Automating Gradual Typing

or; Abstracting Abstracting Gradual Typing

Timothy Jones <`tim@ecs.vuw.ac.nz`>

July 17, 2016

Victoria University of Wellington

Gradual Typing

Lift a type system into its gradual form

- Abstracting Gradual Typing
- Gradualizer

Mechanisation

Types

```
data Type : Set where
  Int : Type
  Bool : Type
  _→_ : (T₁ T₂ : Type) → Type
```

Terms

```
data Term n : Set where
  int : ℤ → Term n
  bool : 𝔹 → Term n
  ...
  _·_ : (t₁ t₂ : Term n) → Term n
  ...
  if_then_else_ : (t₁ t₂ t₃ : Term n) → Term n
```

Typing

```
data _⊢_:_ {n} (Γ : Vec Type n) : Term n → Type → Set where
  int : ∀ {x} → Γ ⊢ int x : Int
  bool : ∀ {x} → Γ ⊢ bool x : Bool
  ...
```

Typing

```
data _⊢_:_ {n} (Γ : Vec Type n) : Term n → Type → Set where
  int : ∀ {x} → Γ ⊢ int x : Int
  bool : ∀ {x} → Γ ⊢ bool x : Bool
  ...
  app : ∀ {t₁ t₂ T T₁ T₂}  → Γ ⊢ t₁ : T → Γ ⊢ t₂ : T₁
                           → T := T₁ → T₂
                           → Γ ⊢ t₁ · t₂ : T₂
  ...
```

Typing

$$
\begin{aligned}
&\text{data } \_\vdash\_:\_ \ \{n\}\ (\Gamma : \text{Vec Type } n) : \text{Term } n \to \text{Type} \to \text{Set where}\\
&\quad \text{int} : \forall\ \{x\} \to \Gamma \vdash \text{int } x : \text{Int}\\
&\quad \text{bool} : \forall\ \{x\} \to \Gamma \vdash \text{bool } x : \text{Bool}\\
&\quad \ldots\\
&\quad \text{app} : \forall\ \{t_1\ t_2\ T\ T_1\ T_2\}\ \to \Gamma \vdash t_1 : T \to \Gamma \vdash t_2 : T_1\\
&\qquad\qquad\qquad\qquad\qquad\ \to T := T_1 \to T_2\\
&\qquad\qquad\qquad\qquad\qquad\ \to \Gamma \vdash t_1 \cdot t_2 : T_2\\
&\quad \ldots\\
&\quad \text{cond} : \forall\ \{t_1\ t_2\ t_3\ T\ T_1\ T_2\}\ \to \Gamma \vdash t_1 : \text{Bool}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \Gamma \vdash t_2 : T_1 \to \Gamma \vdash t_3 : T_2\\
&\qquad\qquad\qquad\qquad\qquad\qquad \to T := T_1 \sqcap T_2\\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T
\end{aligned}
$$

Lifting Relations

$$\mathrm{Lift}^1 : (\mathrm{Type} \to \mathrm{Set}) \to \mathrm{GType} \to \mathrm{Set}$$

$$\mathrm{Lift}^2 : (\mathrm{Type} \to \mathrm{Type} \to \mathrm{Set}) \to \mathrm{GType} \to \mathrm{GType} \to \mathrm{Set}$$

...

Concretisation

$$\gamma : \mathsf{GType} \to \mathbb{P}\ \mathsf{Type}$$

Concretisation

```
data γ : GType → Type → Set where
  ? : ∀ {T} → γ ? T
  Int : γ Int Int
  Bool : γ Bool Bool
  _→_ : ∀ {T̃₁ T̃₂ T₁ T₂} → γ T̃₁ T₁
                          → γ T̃₂ T₂
                          → γ (T̃₁ → T̃₂) (T₁ → T₂)
```

Lifting Relations

```
data Lift² (_≈_ : Rel Type) (T̃₁ T̃₂ : GType) : Set where
  raise : ∀ {T₁ T₂} → T₁ ≈ T₂
                    → T₁ ∈ γ T̃₁
                    → T₂ ∈ γ T̃₂
                    → Lift² _≈_ T̃₁ T̃₂
```

Consistent Equality

$$\_\cong\_ = \mathsf{Lift}^2\ \_\equiv\_$$

$$\mathsf{example} : \mathsf{Int} \to ?\ \cong\ ? \to \mathsf{Bool}$$
$$\mathsf{example} = \mathsf{raise}\ \{\mathsf{Int} \to \mathsf{Bool}\}\ \mathsf{refl}\ (\mathsf{Int} \to ?)\ (? \to \mathsf{Bool})$$

Lifting Functions

$\text{lift}^1 : (\text{Type} \to \text{Type}) \to \text{GType} \to \text{GType}$

$\text{lift}^2 : (\text{Type} \to \text{Type} \to \text{Type}) \to \text{GType} \to \text{GType} \to \text{GType}$

...

Abstraction

Not computable

$$\alpha : \mathbb{P}\ \mathrm{Type} \rightarrow \mathrm{GType}$$

Cannot just lift equality predicates: must preserve optimality

Ad-hoc solutions?

```
data _:=_→_ : GType → GType → GType → Set where
    refl : ∀ {T̃₁ T̃₂} → (T̃₁ → T̃₂) := T̃₁ → T̃₂
    ? : ? := ? → ?
```

Automation

Abstracting Language Implementation

Gradual Typing as a library

- Describe languages in a uniform, abstract way
- Provide a mechanism to apply this abstraction
- Different type systems for different applications

GType = Maybe?

```
data Maybe A : Set where
  ? : Maybe A
  type : A → Maybe A
```

Maybe Type not enough

Abstractly Typed Functional Language

```
data Type (F : Set → Set) : Set where
  Int : Type F
  Bool : Type F
  _→_ : (T₁ T₂ : F (Type F)) → Type F
```

Abstractly Typed Functional Language

$$
\begin{aligned}
&\text{data Type } (F : \text{Set} \to \text{Set}) : \text{Set where} \\
&\quad \text{Int} : \text{Type } F \\
&\quad \text{Bool} : \text{Type } F \\
&\quad \_\!\to\!\_ : (T_1\ T_2 : F\ (\text{Type } F)) \to \text{Type } F
\end{aligned}
$$

$$
\begin{aligned}
&\text{Type} = \text{id } (\text{Type id}) \\
&\text{GType} = \text{Maybe } (\text{Type Maybe})
\end{aligned}
$$

Abstractly Typed Functional Language

```
data Type (F : Set → Set) : Set where
  Int : Type F
  Bool : Type F
  _→_ : (T₁ T₂ : F (Type F)) → Type F

Type = id (Type id)
GType = Maybe (Type Maybe)
```

(Not necessarily strictly positive — no way to negotiate this)

F is for Functor

$\text{Type } (F : \text{Set} \to \text{Set}) : \text{Set}$

$\text{lift} : \forall \, \{A \; B\} \to (A \to B) \to F \, A \to F \, B$

F is for Functor

$\text{Type}\ (F : \text{Set} \to \text{Set}) : \text{Set}$

$\text{lift} : \forall\ \{A\ B\} \to (A \to B) \to F\ A \to F\ B$

$\text{unit} : \forall\ \{A\} \to A \to F\ A$

Using Unit

$$
\begin{aligned}
&\text{data } \_\vdash\_:\_ \ \{n\}\ (\Gamma : \text{Vec}\ (F\ (\text{Type}\ F))\ n) : \text{Term}\ n \ \to F\ (\text{Type}\ F) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \text{Set where} \\
&\quad \text{int} : \forall\ \{x\} \to \Gamma \vdash \text{int}\ x : \text{unit Int} \\
&\quad \text{bool} : \forall\ \{x\} \to \Gamma \vdash \text{bool}\ x : \text{unit Bool} \\
&\quad \ldots \\
&\quad \text{app} : \forall\ \{t_1\ t_2\ T\ T_1\ T_2\} \ \to \Gamma \vdash t_1 : T \to \Gamma \vdash t_2 : T_1 \\
&\qquad\qquad\qquad\qquad\qquad\quad \to T := T_1 \to T_2 \\
&\qquad\qquad\qquad\qquad\qquad\quad \to \Gamma \vdash t_1 \cdot t_2 : T_2 \\
&\quad \ldots \\
&\quad \text{cond} : \forall\ \{t_1\ t_2\ t_3\ T\ T_1\ T_2\} \ \to \Gamma \vdash t_1 : \text{unit Bool} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \Gamma \vdash t_2 : T_1 \to \Gamma \vdash t_3 : T_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \to T := T_1 \sqcap T_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T
\end{aligned}
$$

Abstracting Concretisation

Implementing γ relied on knowing the shape of Type

```
data γ : GType → Type → Set where
  ? : ∀ {T} → γ ? T
  Int : γ Int Int
  Bool : γ Bool Bool
  _→_ : ∀ {T̃₁ T̃₂ T₁ T₂}  → γ T̃₁ T₁
                         → γ T̃₂ T₂
                         → γ (T̃₁ → T̃₂) (T₁ → T₂)
```

We need to have a single rule for all variants of Type

```
data γ : GType → Type → Set where
  ? : ∀ {T} → γ ? T
  type :  (T : Type □)
          → γ  (type   T → Type Maybe)
               (id     T → Type id)
```

Mapping Functors

Need a mechanism to transform the indexed functor

$$\begin{aligned}\mathsf{map} : \forall\ \{F\ G\}\ &\to (F\ (\mathsf{Type}\ G) \to G\ (\mathsf{Type}\ G))\\ &\to \mathsf{Type}\ F \to \mathsf{Type}\ G\end{aligned}$$

In the ATFL

$$
\begin{aligned}
\text{map}\ f\ \text{Int} &= \text{Int} \\
\text{map}\ f\ \text{Bool} &= \text{Bool} \\
\text{map}\ f\ (T_1 \to T_2) &= f\ (\text{lift}\ (\text{map}\ f)\ T_1) \to f\ (\text{lift}\ (\text{map}\ f)\ T_2)
\end{aligned}
$$

In the ATFL

$$\begin{aligned} \text{map}\; f\; \text{Int} &= \text{Int} \\ \text{map}\; f\; \text{Bool} &= \text{Bool} \\ \text{map}\; f\; (T_1 \rightarrow T_2) &= f\; (\text{lift}\; (\text{map}\; f)\; T_1) \rightarrow f\; (\text{lift}\; (\text{map}\; f)\; T_2) \end{aligned}$$

(Not guaranteed to terminate — also no way to negotiate this)

Now we just need to choose the initial functor

```
data γ : GType → Type → Set where
  ? : ∀ {T} → γ ? T
  type :  (T : Type □)
          → γ  (type  (map □ → Maybe T))
               (id    (map □ → id T))
```

Constant Functor

We don't care about the recursive type: ignore it

```
data γ : GType → Type → Set where
  ? : ∀ {T} → γ ? T
  type : (T : Type (const ○))
         → γ (type (map ○ → GType T))
             (id   (map ○ → Type T))
```

Manual Recursion

Embed a pair of GType and Type at each point of recursion

```
data γ : GType → Type → Set where
  ? : ∀ {T} → γ ? T
  type :  (T : Type (const (GType × Type)))
          → γ  (type  (map proj₁ T))
               (id    (map proj₂ T))
```

Recursive Proof

γ ensured that matching components were recursively related

```
data γ : GType → Type → Set where
  ? : ∀ {T} → γ ? T
  Int : γ Int Int
  Bool : γ Bool Bool
  _→_ : ∀ {T̃₁ T̃₂ T₁ T₂} → γ T̃₁ T₁
                        → γ T̃₂ T₂
                        → γ (T̃₁ → T̃₂) (T₁ → T₂)
```

Recursive Proof

Also embed a proof that the elements of the pair are related

```
data γ : GType → Type → Set where
  ? : ∀ {T} → γ ? T
  type : (T : Type (const (Σ (GType × Type) (uncurry γ))))
       → γ (type (map (proj₁ ∘ proj₁) T))
           (id   (map (proj₂ ∘ proj₁) T))
```

Abstracted Consistent Equality

$\_\cong\_ = \mathsf{Lift}^2\ \_\equiv\_$

$\mathsf{example} : \mathsf{type}\ (\mathsf{type\ Int} \to ?) \cong \mathsf{type}\ (? \to \mathsf{type\ Bool})$

$\mathsf{example} =$

$\quad \mathsf{raise}\ \ \{\mathsf{Int} \to \mathsf{Bool}\}\ \mathsf{refl}$

$\qquad\qquad \mathsf{Int} \to ?$

$\qquad\qquad ? \to \mathsf{Bool}$

Abstracted Consistent Equality

$$\_\cong\_ = \mathsf{Lift}^2\ \_\equiv\_$$

$$\begin{aligned}
&\mathsf{example} : \mathsf{type}\ (\mathsf{type\ Int} \to ?) \cong \mathsf{type}\ (? \to \mathsf{type\ Bool})\\
&\mathsf{example} =\\
&\quad \mathsf{raise}\ \ \{\mathsf{Int} \to \mathsf{Bool}\}\ \mathsf{refl}\\
&\qquad\qquad (\mathsf{type}\ (((\mathsf{type\ Int}\ ,\ \mathsf{Int})\ ,\ \mathsf{type\ Int}) \to ((?\ ,\ \mathsf{Bool})\ ,\ ?)))\\
&\qquad\qquad (\mathsf{type}\ (((?\ ,\ \mathsf{Bool})\ ,\ ?) \to ((\mathsf{type\ Bool}\ ,\ \mathsf{Bool})\ ,\ \mathsf{type\ Bool})))
\end{aligned}$$

$$\_\cong\_ = \mathsf{Lift}^2\ \_\equiv\_$$

$$
\begin{aligned}
&\mathsf{example} : \mathsf{type}\ (\mathsf{type}\ \mathsf{Int} \to ?) \cong \mathsf{type}\ (? \to \mathsf{type}\ \mathsf{Bool})\\
&\mathsf{example} =\\
&\quad \mathsf{raise}\ \ \{\mathsf{Int} \to \mathsf{Bool}\}\ \mathsf{refl}\\
&\qquad\qquad (\mathsf{type}\ ((,\ \mathsf{type}\ \mathsf{Int}) \to (,\ ?)))\\
&\qquad\qquad (\mathsf{type}\ ((,\ ?) \to (,\ \mathsf{type}\ \mathsf{Bool})))
\end{aligned}
$$

Abstraction

Other Functors = Other Type Systems

$$\text{Type} = \text{id}\ (\text{Type id})$$

$$\text{GType} = \text{Maybe}\ (\text{Type Maybe})$$

Other Functors = Other Type Systems

$$\text{Type} = \text{id}\ (\text{Type id})$$

$$\text{GType} = \text{Maybe}\ (\text{Type Maybe})$$

$$\text{DType} = \text{const}\ \top\ (\text{Type}\ (\text{const}\ \top))$$

Other Functors = Other Type Systems

$$\mathsf{Type} = \mathsf{id}\ (\mathsf{Type}\ \mathsf{id})$$

$$\mathsf{GType} = \mathsf{Maybe}\ (\mathsf{Type}\ \mathsf{Maybe})$$

$$\mathsf{DType} = \mathsf{const}\ \top\ (\mathsf{Type}\ (\mathsf{const}\ \top))$$

$$\mathsf{LType} = \mathsf{List}\ (\mathsf{Type}\ \mathsf{List})$$

Other Functors = Other Type Systems

$$\mathsf{Type} = \mathsf{id}\ (\mathsf{Type}\ \mathsf{id})$$

$$\mathsf{GType} = \mathsf{Maybe}\ (\mathsf{Type}\ \mathsf{Maybe})$$

$$\mathsf{DType} = \mathsf{const}\ \top\ (\mathsf{Type}\ (\mathsf{const}\ \top))$$

$$\mathsf{LType} = \mathsf{List}\ (\mathsf{Type}\ \mathsf{List})$$

$$\mathsf{EType} = (A : \mathsf{Set}) \to A \uplus \mathsf{Type}\ \lambda\ T \to A \uplus T$$

Other Functors = Other Type Systems

$\mathsf{Type} = \mathsf{id}\ (\mathsf{Type}\ \mathsf{id})$

$\mathsf{GType} = \mathsf{Maybe}\ (\mathsf{Type}\ \mathsf{Maybe})$

$\mathsf{DType} = \mathsf{const}\ \top\ (\mathsf{Type}\ (\mathsf{const}\ \top))$

$\mathsf{LType} = \mathsf{List}\ (\mathsf{Type}\ \mathsf{List})$

$\mathsf{EType} = (A : \mathsf{Set}) \to A \uplus \mathsf{Type}\ \lambda\ T \to A \uplus T$

$\mathsf{RType} = (A : \mathsf{Set}) \to A \to \mathsf{Type}\ \lambda\ T \to A \to T$

$\mathsf{WType} = \forall\ \{A\} \to \mathsf{Monoid}\ A \to A \times \mathsf{Type}\ \lambda\ T \to A \times T$

Other Functors = Other Type Systems

$$\mathsf{Type} = \mathsf{id}\ (\mathsf{Type}\ \mathsf{id})$$

$$\mathsf{GType} = \mathsf{Maybe}\ (\mathsf{Type}\ \mathsf{Maybe})$$

$$\mathsf{DType} = \mathsf{const}\ \top\ (\mathsf{Type}\ (\mathsf{const}\ \top))$$

$$\mathsf{LType} = \mathsf{List}\ (\mathsf{Type}\ \mathsf{List})$$

$$\mathsf{EType} = (A : \mathsf{Set}) \to A \uplus \mathsf{Type}\ \lambda\ T \to A \uplus T$$

$$\mathsf{RType} = (A : \mathsf{Set}) \to A \to \mathsf{Type}\ \lambda\ T \to A \to T$$

$$\mathsf{WType} = \forall\ \{A\} \to \mathsf{Monoid}\ A \to A \times \mathsf{Type}\ \lambda\ T \to A \times T$$

$$\mathsf{SType} = \forall\ \{A\} \to \mathsf{Monoid}\ A \to A \to \mathsf{Type}\ \lambda\ T \to A \to T \times A$$

Abstracting Abstracting Concretisation

The definition of γ was for gradual types only

$$
\begin{array}{l}
\textsf{data}\ \gamma : \textsf{GType} \to \textsf{Type} \to \textsf{Set}\ \textsf{where} \\
\quad ? : \forall\ \{T\} \to \gamma\ ?\ T \\
\quad \textsf{type} : (T : \textsf{Type}\ (\textsf{const}\ (\Sigma\ (\textsf{GType} \times \textsf{Type})\ (\textsf{uncurry}\ \gamma)))) \\
\qquad\qquad \to \gamma\ (\textsf{type}\ (\textsf{map}\ (\textsf{proj}_1 \circ \textsf{proj}_1)\ T)) \\
\qquad\qquad\qquad (\textsf{id}\ (\textsf{map}\ (\textsf{proj}_2 \circ \textsf{proj}_1)\ T))
\end{array}
$$

Abstracting Abstracting Concretisation

The definition of γ was for gradual types only

```
data γ : GType → Type → Set where
  ? : ∀ {T} → γ ? T
  type :  (T : Type (const (Σ (GType × Type) (uncurry γ))))
          → γ  (type  (map (proj₁ ∘ proj₁) T))
               (id    (map (proj₂ ∘ proj₁) T))
```

This looks suspiciously like an application of Maybe...

Abstracting Abstracting Concretisation

Define γ for any functor *F*

```
data γ {F} : F (Type F) → Type → Set where
  rel :  ∀ {T}
         → (x : F (Σ  (Type (const (Σ  (F (Type F) × Type)
                                        (uncurry γ))))
                      (_≡_ T ∘ map (proj₂ ∘ proj₁))))
         → γ (lift (map (proj₁ ∘ proj₁) ∘ proj₁) x) T
```

Abstracting Abstracting Concretisation

Define γ for any functor *F*

```
data γ {F} : F (Type F) → Type → Set where
  rel : ∀ {T}
        → (x : F (Σ  (Type (const (Σ  (F (Type F) × Type)
                                       (uncurry γ))))
                     (_≡_ T ∘ map (proj₂ ∘ proj₁))))
        → γ (lift (map (proj₁ ∘ proj₁) ∘ proj₁) x) T
```

Why is Type special?

Abstracting Abstracting Concretisation

Define γ for any *two* functors *F* and *G*

```
data γ {F G} : F (Type F) → G (Type G) → Set where
  rel : ∀ {T}
        → (x : F (Σ  (Type (const (Σ  (F (Type F) × G (Type G))
                                      (uncurry γ))))
                     (_≡_ T ∘ unit ∘ map (proj₂ ∘ proj₁))))
        → γ (lift (map (proj₁ ∘ proj₁) ∘ proj₁) x) T
```

Abstracting Abstracting Concretisation

Define γ for any *two* functors *F* and *G*

```
data γ {F G} : F (Type F) → G (Type G) → Set where
  rel :  ∀ {T}
         → (x : F (Σ  (Type (const (Σ  (F (Type F) × G (Type G))
                                        (uncurry γ))))
                      (_≡_ T ∘ unit ∘ map (proj₂ ∘ proj₁))))
         → γ (lift (map (proj₁ ∘ proj₁) ∘ proj₁) x) T
```

If the functors are the same, then γ is the precision relation ⊒

Abstracted Abstracted Consistent Equality

$\_\cong\_ = \text{Lift}^2\ \_\equiv\_$

$\text{example} : \text{type}\ (\text{type Int} \to ?) \cong \text{type}\ (? \to \text{type Bool})$
$\text{example} =$
$\quad \text{raise}\ \ \{\text{Int} \to \text{Bool}\}\ \text{refl}$
$\qquad (\text{rel}\ (\text{type}\ (((,\ \text{rel}\ (\text{type}\ (,\ \text{refl}))) \to (,\ \text{rel}\ ?))\ ,\ \text{refl})))$
$\qquad (\text{rel}\ (\text{type}\ (((,\ \text{rel}\ ?) \to (,\ \text{rel}\ (\text{type}\ (,\ \text{refl}))))\ ,\ \text{refl})))$

TODO

Github: `zmthy/automating-gradual-typing`

- Apply beyond STFL
- Investigate alternative type systems
- Dynamic semantics
- Proofs