Polymorphic Gradual Typing
A Set-Theoretic Perspective

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

TYPES 2018
Braga, Portugal, 18–21 June 2018
– **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.

\[(\text{Int} \rightarrow \text{Int}) \land (\text{Bool} \rightarrow \text{Bool}) = \text{overloaded function}\]
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
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 : ) : =
```

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 : \alpha \rightarrow \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:

```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
```
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
```

Runtime checks or casts are then inserted automatically by the compiler.
let map condition f
  (data : (\alpha list \or \alpha 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

– 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)

```haskell
let map condition (f : \alpha \rightarrow \beta)
    (data : (\alpha list \lor \alpha array) \lor ?) =
  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
Motivating Example (2/2)

```haskell
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
1. Define a **subtype-consistency** relation $\sim \leq$.
1. Define a **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 \( \tilde{\leq} \).

   This relation is not transitive! \( ? \tilde{\leq} \tau \tilde{\leq} ? \) for all \( \tau \)

2. Embed this relation into typing rules.

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

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

2. Embed this relation into typing rules.

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

$$
\Gamma \vdash e_1 \; e_2 : \tau_1 \circ \tau_2
$$

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

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

1. Translate gradual types to static types with variables.
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.
We first define the **discrimination** of a gradual type:

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

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

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

\[
? \rightarrow \text{Nat} \leq ? \rightarrow \text{Int} \text{ since } X \rightarrow \text{Nat} \leq_T X \rightarrow \text{Int}
\]
**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.
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.

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

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

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

? \rightarrow ? \preceq \tau_1 \rightarrow \tau_2 \quad \text{for every } \tau_1, \tau_2
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.
The two previously defined relations are transitive.

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

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

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

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

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

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

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

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

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

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

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

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

Back to the example:

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

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

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

\[
\Gamma \vdash e : \tau_1 \mapsto e' \quad \tau_1 \preccurlyeq \tau_2 \\
\Gamma \vdash e : \tau_2 \mapsto e' \langle \tau_2 \rangle
\]
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.

\[
\Gamma \vdash e : \tau_1 \leftrightarrow e' \quad \tau_1 \preceq \tau_2 \\
\Gamma \vdash e : \tau_2 \leftrightarrow e' \langle \tau_2 \rangle
\]

Back to the example:

\[
\text{Array.map } f \text{ data} \mapsto \text{Array.map } f \left(\text{data}\langle(\alpha \text{ array} \lor \alpha \text{ list}) \land \alpha \text{ array}\rangle\right) \\
= \text{Array.map } f \left(\text{data}\langle\alpha \text{ array}\rangle\right)
\]
1. We defined a **simple, declarative way** to add gradual typing to existing type systems.
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.
1. More results: gradual guarantee, blame safety, ...
1. More results: gradual guarantee, blame safety, . . .

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