A Semantic Foundation for Gradual Set-theoretic Types

GPL Award Accessit

Victor Lanvin

June 7th, 2022

Gradual Typing

- Goal: have both **static** and **dynamic** typing in the same language.

- How: by adding a **dynamic type**, denoted "`?`".

Gradual Typing

- Goal: have both **static** and **dynamic** typing in the same language.
- How: by adding a **dynamic type**, denoted "?".
- **Allows for a trade-off between safety and programming productivity.**

Gradual Typing

- Goal: have both **static** and **dynamic** typing in the same language.
- How: by adding a **dynamic type**, denoted "?".
- **Allows for a trade-off between safety and programming productivity.**

The transition is **gradual**:

$$? \preccurlyeq ? \to ? \preccurlyeq \mathtt{Int} \to ? \preccurlyeq \mathtt{Int} \to \mathtt{Bool}$$

Set-Theoretic Types

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

Set-Theoretic Types

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

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

Set-Theoretic Types

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

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

`if x then 3 else true :` `Int` $\vee$ `Bool`

Set-Theoretic Types

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

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

  `if x then 3 else true` : `Int` $\vee$ `Bool`

  $(\mathtt{x} = \mathtt{e} \in \mathtt{Int})\,?\,\mathtt{true} : \mathtt{x}$: `Bool` $\vee$ $\neg$ `Int`

Set-Theoretic Types

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

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

  `if x then 3 else true :` `Int` $\vee$ `Bool`

  `(x = e` $\in$ `Int) ? true : x :` `Bool` $\vee$ $\neg$ `Int`

- **Powerful but often syntactically heavy.**

Set-Theoretic Types

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

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

  `if x then 3 else true` : `Int` $\vee$ `Bool`

  `(x = e ∈ Int) ? true : x` : `Bool` $\vee$ $\neg$ `Int`

- **Powerful but often syntactically heavy.**

- 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 (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 ⟨α list⟩
  else
    Array.map f 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:

```
let map (condition : Bool) (f : α -> β) (data : ?) :  ? =
  if condition then
    List.map f data ⟨α list⟩
  else
    Array.map f data ⟨α array⟩
```

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:

```
let map (condition : Bool) (f : α -> β) (data : ?) :  ? =
  if condition then
    List.map f data ⟨α list⟩
  else
    Array.map f data ⟨α array⟩
```

Runtime **checks** or **casts** are then inserted **automatically** by the compiler.

This is however very **unsafe**, as it accepts a **string** for example.

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
```

- By **subtyping**, $(\alpha\ \texttt{list} \vee \alpha\ \texttt{array}) \wedge\ ? \;\leq\; ?$.

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
```

- By **subtyping**, $(\alpha\ \texttt{list} \vee \alpha\ \texttt{array}) \wedge\ ? \ \leq\ ?$.
- 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
```

- By **subtyping**, $(\alpha\ \texttt{list} \vee \alpha\ \texttt{array}) \wedge\ ? \;\leq\; ?$.
- Can only be used with **lists or arrays**.
- No need for **manual type checks**.
- We want to infer **all non-gradual types** (including the return type).

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
```

- By **subtyping**, $(\alpha\ \texttt{list} \vee \alpha\ \texttt{array}) \wedge\ ? \leq\ ?$.
- Can only be used with **lists or arrays**.
- No need for **manual type checks**.
- We want to infer **all non-gradual types** (including the return type).

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
```

- By **subtyping**, $(\alpha$ `list` $\vee$ α `array`$) \wedge$ `?` $\leq$ `?`.
- Can only be used with **lists or arrays**.
- No need for **manual type checks**.
- We want to infer **all non-gradual types** (including the return type).

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
```

- By **subtyping**, $(\alpha$ `list` $\vee\ \alpha$ `array`$) \wedge\ ?\ \leq\ ?$.
- Can only be used with **lists or arrays**.
- No need for **manual type checks**.
- We want to infer **all non-gradual types** (including the return type).

How it is Usually Done

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

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

How it is Usually Done

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

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

2. Embed this relation into typing rules.

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

How it is Usually Done

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

   This relation is not transitive! $? \mathrel{\widetilde{\leq}} \tau \mathrel{\widetilde{\leq}} ?$ 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 (First) Approach

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

Our (First) Approach

1. Translate gradual types to **static types** (types without $?$) **with variables**.
2. Define **transitive** relations on gradual types, and in particular **"precision"** which contains the **essence of gradual typing.**

Our (First) Approach

1. Translate gradual types to **static types** (types without ?) **with variables**.
2. Define **transitive** relations on gradual types, and in particular **“precision”** which contains the **essence of gradual typing.**
3. Embed precision into **more and more complex systems** (Hindley-Milner, with subtyping, and with semantic subtyping).

Our (First) Approach

1. Translate gradual types to **static types** (types without ?) **with variables**.
2. Define **transitive** relations on gradual types, and in particular **"precision"** which contains the **essence of gradual typing.**
3. Embed precision into **more and more complex systems** (Hindley-Milner, with subtyping, and with semantic subtyping).

   Important remark: this translation is **only used** to define and compute relations, and **is not done in the source program**.

Subtyping

Subtyping only allows us to move **inside** the dynamic world, or **inside** the static world. It **does not** allow crossing the barrier.

Subtyping

Subtyping only allows us to move **inside** the dynamic world, or **inside** the static world. It **does not** allow crossing the barrier.

As opposed to consistent subtyping, it is **transitive**:

$$? \leq ? \qquad ? \not\leq \texttt{Int} \qquad \texttt{Int} \not\leq ?$$

Subtyping

Subtyping only allows us to move **inside** the dynamic world, or **inside** the static world. It **does not** allow crossing the barrier.

As opposed to consistent subtyping, it is **transitive**:

$$? \leq ? \qquad ? \not\leq \texttt{Int} \qquad \texttt{Int} \not\leq ?$$

It can be used to handle unions and intersections, by **simply plugging-in** the static version of **semantic subtyping**:

$$? \leq ? \vee \texttt{Int} \qquad \texttt{Int} \wedge ? \leq ?$$

Precision

Precision is what allows us to **cross the barrier** from the dynamic world into the static world (**and only this way**).

Precision

Precision is what allows us to **cross the barrier** from the dynamic world into the static world (**and only this way**).

$$? \preccurlyeq \tau \qquad \text{for every } \tau$$

$$? \to ? \preccurlyeq \tau_1 \to \tau_2 \qquad \text{for every } \tau_1, \tau_2$$

Precision

Precision is what allows us to **cross the barrier** from the dynamic world into the static world (**and only this way**).

$$? \preccurlyeq \tau \quad \text{for every } \tau$$

$$? \to ? \preccurlyeq \tau_1 \to \tau_2 \quad \text{for every } \tau_1, \tau_2$$

And it is **transitive**:

$$? \ \preccurlyeq \ ? \to ? \ \preccurlyeq \ ? \to \mathtt{Int} \ \preccurlyeq \ \mathtt{Int} \to \mathtt{Int}$$

Precision

Precision is what allows us to **cross the barrier** from the dynamic world into the static world (**and only this way**).

$$? \preccurlyeq \tau \quad \text{for every } \tau$$

$$? \to ? \preccurlyeq \tau_1 \to \tau_2 \quad \text{for every } \tau_1, \tau_2$$

And it is **transitive**:

$$? \preccurlyeq ? \to ? \preccurlyeq ? \to \mathtt{Int} \preccurlyeq \mathtt{Int} \to \mathtt{Int}$$

Therefore it can be embedded into a type system as a **subsumption-like** rule: **materialization**.

Declarative Type Systems

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

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

Declarative Type Systems

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

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

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

Declarative Type Systems

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

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

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

Declarative Type Systems

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

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

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

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

Declarative Type Systems

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

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

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

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

Declarative Type Systems

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

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

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

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

And as a bonus, we get the **static gradual guarantee** for free!

Extremal Materializations

Theorem

For every type $\tau \in \texttt{GTypes}$, there exists $t_1, t_2 \in \texttt{STypes}$ such that:

$\tau \preccurlyeq t_1$ and $\tau \preccurlyeq t_2$

$\forall \tau' \in \texttt{GTypes}.\ \tau \preccurlyeq \tau' \implies t_1 \leq \tau' \leq t_2$

Extremal Materializations

Theorem

For every type $\tau \in \texttt{GTypes}$, there exists $t_1, t_2 \in \texttt{STypes}$ such that:

$$\tau \preccurlyeq t_1 \text{ and } \tau \preccurlyeq t_2$$

$$\forall \tau' \in \texttt{GTypes}.\ \tau \preccurlyeq \tau' \implies t_1 \leq \tau' \leq t_2$$

We write $t_1 = \tau^{\Downarrow}$ and $t_2 = \tau^{\Uparrow}$.

Theorem

For every type $\tau \in \texttt{GTypes}$, there exists $t_1, t_2 \in \texttt{STypes}$ such that:

$$\tau \preccurlyeq t_1 \text{ and } \tau \preccurlyeq t_2$$

$$\forall \tau' \in \texttt{GTypes}.\ \tau \preccurlyeq \tau' \implies t_1 \leq \tau' \leq t_2$$

We write $t_1 = \tau^{\Downarrow}$ and $t_2 = \tau^{\Uparrow}$.

$$(? \to ?)^{\Uparrow} = \mathbb{0} \to \mathbb{1} \qquad (? \to ?)^{\Downarrow} = \mathbb{1} \to \mathbb{0}$$

Extremal Materializations

Theorem

For every type $\tau \in \texttt{GTypes}$, there exists $t_1, t_2 \in \texttt{STypes}$ such that:

$$\tau \preccurlyeq t_1 \text{ and } \tau \preccurlyeq t_2$$

$$\forall \tau' \in \texttt{GTypes}.\ \tau \preccurlyeq \tau' \implies t_1 \leq \tau' \leq t_2$$

We write $t_1 = \tau^{\Downarrow}$ and $t_2 = \tau^{\Uparrow}$.

$$(? \to ?)^{\Uparrow} = \mathbb{0} \to \mathbb{1} \qquad (? \to ?)^{\Downarrow} = \mathbb{1} \to \mathbb{0}$$

These types are computed in **linear time**!

An Equivalent Representation of Gradual Types

We show the following:

$$\tau_1 \leq \tau_2 \iff \begin{cases} {\tau_1}^{\Downarrow} \;\leq\; {\tau_2}^{\Downarrow} \\ {\tau_1}^{\Uparrow} \;\leq\; {\tau_2}^{\Uparrow} \end{cases}$$

An Equivalent Representation of Gradual Types

We show the following:

$$\tau_1 \leq \tau_2 \iff \begin{cases} {\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow} \\ {\tau_1}^{\Uparrow} \leq {\tau_2}^{\Uparrow} \end{cases} \qquad \tau_1 \preccurlyeq \tau_2 \iff \begin{cases} {\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow} \\ {\tau_2}^{\Uparrow} \leq {\tau_1}^{\Uparrow} \end{cases}$$

An Equivalent Representation of Gradual Types

We show the following:

$$\tau_1 \leq \tau_2 \iff \begin{cases} {\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow} \\ {\tau_1}^{\Uparrow} \leq {\tau_2}^{\Uparrow} \end{cases} \qquad \tau_1 \preccurlyeq \tau_2 \iff \begin{cases} {\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow} \\ {\tau_2}^{\Uparrow} \leq {\tau_1}^{\Uparrow} \end{cases}$$

Moreover, we have that **for every gradual type** τ,

$$\tau \simeq \tau^{\Downarrow} \vee (? \wedge \tau^{\Uparrow})$$

An Equivalent Representation of Gradual Types

We show the following:

$$\tau_1 \leq \tau_2 \iff \begin{cases} \tau_1{}^{\Downarrow} \leq \tau_2{}^{\Downarrow} \\ \tau_1{}^{\Uparrow} \leq \tau_2{}^{\Uparrow} \end{cases} \qquad \tau_1 \preccurlyeq \tau_2 \iff \begin{cases} \tau_1{}^{\Downarrow} \leq \tau_2{}^{\Downarrow} \\ \tau_2{}^{\Uparrow} \leq \tau_1{}^{\Uparrow} \end{cases}$$

Moreover, we have that **for every gradual type** τ,

$$\tau \simeq \tau^{\Downarrow} \vee (? \wedge \tau^{\Uparrow})$$

We can use this representation to **lift operators** to gradual types!

$$\mathtt{dom}(\tau) \stackrel{\text{def}}{=} \mathtt{dom}(\tau^{\Uparrow}) \vee (? \wedge \mathtt{dom}(\tau^{\Downarrow}))$$

Conclusion

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

Conclusion

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

We present:

1. A simple method of **declaratively adding** gradual typing to any existing type system.

Conclusion

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

We present:

1. A simple method of **declaratively adding** gradual typing to any existing type system.
2. A **set-theoretic interpretation** of gradual types that has considerable consequences.

Conclusion

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

We present:

1. A simple method of **declaratively adding** gradual typing to any existing type system.
2. A **set-theoretic interpretation** of gradual types that has considerable consequences.
3. The **algorithmic systems** for our GTLC with set-theoretic types.

Conclusion

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

We present:

1. A simple method of **declaratively adding** gradual typing to any existing type system.
2. A **set-theoretic interpretation** of gradual types that has considerable consequences.
3. The **algorithmic systems** for our GTLC with set-theoretic types.
4. **Denotational semantics** for several calculi, including CDuce, and a GTLC without set-theoretic types.

Future work

– Fully **unify** our logical approach and our denotational semantics.

Future work

- Fully **unify** our logical approach and our denotational semantics.
- Sound and complete **type inference** for gradual set-theoretic types.

Future work

- Fully **unify** our logical approach and our denotational semantics.
- Sound and complete **type inference** for gradual set-theoretic types.
- Add **more features** to our calculus, such as intersection types for functions.

Future work

– Fully **unify** our logical approach and our denotational semantics.

– Sound and complete **type inference** for gradual set-theoretic types.

– Add **more features** to our calculus, such as intersection types for functions.

– A **denotational semantics** for a cast calculus with set-theoretic types.