Polymorphic Gradual Typing

A Set-Theoretic Perspective

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

TYPES 2018
Braga, Portugal, 18–21 June 2018

Set-Theoretic Types

- **Types** are interpreted as **sets of values**.
- **Subtyping** is defined as **set-containment**.

Set-Theoretic Types

- **Types** are interpreted as **sets of values**.
- **Subtyping** is defined as **set-containment**.
- **Useful for overloading, branching, etc, but often syntactically heavy.**

Set-Theoretic Types

- **Types** are interpreted as **sets of values**.
- **Subtyping** is defined as **set-containment**.
- **Useful for overloading, branching, etc, but often syntactically heavy.**

`(Int -> Int) /\ (Bool -> Bool)` = overloaded function

Gradual Typing

- Makes the transition between **static** and **dynamic** typing.
- Adds a **dynamic type**, denoted "`?`".

Gradual Typing

- Makes the transition between **static** and **dynamic** typing.
- Adds a **dynamic type**, denoted "?".
- **Allows for a trade-off between safety and programming efficiency.**

Gradual Typing

- Makes the transition between **static** and **dynamic** typing.
- Adds a **dynamic type**, denoted "?".
- **Allows for a trade-off between safety and programming efficiency.**

? = arbitrary value

Gradual Typing

- Makes the transition between **static** and **dynamic** typing.
- Adds a **dynamic type**, denoted "`?`".
- **Allows for a trade-off between safety and programming efficiency.**

`?` = arbitrary value

`(? -> ?)` = arbitrary function

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
  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
- Non-gradual types are inferred

How it is Usually Done

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

How it is Usually Done

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

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

How it is Usually Done

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

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

2. Embed this relation into typing rules.

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

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

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

2. Embed this relation into typing rules.

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

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 with variables**.

Our Approach

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

1. Translate gradual types to **static types 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 with variables**.

2. Define a **transitive subtyping** relation on gradual types.

3. Define a **transitive "materialization"** relation to add gradual typing.

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:

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

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

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

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

Materialization

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

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.

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

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.

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

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

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

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**.

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

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

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

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

And the following materialization:

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

Back to the Example

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

And the following materialization:

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

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

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 materialization 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 materialization 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 materialization 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 materialization 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:

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

Conclusion

1. We defined a **simple, declarative way** to add gradual typing to existing type systems.

Conclusion

1. We defined a **simple, declarative way** to add gradual typing to existing type systems.

2. We also defined **algorithmic typing rules and compilation rules**.

Conclusion

1. We defined a **simple, declarative way** to add gradual typing to existing type systems.
2. We also defined **algorithmic typing rules and compilation rules**.
3. Most concepts are based or **efficiently reduce** to existing work on static types.

Perspectives

1. More results: gradual guarantee, blame safety, . . .

Perspectives

1. More results: gradual guarantee, blame safety, . . .

2. Study the underlying logic associated to expressions of the cast language.

Perspectives

1. More results: gradual guarantee, blame safety, . . .

2. Study the underlying logic associated to expressions of the cast language.

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