Gradual Typing with Union and Intersection Types

Giuseppe Castagna, **Victor Lanvin**

ICFP '17

September 6, 2017

Outline

Motivating Example (1/3)

[Siek and Vachharajani, 2008]

```
let succ : Int -> Int = ...
let not : Bool -> Bool = ...

let f (condition : Bool) (x :  ) :   =
  if condition then
    succ x
  else
    not x
```

Motivating Example (1/3)

[Siek and Vachharajani, 2008]

```
let succ : Int -> Int = ...
let not : Bool -> Bool = ...

let f (condition : Bool) (x : ?) : ? =
  if condition then
    succ x
  else
    not x
```

→ Cannot be typed with simple types, but valid with gradual types.

Motivating Example (1/3)

[Siek and Vachharajani, 2008]

```
let succ : Int -> Int = ...
let not : Bool -> Bool = ...

let f (condition : Bool) (x : ?) : ? =
  if condition then
    succ x
  else
    not x
```

→ Cannot be typed with simple types, but valid with gradual types.
→ What if we apply it to a string?

Motivating Example (2/3)

Set-theoretic version:

```
let f (condition : Bool) (x : (Int | Bool))
      : (Int | Bool) =
  if condition then
    if x ∈ Int then succ x else assert false
  else
    if x ∈ Bool then not x else assert false
```

Motivating Example (2/3)

Set-theoretic version:

```
let f (condition : Bool) (x : (Int | Bool))
      : (Int | Bool) =
  if condition then
    if x ∈ Int then succ x else assert false
  else
    if x ∈ Bool then not x else assert false
```

→ Syntactically heavy, but safe

Motivating Example (3/3)

Mixing the two:

```
let f (condition : Bool) (x : (Int | Bool) & ?)
      : (Int | Bool) =
  if condition then
    succ x
  else
    not x
```

Motivating Example (3/3)

Mixing the two:

```
let f (condition : Bool) (x : (Int | Bool) & ?)
      : (Int | Bool) =
  if condition then
    succ x
  else
    not x
```

→ Cannot be applied to something else than an integer or a boolean, and has a precise return type
→ Syntactically straightforward

Summary of the Motivations

– Gradualization of single expressions is sometimes too coarse

– Set-theoretic types are powerful but syntactically heavy (no reconstruction)

Summary of the Motivations

– Gradualization of single expressions is sometimes too coarse

– Set-theoretic types are powerful but syntactically heavy (no reconstruction)

Mixing the two would:

- Make the transition between dynamic types and static types smoother
- Reduce the syntactic overhead of set-theoretic types

Outline

Type Syntax

$$t \in \text{STypes} ::= b \mid t \to t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$$

$$\tau \in \text{GTypes} ::= \textbf{?} \mid b \mid \tau \to \tau \mid \tau \vee \tau \mid \tau \wedge \tau \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$$

Type Syntax

$$t \in \text{STypes} ::= b \mid t \to t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$$

$$\tau \in \text{GTypes} ::= \texttt{?} \mid b \mid \tau \to \tau \mid \tau \vee \tau \mid \tau \wedge \tau \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$$

Note: $\texttt{?} \neq \texttt{Any}$

$\texttt{Any}$ = unknown type, explicitly deconstructed

$\texttt{?}$ = unknown type, implicitly deconstructed

Type Semantics: Concretization

First idea: apply AGT [Garcia et al., 2016]

Type Semantics: Concretization

First idea: apply AGT [Garcia et al., 2016]

We define a *concretization function* $\gamma : \mathrm{GTypes} \to \mathcal{P}(\mathrm{STypes})$.

$$\begin{aligned}
\gamma(\mathbf{?}) &= \mathrm{STypes} \\
\gamma(\tau_1 \vee \tau_2) &= \{t_1 \vee t_2 \mid t_i \in \gamma(\tau_i)\} \\
\gamma(\tau_1 \to \tau_2) &= \{t_1 \to t_2 \mid t_i \in \gamma(\tau_i)\} \\
\gamma(b) &= \{b\}
\end{aligned}$$

etc...

Type Semantics: Concretization

First idea: apply AGT [Garcia et al., 2016]

We define a *concretization function* $\gamma : \mathrm{GTypes} \to \mathcal{P}(\mathrm{STypes})$.

$$\begin{aligned}
\gamma(\mathbf{?}) &= \mathrm{STypes} \\
\gamma(\tau_1 \vee \tau_2) &= \{t_1 \vee t_2 \mid t_i \in \gamma(\tau_i)\} \\
\gamma(\tau_1 \to \tau_2) &= \{t_1 \to t_2 \mid t_i \in \gamma(\tau_i)\} \\
\gamma(b) &= \{b\} \\
&\text{etc...}
\end{aligned}$$

For example,

$$\gamma((\mathbf{?} \to \mathtt{Int}) \wedge \mathbf{?}) = \{(t \to \mathtt{Int}) \wedge t' \mid (t, t') \in \mathrm{STypes}^2\}$$

Type Semantics: Subtyping (1/2)

Consistent subtyping [Garcia et al., 2016]

$$\sigma \widetilde{\leq} \tau \iff \exists (s, t) \in \gamma(\sigma) \times \gamma(\tau), s \leq t$$

Type Semantics: Subtyping (1/2)

Consistent subtyping [Garcia et al., 2016]

$$\sigma \mathrel{\widetilde{\leq}} \tau \iff \exists (s, t) \in \gamma(\sigma) \times \gamma(\tau), s \leq t$$

However, we can show the existence of "extremal" concretizations:

$$\forall t \in \gamma(\tau),\ \tau^{\Downarrow} \leq t \leq \tau^{\Uparrow}$$

Type Semantics: Subtyping (1/2)

Consistent subtyping [Garcia et al., 2016]

$$\sigma \lesssim \tau \iff \exists (s, t) \in \gamma(\sigma) \times \gamma(\tau), s \leq t$$

However, we can show the existence of "extremal" concretizations:

$$\forall t \in \gamma(\tau),\ \tau^{\Downarrow} \leq t \leq \tau^{\Uparrow}$$

$$(\texttt{?} \to \texttt{?})^{\Downarrow} = (\texttt{Any} \to \texttt{Empty}) \qquad (\texttt{?} \to \texttt{?})^{\Uparrow} = (\texttt{Empty} \to \texttt{Any})$$

Type Semantics: Subtyping (2/2)

Consistent subtyping

$$\sigma \mathrel{\widetilde{\leq}} \tau \iff \sigma^{\Downarrow} \leq \tau^{\Uparrow}$$

Type Semantics: Subtyping (2/2)

Consistent subtyping

$$\sigma \widetilde{\leq} \tau \iff \sigma^{\Downarrow} \leq \tau^{\Uparrow}$$

$\Longrightarrow$ Consistent subtyping reduces in linear time to semantic subtyping!

Type Semantics: Subtyping (2/2)

Consistent subtyping

$$\sigma \widetilde{\leq} \tau \iff \sigma^{\Downarrow} \leq \tau^{\Uparrow}$$

$\Longrightarrow$ Consistent subtyping reduces in linear time to semantic subtyping!

Note: emphasizes the fact that consistent subtyping is *not* transitive.

Type Semantics: Abstraction?

What about the abstraction function?

Type Semantics: Abstraction?

What about the abstraction function?

Problem: $\wedge$ and $\vee$ are *connectives*, not *constructors*.

Type Semantics: Abstraction?

What about the abstraction function?

Problem: $\wedge$ and $\vee$ are *connectives*, not *constructors*.

$$\alpha(\{\texttt{Int}\vee\texttt{Bool}, \texttt{Int}\vee\texttt{Int}, \texttt{Int}\vee\texttt{Empty}\}) =$$

Type Semantics: Abstraction?

What about the abstraction function?

Problem: $\wedge$ and $\vee$ are *connectives*, not *constructors*.

$$\alpha(\{\texttt{Int} \vee \texttt{Bool}, \texttt{Int} \vee \texttt{Int}, \texttt{Int} \vee \texttt{Empty}\}) =$$

Type Semantics: Abstraction?

What about the abstraction function?

Problem: $\wedge$ and $\vee$ are *connectives*, not *constructors*.

$$\alpha(\{\texttt{Int}\vee\texttt{Bool}, \texttt{Int}\vee\texttt{Int}, \texttt{Int}\vee\texttt{Empty}\})$$
$$= \alpha(\{\texttt{Int}, \texttt{Int}, \texttt{Int}\}) \vee \alpha(\{\texttt{Bool}, \texttt{Int}, \texttt{Empty}\})$$

Type Semantics: Abstraction?

What about the abstraction function?

Problem: $\wedge$ and $\vee$ are *connectives*, not *constructors*.

$$\begin{aligned}\alpha(\{\texttt{Int}\vee\texttt{Bool}, \texttt{Int}\vee\texttt{Int}, \texttt{Int}\vee\texttt{Empty}\}) \\ &= \alpha(\{\texttt{Int}, \texttt{Int}, \texttt{Int}\}) \vee \alpha(\{\texttt{Bool}, \texttt{Int}, \texttt{Empty}\}) \\ &= \texttt{Int} \vee\end{aligned}$$

Type Semantics: Abstraction?

What about the abstraction function?

Problem: $\wedge$ and $\vee$ are *connectives*, not *constructors*.

$$\begin{aligned}\alpha(\{\texttt{Int}\vee\texttt{Bool}, \texttt{Int}\vee\texttt{Int}, \texttt{Int}\vee\texttt{Empty}\}) \\ &= \alpha(\{\texttt{Int}, \texttt{Int}, \texttt{Int}\}) \vee \alpha(\{\texttt{Bool}, \texttt{Int}, \texttt{Empty}\}) \\ &= \texttt{Int} \vee \textbf{?}\end{aligned}$$

Type Semantics: Abstraction?

What about the abstraction function?

Problem: $\wedge$ and $\vee$ are *connectives*, not *constructors*.

$$\begin{aligned}\alpha(\{\texttt{Int}\vee\texttt{Bool},\texttt{Int}\vee\texttt{Int},\texttt{Int}\vee\texttt{Empty}\}) \\ &= \alpha(\{\texttt{Int},\texttt{Int},\texttt{Int}\}) \vee \alpha(\{\texttt{Bool},\texttt{Int},\texttt{Empty}\}) \\ &= \texttt{Int} \vee \textbf{?}\end{aligned}$$

$$\alpha(\{\texttt{Int} \vee \texttt{Bool}, \texttt{Bool} \vee \texttt{Int}\}) = \ ???$$

Type Semantics: Abstraction?

What about the abstraction function?

Problem: $\wedge$ and $\vee$ are *connectives*, not *constructors*.

$$\begin{aligned}\alpha(\{\texttt{Int}\vee\texttt{Bool},\texttt{Int}\vee\texttt{Int},\texttt{Int}\vee\texttt{Empty}\}) \\ &= \alpha(\{\texttt{Int},\texttt{Int},\texttt{Int}\}) \vee \alpha(\{\texttt{Bool},\texttt{Int},\texttt{Empty}\}) \\ &= \texttt{Int} \vee \textbf{?}\end{aligned}$$

$$\alpha(\{\texttt{Int} \vee \texttt{Bool}, \texttt{Bool} \vee \texttt{Int}\}) = \textbf{?} \vee \textbf{?}$$

Type Semantics: Abstraction?

What about the abstraction function?

Problem: $\wedge$ and $\vee$ are *connectives*, not *constructors*.

$$\begin{aligned}\alpha(\{\texttt{Int}\vee\texttt{Bool}, \texttt{Int}\vee\texttt{Int}, \texttt{Int}\vee\texttt{Empty}\}) \\ &= \alpha(\{\texttt{Int}, \texttt{Int}, \texttt{Int}\}) \vee \alpha(\{\texttt{Bool}, \texttt{Int}, \texttt{Empty}\}) \\ &= \texttt{Int} \vee \textbf{?}\end{aligned}$$

$$\alpha(\{\texttt{Int} \vee \texttt{Bool}, \texttt{Bool} \vee \texttt{Int}\}) = \texttt{Int} \vee \texttt{Bool}$$

Outline

Typing Applications: Modus Ponens

We start from the usual rule:

$$\frac{\Gamma \vdash e_1 : \sigma_1 \to \tau_1 \qquad \Gamma \vdash e_2 : \sigma_2 \qquad \sigma_2 \mathrel{\widetilde{\leq}} \sigma_1}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

Three components:

Typing Applications: Modus Ponens

We start from the usual rule:

$$\frac{\Gamma \vdash e_1 : \sigma_1 \to \tau_1 \qquad \Gamma \vdash e_2 : \sigma_2 \qquad \sigma_2 \mathrel{\widetilde{\leq}} \sigma_1}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

Three components:

- Domain

Typing Applications: Modus Ponens

We start from the usual rule:

$$\frac{\Gamma \vdash e_1 : \sigma_1 \to \tau_1 \qquad \Gamma \vdash e_2 : \sigma_2 \qquad \sigma_2 \mathrel{\widetilde{\leq}} \sigma_1}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

Three components:

- Domain
- Subtyping check

Typing Applications: Modus Ponens

We start from the usual rule:

$$\frac{\Gamma \vdash e_1 : \sigma_1 \to \tau_1 \qquad \Gamma \vdash e_2 : \sigma_2 \qquad \sigma_2 \widetilde{\leq} \sigma_1}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

Three components:

- Domain
- Subtyping check
- Result

Typing Applications: Modus Ponens Revisited

Problem: what is the domain of
$((\texttt{Int} \to \texttt{Bool}) \wedge \neg\texttt{Int}) \vee (\neg(\texttt{Bool} \to \texttt{Int}) \wedge (\texttt{Int} \to \texttt{Int}))$?

Typing Applications: Modus Ponens Revisited

Problem: what is the domain of
$((\texttt{Int} \to \texttt{Bool}) \wedge \neg\texttt{Int}) \vee (\neg(\texttt{Bool} \to \texttt{Int}) \wedge (\texttt{Int} \to \texttt{Int}))$?

What is the result of
$((\texttt{Int} \to \texttt{Bool}) \wedge (\texttt{Bool} \to \texttt{Int})) \vee (\texttt{Nat} \to \texttt{Nat})$ applied to $\texttt{Nat}$?

Typing Applications: Modus Ponens Revisited

Problem: what is the domain of
$((\texttt{Int} \to \texttt{Bool}) \wedge \neg\texttt{Int}) \vee (\neg(\texttt{Bool} \to \texttt{Int}) \wedge (\texttt{Int} \to \texttt{Int}))$?

What is the result of
$((\texttt{Int} \to \texttt{Bool}) \wedge (\texttt{Bool} \to \texttt{Int})) \vee (\texttt{Nat} \to \texttt{Nat})$ applied to $\texttt{Nat}$?

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash e_2 : \tau \qquad \tau \mathrel{\widetilde{\leq}} \widetilde{dom}(\sigma)}{\Gamma \vdash e_1\, e_2 : \sigma \mathbin{\widetilde{\circ}} \tau}$$

Typing Applications: Modus Ponens Revisited

Problem: what is the domain of
$((\texttt{Int} \to \texttt{Bool}) \wedge \neg\texttt{Int}) \vee (\neg(\texttt{Bool} \to \texttt{Int}) \wedge (\texttt{Int} \to \texttt{Int}))$?

$\texttt{Int}$

What is the result of
$((\texttt{Int} \to \texttt{Bool}) \wedge (\texttt{Bool} \to \texttt{Int})) \vee (\texttt{Nat} \to \texttt{Nat})$ applied to $\texttt{Nat}$?

$\texttt{Nat} \vee \texttt{Bool}$

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash e_2 : \tau \qquad \tau \mathrel{\widetilde{\leq}} \widetilde{dom}(\sigma)}{\Gamma \vdash e_1\, e_2 : \sigma \mathbin{\widetilde{\circ}} \tau}$$

Function Operators: Examples

$$\widetilde{dom}((\texttt{Int} \to \texttt{Int}) \wedge (\texttt{?} \to \texttt{?})) =$$

Function Operators: Examples

$\widetilde{dom}((\texttt{Int} \to \texttt{Int}) \wedge (\texttt{?} \to \texttt{?})) = \texttt{Any}$ since it can accept any value

Function Operators: Examples

$\widetilde{dom}((\texttt{Int} \to \texttt{Int}) \wedge (\texttt{?} \to \texttt{?})) = \texttt{Any}$ since it can accept any value

$\widetilde{dom}((\texttt{Int} \to \texttt{Int}) \vee (\texttt{?} \to \texttt{?})) =$

Function Operators: Examples

$\widetilde{dom}((\texttt{Int} \to \texttt{Int}) \wedge (\texttt{?} \to \texttt{?})) = \texttt{Any}$ since it can accept any value

$\widetilde{dom}((\texttt{Int} \to \texttt{Int}) \vee (\texttt{?} \to \texttt{?})) = \texttt{Int}$

Function Operators: Examples

$\widetilde{dom}((\texttt{Int} \to \texttt{Int}) \wedge (\texttt{?} \to \texttt{?})) = \texttt{Any}$ since it can accept any value

$\widetilde{dom}((\texttt{Int} \to \texttt{Int}) \vee (\texttt{?} \to \texttt{?})) = \texttt{Int}$

$((\texttt{?} \to \texttt{Int}) \wedge (\texttt{?} \to \texttt{Bool}))\widetilde{\circ}\texttt{Int} =$

Function Operators: Examples

$\widetilde{dom}((\texttt{Int} \to \texttt{Int}) \wedge (\texttt{?} \to \texttt{?})) = \texttt{Any}$ since it can accept any value

$\widetilde{dom}((\texttt{Int} \to \texttt{Int}) \vee (\texttt{?} \to \texttt{?})) = \texttt{Int}$

$((\texttt{?} \to \texttt{Int}) \wedge (\texttt{?} \to \texttt{Bool}))\widetilde{\circ}\texttt{Int} = \texttt{Int} \vee \texttt{Bool}$

Outline

Conclusion and Future Work

Current results:

- \+ Efficient characterization of consistent subtyping
- \+ Sound system
- \+ Supports polymorphism (W.I.P.)
- – No blame theorem, and no gradual guarantee yet

Conclusion and Future Work

Current results:

- \+ Efficient characterization of consistent subtyping
- \+ Sound system
- \+ Supports polymorphism (W.I.P.)
- – No blame theorem, and no gradual guarantee yet

Future work:

- – Provide a statically-typed cast-calculus and prove the gradual guarantee
- – Study blame