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 _⊢_:_ $\{n\}$ $(\Gamma$: Vec Type $n)$: Term $n \rightarrow$ Type $\rightarrow$ Set where

  int : $\forall\ \{x\} \rightarrow \Gamma \vdash$ int x : Int

  bool : $\forall\ \{x\} \rightarrow \Gamma \vdash$ bool x : Bool

  ...

  app : $\forall\ \{t_1\ t_2\ T\ T_1\ T_2\} \rightarrow \Gamma \vdash t_1 : T \rightarrow \Gamma \vdash t_2 : T_1$

  $\rightarrow T := T_1 \rightarrow T_2$

  $\rightarrow \Gamma \vdash t_1 \cdot t_2 : T_2$

  ...

Typing

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

Lifting Relations

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

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

...

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

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{Lift}^2\ (\_{\approx}\_ : \mathsf{Rel\ Type})\ (\widetilde{T_1}\ \widetilde{T_2} : \mathsf{GType}) : \mathsf{Set}\ \mathsf{where}\\
&\quad \mathsf{raise} : \forall\ \{T_1\ T_2\} \to T_1 \approx T_2\\
&\qquad\qquad\qquad\qquad \to T_1 \in \gamma\ \widetilde{T_1}\\
&\qquad\qquad\qquad\qquad \to T_2 \in \gamma\ \widetilde{T_2}\\
&\qquad\qquad\qquad\qquad \to \mathsf{Lift}^2\ \_{\approx}\_\ \widetilde{T_1}\ \widetilde{T_2}
\end{aligned}
$$

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

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

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

...

Abstraction

Not computable

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

Cannot just lift equality predicates: must preserve optimality

Ad-hoc solutions?

$\mathsf{data}\ \_{:=}\_{\to}\_ : \mathsf{GType} \to \mathsf{GType} \to \mathsf{GType} \to \mathsf{Set}\ \mathsf{where}$
$\quad \mathsf{refl} : \forall\ \{\widetilde{T_1}\ \widetilde{T_2}\} \to (\widetilde{T_1} \to \widetilde{T_2}) := \widetilde{T_1} \to \widetilde{T_2}$
$\quad ? : ? := ? \to ?$

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 : $A \to$ 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

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

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

Abstractly Typed Functional Language

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

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

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

F is for Functor

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

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

F is for Functor

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

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

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

Using Unit

$$\mathsf{data}\ \_\vdash\_:\_\ \{n\}\ (\Gamma : \mathsf{Vec}\ (F\ (\mathsf{Type}\ F))\ n) : \mathsf{Term}\ n \to F\ (\mathsf{Type}\ F)$$
$$\to \mathsf{Set}\ \mathsf{where}$$

$$\mathsf{int} : \forall\ \{x\} \to \Gamma \vdash \mathsf{int}\ x : \mathsf{unit}\ \mathsf{Int}$$

$$\mathsf{bool} : \forall\ \{x\} \to \Gamma \vdash \mathsf{bool}\ x : \mathsf{unit}\ \mathsf{Bool}$$

...

$$\mathsf{app} : \forall\ \{t_1\ t_2\ T\ T_1\ T_2\} \to \Gamma \vdash t_1 : T \to \Gamma \vdash t_2 : T_1$$
$$\to T := T_1 \to T_2$$
$$\to \Gamma \vdash t_1 \cdot t_2 : T_2$$

...

$$\mathsf{cond} : \forall\ \{t_1\ t_2\ t_3\ T\ T_1\ T_2\} \to \Gamma \vdash t_1 : \mathsf{unit}\ \mathsf{Bool}$$
$$\to \Gamma \vdash t_2 : T_1 \to \Gamma \vdash t_3 : T_2$$
$$\to T := T_1 \sqcap T_2$$
$$\to \Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T$$

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

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

In the ATFL

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

In the ATFL

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

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

Abstracting Concretisation

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

$$
\begin{aligned}
&\mathsf{data}\ \gamma : \mathsf{GType} \to \mathsf{Type} \to \mathsf{Set}\ \mathsf{where}\\
&\quad ? : \forall\ \{T\} \to \gamma\ ?\ T\\
&\quad \mathsf{Int} : \gamma\ \mathsf{Int}\ \mathsf{Int}\\
&\quad \mathsf{Bool} : \gamma\ \mathsf{Bool}\ \mathsf{Bool}\\
&\quad \_{\to}\_ : \forall\ \{\widetilde{T_1}\ \widetilde{T_2}\ T_1\ T_2\} \to \gamma\ \widetilde{T_1}\ T_1\\
&\qquad\qquad\qquad\qquad\qquad\quad \to \gamma\ \widetilde{T_2}\ T_2\\
&\qquad\qquad\qquad\qquad\qquad\quad \to \gamma\ (\widetilde{T_1} \to \widetilde{T_2})\ (T_1 \to T_2)
\end{aligned}
$$

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

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

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

Abstracted Consistent Equality

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

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

Abstracted Consistent Equality

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

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

Abstraction

Other Functors = Other Type Systems

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

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

Other Functors = Other Type Systems

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

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

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

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

```
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))
```

The definition of γ was for gradual types only

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

This looks suspiciously like an application of Maybe...

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

$$
\begin{aligned}
&\mathsf{data}\ \gamma\ \{F\ G\} : F\ (\mathsf{Type}\ F) \to G\ (\mathsf{Type}\ G) \to \mathsf{Set}\ \mathsf{where}\\
&\quad \mathsf{rel} : \forall\ \{T\}\\
&\qquad \to (x : F\ (\Sigma\ (\mathsf{Type}\ (\mathsf{const}\ (\Sigma\ (F\ (\mathsf{Type}\ F) \times G\ (\mathsf{Type}\ G))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{uncurry}\ \gamma))))\\
&\qquad\qquad\qquad (\_{\equiv}\_\ T \circ \mathsf{unit} \circ \mathsf{map}\ (\mathsf{proj}_2 \circ \mathsf{proj}_1))))\\
&\qquad \to \gamma\ (\mathsf{lift}\ (\mathsf{map}\ (\mathsf{proj}_1 \circ \mathsf{proj}_1) \circ \mathsf{proj}_1)\ x)\ T
\end{aligned}
$$

Abstracting Abstracting Concretisation

Define γ for any *two* functors F and G

$$
\begin{array}{l}
\mathsf{data}\ \gamma\ \{F\ G\} : F\ (\mathsf{Type}\ F) \to G\ (\mathsf{Type}\ G) \to \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{rel} : \forall\ \{T\} \\
\qquad \to (x : F\ (\Sigma\ (\mathsf{Type}\ (\mathsf{const}\ (\Sigma\ (F\ (\mathsf{Type}\ F) \times G\ (\mathsf{Type}\ G)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{uncurry}\ \gamma)))) \\
\qquad\qquad\qquad (\_{\equiv}\_\ T \circ \mathsf{unit} \circ \mathsf{map}\ (\mathsf{proj}_2 \circ \mathsf{proj}_1)))) \\
\qquad \to \gamma\ (\mathsf{lift}\ (\mathsf{map}\ (\mathsf{proj}_1 \circ \mathsf{proj}_1) \circ \mathsf{proj}_1)\ x)\ T
\end{array}
$$

If the functors are the same, then γ is the precision relation $\sqsupseteq$

Abstracted Abstracted Consistent Equality

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

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

TODO

Github: `zmthy/automating-gradual-typing`

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