ Set-containment
Motivating Example (1/2)

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.
Motivating Example (1/2)

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
```
Motivating Example (1/2)

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
```
Motivating Example (1/2)

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:

```plaintext
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 map condition f
data : (α list \lor α array) ) =
if condition then
    List.map f data
else
    Array.map f data
Motivating Example (2/2)

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

- Can only be used with lists or arrays
- No need for manual type checks
- All non-gradual types are inferred, and the return type is not gradual anymore
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

– Can only be used with lists or arrays
– No need for manual type checks
let map (condition : Bool) f (data : (α list ∨ α array) ∧ ?) =
  if condition then
    List.map f data
  else
    Array.map f data

– Can only be used with lists or arrays
– No need for manual type checks
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

– Can only be used with lists or arrays
– No need for manual type checks
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

– Can only be used with lists or arrays
– No need for manual type checks
let map condition f
(data : (\alpha list \lor \alpha array) \land ?) =
if condition then
  List.map f data
else
  Array.map f data

- Can only be used with lists or arrays
- No need for manual type checks
- All non-gradual types are inferred, and the return type is not gradual anymore
How it is Usually Done

1. Define a **subtype-consistency** relation \( \lesssim \).

This gets even more complicated with set-theoretic types!
1. Define a **subtype-consistency** relation $\lesssim$.

This relation is not transitive! $? \lesssim \tau \lesssim ?$ for all $\tau$. 
1. Define a **subtype-consistency** relation $\sim \leq$.

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

2. Embed this relation into typing rules.

\[
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau'_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\frac{\tau_2 \sim \leq \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau'_1}
\]
1. Define a **subtype-consistency** relation \( \sim \).

This relation is not transitive! \( ? \sim \tau \sim ? \) for all \( \tau \)

2. Embed this relation into typing rules.

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \sim \text{dom}(\tau_1)
\]

\[
\Gamma \vdash e_1 \ e_2 : \tau_1 \circ \tau_2
\]

This gets even more complicated with set-theoretic types!
Main idea: interpret occurrences of \(?\) as arbitrary type variables.
Main idea: interpret occurrences of \(?\) as arbitrary type variables.

1. Translate gradual types to static types (types without \(?\)) with variables.
Main idea: interpret occurrences of ? as arbitrary type variables.

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

2. Define a transitive subtyping relation on gradual types.
Main idea: interpret occurrences of \(?\) as arbitrary type variables.

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

2. Define a transitive subtyping relation on gradual types.

3. Define a transitive “materialization” relation to add gradual typing.
Our Approach

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

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

2. Define a transitive subtyping relation on gradual types.

3. Define a transitive “materialization” relation to add gradual typing.

Important: this idea is only used to define relations on gradual types!
We first define the **discrimination** of a gradual type:

\[ D(\_ \_ \_ \_ \_ \_ \_) = \{ X_1; X_2; \ldots \} \]
Discrimination and Subtyping

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

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

\[ \mathcal{D}((\text{Int} \to ?) \land ?) = \{ (\text{Int} \to X_1) \land X_1; (\text{Int} \to X_1) \land X_2; \ldots \} \]
Discrimination and Subtyping

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

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

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

Subtyping on **gradual types** is then defined using subtyping on **static types**:

\[
\tau_1 \leq \tau_2 \iff \exists (T_1, T_2) \in \mathcal{D}(\tau_1) \times \mathcal{D}(\tau_2), \ T_1 \leq_T T_2
\]
Discrimination and Subtyping

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

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

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

Subtyping on **gradual types** is then defined using subtyping on **static types**:

\[ \tau_1 \leq \tau_2 \iff \exists (T_1, T_2) \in D(\tau_1) \times D(\tau_2), \ T_1 \leq_T T_2 \]

\[ ? \to \text{Nat} \leq ? \to \text{Int} \text{ since } X \to \text{Nat} \leq_T X \to \text{Int} \]
Subtyping only allows us to “move” inside the dynamic or static world.
**Subtyping** only allows us to “move” **inside** the dynamic or static world.

**Materialization** is what allows to **crossing the barrier** from the dynamic world into the static world.
Subtyping only allows us to “move” inside the dynamic or static world.

Materialization is what allows to crossing the barrier from the dynamic world into the static world.

\[ \tau_1 \preceq \tau_2 \overset{\text{def}}{\iff} \exists T_1 \in D(\tau_1), \sigma : \text{Vars} \to \text{GTypes}, T_1 \sigma = \tau_2 \]

? \preceq \tau \quad \text{for every } \tau

? \rightarrow ? \preceq \tau_1 \rightarrow \tau_2 \quad \text{for every } \tau_1, \tau_2
Subtyping only allows us to “move” inside the dynamic or static world.

Materialization is what allows to crossing the barrier from the dynamic world into the static world.

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

? \preceq \tau \quad \text{for every } \tau

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

Note: it is the inverse of precision (Garcia [2013]).
The two previously defined relations are **transitive**.
The two previously defined relations are transitive.

They can be embedded into a type system as subsumption-like rules.
Declarative Type System

The two previously defined relations are **transitive**.

They can be embedded into a type system as **subsumption-like rules**.

\[
\begin{align*}
\Gamma, x : \tau &\vdash x : \tau \\
\Gamma, x : \tau_1 \vdash e : \tau_2 &\quad \Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 &\quad \Gamma \vdash e_2 : \tau_1 \\
\hline
\Gamma \vdash e_1 \ e_2 : \tau_2
\end{align*}
\]
The two previously defined relations are transitive.

They can be embedded into a type system as subsumption-like rules.

\[
\begin{align*}
\Gamma, x : \tau & \vdash x : \tau \\
\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 & \quad \Gamma \vdash e_2 : \tau_1 \\
& \quad \Gamma \vdash e_1 \ e_2 : \tau_2 \\
\Gamma \vdash e : \tau_1 & \quad \tau_1 \preceq \tau_2 \\
& \quad \Gamma \vdash e : \tau_2 \\
\Gamma \vdash e : \tau_1 & \quad \tau_1 \preceq \tau_2 \\
& \quad \Gamma \vdash e : \tau_2
\end{align*}
\]
The two previously defined relations are transitive. They can be embedded into a type system as subsumption-like rules.

\[
\Gamma, x : \forall \vec{\alpha}. \tau \vdash x : \tau \{ \vec{\alpha} : = \vec{t} \} \quad \Gamma, x : \tau_1 \vdash e : \tau_2
\]

\[
\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \quad \Gamma, \ x:Gen_{\Gamma}(\tau_1) \vdash e_2 : \tau
\]

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

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \text{Gen}_{\Gamma}(\tau_1) \vdash e_2 : \tau \quad \Gamma \vdash \text{let} \ x = e_1 \ \text{in} \ e_2 : \tau
\]
The two previously defined relations are **transitive**.

They can be embedded into a type system as **subsumption-like rules**.

\[
\begin{align*}
\Gamma, x : \forall \vec{\alpha}. \tau & \vdash x : \tau \{ \vec{\alpha} := \vec{t} \} \\
\Gamma, x : \tau_1 & \vdash e : \tau_2 \\
\Gamma & \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \\
\Gamma & \vdash e_1 : \tau_1 \rightarrow \tau_2 \\
& \quad \Gamma \vdash e_2 : \tau_1 \\
\Gamma & \vdash e_1 \ e_2 : \tau_2 \\
\Gamma & \vdash e_1 : \tau_1 \\
& \quad \Gamma, x : \text{Gen}_\Gamma(\tau_1) \vdash e_2 : \tau \\
\Gamma & \vdash \text{let } x = e_1 \text{ in } e_2 : \tau \\
\Gamma & \vdash e : \tau_1 \\
\tau_1 & \preceq \tau_2 \\
\Gamma & \vdash e : \tau_2
\end{align*}
\]
The two previously defined relations are **transitive**.

They can be embedded into a type system as **subsumption-like** rules.

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

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

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

\[
\begin{align*}
\Gamma & \vdash e : \tau_1 & \tau_1 \preceq \tau_2 & \Gamma & \vdash e : \tau_1 & \tau_1 \leq \tau_2 \\
\Gamma & \vdash e : \tau_2 & \Gamma & \vdash e : \tau_2
\end{align*}
\]
We have $\Gamma \vdash \text{data} : (\alpha \text{ array} \lor \alpha \text{ list}) \land ?$. 

Hence $\Gamma \vdash \text{data} : \alpha \text{ array} \Rightarrow \text{Array.map f data}$ is well-typed.
We have $\Gamma \vdash \text{data} : (\alpha \text{ array} \lor \alpha \text{ list}) \land ?$.

And the following materialization:

$$(\alpha \text{ array} \lor \alpha \text{ list}) \land ? \preceq (\alpha \text{ array} \lor \alpha \text{ list}) \land \alpha \text{ array}$$

$$\simeq \alpha \text{ array}$$
We have $\Gamma \vdash \text{data} : (\alpha \text{ array} \lor \alpha \text{ list}) \land ?$.

And the following materialization:

$$(\alpha \text{ array} \lor \alpha \text{ list}) \land ? \preceq (\alpha \text{ array} \lor \alpha \text{ list}) \land \alpha \text{ array}$$

$$\simeq \alpha \text{ array}$$

Hence $\Gamma \vdash \text{data} : \alpha \text{ array}$
We have $\Gamma \vdash \text{data} : (\alpha \text{ array} \lor \alpha \text{ list}) \land ?$.

And the following materialization:

$$(\alpha \text{ array} \lor \alpha \text{ list}) \land ? \preceq (\alpha \text{ array} \lor \alpha \text{ list}) \land \alpha \text{ array} \preceq \alpha \text{ array}$$

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

$\Rightarrow \text{Array.map f data is well-typed.}$
We need to introduce runtime type-checks or casts to ensure dynamic values are not misused.
Translation to a Cast Calculus

We need to introduce runtime type-checks or casts to ensure dynamic values are not misused.

Principle: to every use of the materialization rule corresponds a cast.
We need to introduce runtime type-checks or casts to ensure dynamic values are not misused.

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

\[
\Gamma \vdash e : \tau_1 \quad \tau_1 \preceq \tau_2 \\
\overline{\quad} \\
\Gamma \vdash e : \tau_2
\]
We need to introduce runtime type-checks or casts to ensure dynamic values are not misused.

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

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

\[
\begin{align*}
\Gamma \vdash e : \tau_2 & \qquad \mapsto e' \langle \tau_2 \rangle
\end{align*}
\]
We need to introduce **runtime type-checks** or **casts** to ensure dynamic values are not misused.

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

\[
\frac{\Gamma \vdash e : \tau_1 \mapsto e' \quad \tau_1 \preceq \tau_2}{\Gamma \vdash e : \tau_2 \mapsto e' \langle \tau_2 \rangle}
\]

Back to the example:

\[
\text{Array.map } f \text{ data } \mapsto \text{Array.map } f \text{ (data}\langle (\alpha \text{ array} \lor \alpha \text{ list}) \land \alpha \text{ array})\rangle
\]

\[
= \text{Array.map } f \text{ (data}\langle \alpha \text{ array} \rangle\rangle
\]
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.
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.
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].
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].
Conclusion

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

1. We defined a simple, declarative way to add gradual typing to existing type systems, using two subsumption rules, and by interpreting gradual types as static types with variables.
Your favorite typing rules + Materialization + Subtyping = Your gradual type system

1. We defined a simple, declarative way to add gradual typing to existing type systems, using two subsumption rules, and by interpreting gradual types as static types with variables.

2. We highlight a direct correspondence between compilation and type derivations.
Conclusion

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

1. We defined a simple, declarative way to add gradual typing to existing type systems, using two subsumption rules, and by interpreting gradual types as static types with variables.

2. We highlight a direct correspondence between compilation and type derivations.

3. We defined a language with polymorphism, gradual typing and set-theoretic types that enjoys a conservativity result, blame safety and a soundness property.
1. Study **other features**, such as dynamic type-cases, or overloaded function interfaces.
1. Study **other features**, such as dynamic type-cases, or overloaded function interfaces.

2. What is the **underlying logic** associated to expressions of the cast language?
1. Study **other features**, such as dynamic type-cases, or overloaded function interfaces.

2. What is the **underlying logic** associated to expressions of the cast language?

3. Can we **implement** it? What about **efficiency**?
1. A direct correspondance between the safety of a cast and the polarity of its blame label…

2. …which yields a simpler statement of blame safety, thanks to materialization.

3. The reformulation of the type inference problem for gradual types in terms of static types.

4. Algorithmic typing rules and compilation rules.

5. The full operational semantics of a cast calculus with gradual set-theoretic types, blame, and let-polymorphism.

6. And an open post-doc position at IRIF in Paris, France.