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.**

$$\texttt{?} = \text{arbitrary value}$$

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.**

`?` = arbitrary value
`(? -> ?)` = arbitrary function

Set-Theoretic Types

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

Set-Theoretic Types

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

Set-Theoretic Types

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

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

Set-Theoretic Types

- **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

- **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

Motivating Example (1/2)

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 :  ) :     =
```

Motivating Example (1/2)

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
```

Motivating Example (1/2)

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
```

Motivating Example (1/2)

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
```

Motivating Example (1/2)

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.

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
```

Motivating Example (2/2)

```
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⟩)
```

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
```

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
```

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

Motivating Example (2/2)

```
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

Motivating Example (2/2)

```
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

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
```

- 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 $\widetilde{\leq}$.

How it is Usually Done

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

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

How it is Usually Done

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

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

2. Embed this relation into typing rules.

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

How it is Usually Done

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

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

2. Embed this relation into typing rules.

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

This gets even more complicated with set-theoretic types!

Our Approach

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

Our Approach

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

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

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.

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.

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!

Discrimination and Subtyping

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

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

Discrimination and Subtyping

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

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

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

Discrimination and Subtyping

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

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

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

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

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

Discrimination and Subtyping

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

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

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

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

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

$$? \to \texttt{Nat} \leq ? \to \texttt{Int} \text{ since } X \to \texttt{Nat} \leq_T X \to \texttt{Int}$$

Materialization

Subtyping only allows us to "move" **inside** the dynamic or static world.

Materialization

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.

Materialization

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.

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

$? \preccurlyeq \tau \quad$ for every τ

$? \to ? \preccurlyeq \tau_1 \to \tau_2 \quad$ for every τ_1, τ_2

Materialization

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.

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

$? \preccurlyeq \tau$ for every τ

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

Note: it is the inverse of **precision** (Garcia [2013]).

Declarative Type System

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

Declarative Type System

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**.

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

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

Declarative Type System

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

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

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

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

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

Declarative Type System

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

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

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

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

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

Declarative Type System

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

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

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

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

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

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

Declarative Type System

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

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

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

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

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

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

Back to the Example

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

Back to the Example

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

And the following materialization:

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

Back to the Example

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

And the following materialization:

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

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

Back to the Example

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

And the following materialization:

$$
\begin{aligned}
(\alpha \ \texttt{array} \vee \alpha \ \texttt{list}) \wedge ? &\preccurlyeq (\alpha \ \texttt{array} \vee \alpha \ \texttt{list}) \wedge \alpha \ \texttt{array} \\
&\simeq \alpha \ \texttt{array}
\end{aligned}
$$

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

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

Translation to a Cast Calculus

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.

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.

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

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.

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

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.

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

Back to the example:

$$\begin{aligned}\texttt{Array.map f data} &\mapsto \texttt{Array.map f (data}\langle(\alpha\ \texttt{array} \vee \alpha\ \texttt{list}) \wedge \alpha\ \texttt{array}\rangle\texttt{)} \\ &= \texttt{Array.map f (data}\langle\alpha\ \texttt{array}\rangle\texttt{)}\end{aligned}$$

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**.

$$\begin{tikzcd}
\tau_1 \arrow[rr, leftrightarrow, "\sim"] \arrow[dr, "\preccurlyeq"'] & & \tau_2 \arrow[dl, "\preccurlyeq"] \\
& \tau &
\end{tikzcd}$$

Is This Still Gradual Typing?

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

$$\begin{tikzcd} \tau_1 \arrow[rr, leftrightarrow, "\sim"] \arrow[dr, "\preccurlyeq"'] & & \tau_2 \arrow[dl, "\preccurlyeq"] \\ & \tau & \end{tikzcd}$$

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**.

$$
\begin{tikzcd}
\tau_1 \arrow[rr, leftrightarrow, "\sim"] \arrow[dr, "\preccurlyeq"'] & & \tau_2 \arrow[dl, "\preccurlyeq"] \\
& \tau &
\end{tikzcd}
$$

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**.

$$\begin{tikzcd} \tau_1 \arrow[rr, leftrightarrow, "\sim"] \arrow[dr, "\preccurlyeq"'] & & \tau_2 \arrow[dl, "\preccurlyeq"] \\ & \tau & \end{tikzcd}$$

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

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.

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**.

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.

Perspectives

1. Study **other features**, such as dynamic type-cases, or overloaded function interfaces.

Perspectives

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?

Perspectives

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**?

And there is much more

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.