A Scope-and-Type Safe Universe of Syntaxes with Binding, Their Semantics and Proofs

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, James McKinna

ICFP'18, St Louis, MO

September 24, 2018

Road-map

Problem Statement

$$\mathsf{T} ::= x \mid \mathsf{T}\ \mathsf{T} \mid \lambda x.\mathsf{T}$$

Problem Statement

$$\mathsf{T} ::= x \mid \mathsf{T}\ \mathsf{T} \mid \lambda x.\mathsf{T}$$

Problem Statement

$$\mathsf{T} ::= x \mid \mathsf{T}\ \mathsf{T} \mid \lambda x.\mathsf{T}$$

Problem Statement

$$\mathsf{T} ::= x \mid \mathsf{T}\ \mathsf{T} \mid \lambda x.\mathsf{T}$$

Problem Statement

$$\mathsf{S} ::= x \mid \mathsf{S}\ \mathsf{S} \mid \lambda x.\mathsf{S} \mid \text{let } x = \mathsf{S} \text{ in } \mathsf{S}$$

$$\mathsf{T} ::= x \mid \mathsf{T}\ \mathsf{T} \mid \lambda x.\mathsf{T}$$

Problem Statement

$$\mathsf{S} ::= x \mid \mathsf{S}\ \mathsf{S} \mid \lambda x.\mathsf{S} \mid \text{let } x = \mathsf{S} \text{ in } \mathsf{S}$$

$$\mathsf{T} ::= x \mid \mathsf{T}\ \mathsf{T} \mid \lambda x.\mathsf{T}$$

Problem Statement

$$\mathsf{S} ::= x \mid \mathsf{S}\ \mathsf{S} \mid \lambda x.\mathsf{S} \mid \text{let } x = \mathsf{S} \text{ in } \mathsf{S}$$

$$\mathsf{T} ::= x \mid \mathsf{T}\ \mathsf{T} \mid \lambda x.\mathsf{T}$$

Problem Statement

$$
\begin{aligned}
S &::= x \mid S\ S \mid \lambda x.S \mid \text{let } x = S \text{ in } S \\
T &::= x \mid T\ T \mid \lambda x.T
\end{aligned}
$$

Problem

- *Write a program transformation from S to T inlining $let..in..$*
- *Prove a simulation lemma for this transformation*

Let-elaboration: from S to T

$$[\![\cdot]\!] \cdot : \mathsf{S} \rightarrow (\mathsf{Var} \Rightarrow \mathsf{T}) \rightarrow \mathsf{T}$$

Let-elaboration: from S to T

$$[\![\cdot]\!] \cdot : \mathsf{S} \to (\mathsf{Var} \Rightarrow \mathsf{T}) \to \mathsf{T}$$

Let-elaboration: from S to T

$$[\![\cdot]\!] \cdot : \mathsf{S} \to (\mathsf{Var} \Rightarrow \mathsf{T}) \to \mathsf{T}$$

Let-elaboration: from S to T

$$[\![\ \cdot\ ]\!]\ \cdot : \mathsf{S} \to (\mathsf{Var} \Rightarrow \mathsf{T}) \to \mathsf{T}$$

Let-elaboration: from S to T

$$[\![\cdot]\!] \cdot : \mathsf{S} \to (\mathsf{Var} \Rightarrow \mathsf{T}) \to \mathsf{T}$$
$$[\![x]\!] \rho = \rho(x)$$

Let-elaboration: from S to T

$$\begin{aligned}
[\![\ \cdot \ ]\!] \cdot &: \mathsf{S} \to (\mathsf{Var} \Rightarrow \mathsf{T}) \to \mathsf{T} \\
[\![\ x \ ]\!] \ \rho &= \rho(x) \\
[\![\ f \ \ t \ ]\!] \ \rho &= ([\![\ f \ ]\!] \ \rho) \ ([\![\ t \ ]\!] \ \rho)
\end{aligned}$$

Let-elaboration: from S to T

$$
\begin{aligned}
[\![\cdot]\!] \cdot &: \mathsf{S} \to (\mathsf{Var} \Rightarrow \mathsf{T}) \to \mathsf{T} \\
[\![x]\!]\ \rho &= \rho(x) \\
[\![f\ t]\!]\ \rho &= ([\![f]\!]\ \rho)\ ([\![t]\!]\ \rho) \\
[\![\lambda x.b]\!]\ \rho &= \lambda x.([\![b]\!]\ (\rho \cdot x))
\end{aligned}
$$

Let-elaboration: from S to T

$$\llbracket \cdot \rrbracket \cdot : \mathsf{S} \to (\mathsf{Var} \Rightarrow \mathsf{T}) \to \mathsf{T}$$

$$\llbracket x \rrbracket \rho = \rho(x)$$

$$\llbracket f\ t \rrbracket \rho = (\llbracket f \rrbracket \rho)\ (\llbracket t \rrbracket \rho)$$

$$\llbracket \lambda x.b \rrbracket \rho = \lambda x.(\llbracket b \rrbracket\ (\rho \cdot x))$$

Let-elaboration: from S to T

$$
\begin{aligned}
[\![\cdot]\!] \cdot &: \mathsf{S} \to (\mathsf{Var} \Rightarrow \mathsf{T}) \to \mathsf{T} \\
[\![x]\!] \rho &= \rho(x) \\
[\![f \ t]\!] \rho &= ([\![f]\!] \rho) \ ([\![t]\!] \rho) \\
[\![\lambda x.b]\!] \rho &= \lambda x.([\![b]\!] (\rho \cdot x)) \\
[\![\mathrm{let} \ x = e \ \mathrm{in} \ b]\!] \rho &= [\![b]\!] (\rho \cdot [\![e]\!] \rho)
\end{aligned}
$$

Let-elaboration: from S to T

$$
\begin{aligned}
[\![\cdot]\!] \cdot : \mathsf{S} &\to (\mathsf{Var} \Rightarrow \mathsf{T}) \to \mathsf{T} \\
[\![x]\!] \rho &= \rho(x) \\
[\![f\ t]\!] \rho &= ([\![f]\!] \rho)\ ([\![t]\!] \rho) \\
[\![\lambda x.b]\!] \rho &= \lambda x.([\![b]\!] (\rho \cdot x)) \\
[\![\mathrm{let}\ x = e\ \mathrm{in}\ b]\!] \rho &= [\![b]\!] (\rho \cdot [\![e]\!] \rho)
\end{aligned}
$$

Honesty tax (Ł1): T admits weakening

A Soundness Lemma

Lemma (Simulation)

Given:

We can prove that:

A Soundness Lemma

Lemma (Simulation)

Given:

- s and s' s.t. $s \rightsquigarrow_S s'$

We can prove that:

A Soundness Lemma

Lemma (Simulation)

Given:

- s *and* s*' s.t.* $s \rightsquigarrow_S s$*'*
- ρ *and* ρ*' s.t.* $\forall x.\rho(x) \rightsquigarrow^{\star}_{T} \rho'(x)$

We can prove that:

A Soundness Lemma

Lemma (Simulation)

Given:

- s *and* s' *s.t.* $s \leadsto_S s'$
- ρ *and* ρ' *s.t.* $\forall x.\rho(x) \leadsto_T^\star \rho'(x)$

We can prove that: $[\![\ s\ ]\!]\ \rho \leadsto_T^\star [\![\ s'\ ]\!]\ \rho'$

A Soundness Lemma

Lemma (Simulation)

Given:

- *s and s' s.t.* $s \leadsto_S s'$
- ρ *and* ρ' *s.t.* $\forall x.\rho(x) \leadsto_T^{\star} \rho'(x)$

We can prove that: $[\![\ s\ ]\!]\ \rho \leadsto_T^{\star} [\![\ s'\ ]\!]\ \rho'$

Honesty tax :

- Statement
  - $\leadsto_X$ means X is stable under substitution
- Proof

A Soundness Lemma

Lemma (Simulation)

Given:

- *s and s' s.t.* $s \rightsquigarrow_S s'$
- ρ *and* ρ*' s.t.* $\forall x.\rho(x) \rightsquigarrow_T^\star \rho'(x)$

We can prove that: $[\![\ s\ ]\!]\ \rho \rightsquigarrow_T^\star [\![\ s'\ ]\!]\ \rho'$

Honesty tax :

- Statement
  - $\rightsquigarrow_X$ means X is stable under substitution
  - To define substitution we need to define weakening
- Proof

A Soundness Lemma

Lemma (Simulation)

Given:

- *s and s' s.t.* $s \rightsquigarrow_S s'$
- ρ *and* ρ*' s.t.* $\forall x.\rho(x) \rightsquigarrow_T^\star \rho'(x)$

We can prove that: $[\![\ s\ ]\!]\ \rho \rightsquigarrow_T^\star [\![\ s'\ ]\!]\ \rho'$

Honesty tax :

- Statement
  - $\rightsquigarrow_X$ means X is stable under substitution
  - To define substitution we need to define weakening
- Proof
  - Fusion lemmas

A Soundness Lemma

Lemma (Simulation)

Given:

- s *and* s' *s.t.* $s \rightsquigarrow_S s'$
- ρ *and* ρ' *s.t.* $\forall x. \rho(x) \rightsquigarrow_T^\star \rho'(x)$

We can prove that: $[\![s]\!] \rho \rightsquigarrow_T^\star [\![s']\!] \rho'$

Honesty tax :

- Statement
  - $\rightsquigarrow_X$ means X is stable under substitution
  - To define substitution we need to define weakening
- Proof
  - Fusion lemmas
  - Extensionality lemmas

A Soundness Lemma

Lemma (Simulation)

Given:

- s *and* s' *s.t.* $s \rightsquigarrow_S s'$
- ρ *and* ρ' *s.t.* $\forall x.\rho(x) \rightsquigarrow^\star_T \rho'(x)$

We can prove that: $[\![\ s\ ]\!]\ \rho \rightsquigarrow^\star_T [\![\ s'\ ]\!]\ \rho'$

Honesty tax :

- Statement
  - $\rightsquigarrow_X$ means X is stable under substitution
  - To define substitution we need to define weakening
- Proof
  - Fusion lemmas
  - Extensionality lemmas
  - Identity lemmas

A Soundness Lemma

Lemma (Simulation)

Given:

- *s and s' s.t.* $s \rightsquigarrow_S s'$
- ρ *and* ρ*' s.t.* $\forall x. \rho(x) \rightsquigarrow^{\star}_T \rho'(x)$

We can prove that: $[\![\, s \,]\!] \, \rho \rightsquigarrow^{\star}_T [\![\, s' \,]\!] \, \rho'$

Honesty tax (Ł12+):

- Statement
  - $\rightsquigarrow_X$ means X is stable under substitution
  - To define substitution we need to define weakening
- Proof
  - Fusion lemmas
  - Extensionality lemmas
  - Identity lemmas

Grand Total

Summary:

Shortcomings:

Grand Total

Summary:

1. Simple languages

Shortcomings:

Grand Total

Summary:

1. Simple languages
2. Problem easy to state

Shortcomings:

Grand Total

Summary:

1. Simple languages
2. Problem easy to state
3. 4 lemmas before we can state the problem

Shortcomings:

Grand Total

Summary:

1. Simple languages
2. Problem easy to state
3. 4 lemmas before we can state the problem
4. 12+ lemmas before we can start proving

Shortcomings:

Grand Total

Summary:

1. Simple languages
2. Problem easy to state
3. 4 lemmas before we can state the problem
4. 12+ lemmas before we can start proving

Shortcomings:

1. Everything is ad-hoc (change the language, re-do the proofs!)

Grand Total

Summary:

1. Simple languages
2. Problem easy to state
3. 4 lemmas before we can state the problem
4. 12+ lemmas before we can start proving

Shortcomings:

1. Everything is ad-hoc (change the language, re-do the proofs!)
2. Quite noisy ($[\![\cdot]\!] \cdot$ is painfully explicit about the structural cases)

Anatomy of a Language's Syntax

$$\mathsf{lam} : \forall\{\sigma\ \tau\ \Gamma\} \to \mathsf{Tm}\ \tau\ (\sigma :: \Gamma) \to \mathsf{Tm}\ (\sigma \Rightarrow \tau)\ \Gamma$$
$$\mathsf{app} : \forall\{\sigma\ \tau\ \Gamma\} \to \mathsf{Tm}\ (\sigma \Rightarrow \tau)\ \Gamma \to \mathsf{Tm}\ \sigma\ \Gamma \to \mathsf{Tm}\ \tau\ \Gamma$$

A constructor needs to be able to:

Anatomy of a Language's Syntax

$$\mathsf{lam} : \forall\ \{\sigma\ \tau\ \Gamma\} \to \mathsf{Tm}\ \tau\ (\sigma :: \Gamma) \to \mathsf{Tm}\ (\sigma \Rightarrow \tau)\ \Gamma$$

$$\mathsf{app} : \forall\ \{\sigma\ \tau\ \Gamma\} \to \mathsf{Tm}\ (\sigma \Rightarrow \tau)\ \Gamma \to \mathsf{Tm}\ \sigma\ \Gamma \to \mathsf{Tm}\ \tau\ \Gamma$$

A constructor needs to be able to:

1. Store values (and the rest of the constructor's telescope may depend on them)

Anatomy of a Language's Syntax

$$\mathsf{lam} : \forall\{\sigma\ \tau\ \Gamma\} \to \mathsf{Tm}\ \tau\ (\sigma :: \Gamma) \to \mathsf{Tm}\ (\sigma \Rightarrow \tau)\ \Gamma$$

$$\mathsf{app} : \forall\{\sigma\ \tau\ \Gamma\} \to \mathsf{Tm}\ (\sigma \Rightarrow \tau)\ \Gamma \to \mathsf{Tm}\ \sigma\ \Gamma \to \mathsf{Tm}\ \tau\ \Gamma$$

A constructor needs to be able to:

1. Store values (and the rest of the constructor's telescope may depend on them)
2. Have recursive substructures

Anatomy of a Language's Syntax

$$\mathsf{lam} : \forall\{\sigma\ \tau\ \Gamma\} \to \mathsf{Tm}\ \tau\ (\sigma :: \Gamma) \to \mathsf{Tm}\ (\sigma \Rightarrow \tau)\ \Gamma$$

$$\mathsf{app} : \forall\{\sigma\ \tau\ \Gamma\} \to \mathsf{Tm}\ (\sigma \Rightarrow \tau)\ \Gamma \to \mathsf{Tm}\ \sigma\ \Gamma \to \mathsf{Tm}\ \tau\ \Gamma$$

A constructor needs to be able to:

1. Store values (and the rest of the constructor's telescope may depend on them)
2. Have recursive substructures (sometimes with extra variables in scope)

Anatomy of a Language's Syntax

$$\mathsf{lam} : \forall\{\sigma\ \tau\ \Gamma\} \to \mathsf{Tm}\ \tau\ (\sigma :: \Gamma) \to \mathsf{Tm}\ (\sigma \Rightarrow \tau)\ \Gamma$$

$$\mathsf{app} : \forall\{\sigma\ \tau\ \Gamma\} \to \mathsf{Tm}\ (\sigma \Rightarrow \tau)\ \Gamma \to \mathsf{Tm}\ \sigma\ \Gamma \to \mathsf{Tm}\ \tau\ \Gamma$$

A constructor needs to be able to:

1. Store values (and the rest of the constructor's telescope may depend on them)
2. Have recursive substructures (sometimes with extra variables in scope)
3. Constraint the shape of the branche's indices

Codes for Syntaxes

Requirements:

1. Store values
2. Have recursive substructures
3. Constraint the shape of the indices

$$\text{data Desc } (I : \text{Set}) : \text{Set}$$

$$[\![\cdot]\!] : \text{Desc } I \to (I \to \text{List } I \to \text{Set}) \to (I \to \text{List } I \to \text{Set})$$

Codes for Syntaxes

Requirements:

1. Store values
2. Have recursive substructures
3. Constraint the shape of the indices

$$`\sigma : (A : \mathsf{Set}) \to (A \to \mathsf{Desc}\ I) \to \mathsf{Desc}\ I$$

$$[\![\ `\sigma\ A\ d\ ]\!]\ X\ i\ \Gamma = \Sigma_{a:A} [\![\ d\ a\ ]\!]\ X\ i\ \Gamma$$

Codes for Syntaxes

Requirements:

1. Store values
2. Have recursive substructures
3. Constraint the shape of the indices

$$\text{`X} : I \to \text{List } I \to \text{Desc } I \to \text{Desc } I$$

$$[\![\, \text{`X}\ j\ \Delta\ d\,]\!]\ X\ i\ \Gamma = X\ j\ (\Delta +\!\!+ \Gamma) \times [\![\, d\,]\!]\ X\ i\ \Gamma$$

Codes for Syntaxes

Requirements:

1. Store values
2. Have recursive substructures
3. Constraint the shape of the indices

$$\text{`X} : I \to \mathsf{List}\ I \to \mathsf{Desc}\ I \to \mathsf{Desc}\ I$$

$$[\![\ \text{`X}\ j\ \Delta\ \ d\ ]\!]\ X\ i\ \Gamma = X\ j\ (\Delta +\!\!+ \Gamma)\ \times [\![\ d\ ]\!]\ X\ i\ \Gamma$$

Codes for Syntaxes

Requirements:

1. Store values
2. Have recursive substructures
3. Constraint the shape of the indices

$$`\blacksquare : \kappa \rightarrow \text{Desc } I$$

$$[\![`\blacksquare\ j]\!]\ X\ i\ \Gamma = i \equiv j$$

Example: Code for STLC

```
data ‘STLC : Set where
    ‘app : (σ τ : Type) → ‘STLC
    ‘lam : (σ τ : Type) → ‘STLC

STLC : Desc Type
STLC = ‘σ ‘STLC $ λ where
    (‘app σ τ) → ‘X (σ ⇒ τ) [] (‘X σ [] (‘■ τ))
    (‘lam σ τ) → ‘X τ (σ :: []) (‘■ (σ⇒τ))
```

Example: Code for STLC

```
data ‘STLC : Set where
    ‘app : (σ τ : Type) → ‘STLC
    ‘lam : (σ τ : Type) → ‘STLC

STLC : Desc Type
STLC = ‘σ ‘STLC $ λ where
    (‘app σ τ) → ‘X (σ ⇒ τ) [] (‘X σ [] (‘■ τ))
    (‘lam σ τ) → ‘X τ (σ :: []) (‘■ (σ⇒τ))
```

Example: Code for STLC

```
data ‘STLC : Set where
    ‘app : (σ τ : Type) → ‘STLC
    ‘lam : (σ τ : Type) → ‘STLC

STLC : Desc Type
STLC = ‘σ ‘STLC $ λ where
    (‘app σ τ) → ‘X (σ ⇒ τ) [] (‘X σ [] (‘■ τ))
    (‘lam σ τ) → ‘X τ (σ :: []) (‘■ (σ⇒τ))
```

Example: Code for STLC

```
data ‘STLC : Set where
  ‘app : (σ τ : Type) → ‘STLC
  ‘lam : (σ τ : Type) → ‘STLC

STLC : Desc Type
STLC = ‘σ ‘STLC $ λ where
  (‘app σ τ) → ‘X (σ ⇒ τ) [] (‘X σ [] (‘■ τ))
  (‘lam σ τ) → ‘X τ (σ :: []) (‘■ (σ⇒τ))
```

Example: Code for STLC

```
data ‘STLC : Set where
  ‘app : (σ τ : Type) → ‘STLC
  ‘lam : (σ τ : Type) → ‘STLC

STLC : Desc Type
STLC = ‘σ ‘STLC $ λ where
  (‘app σ τ) → ‘X (σ ⇒ τ) [] (‘X σ [] (‘■ τ))
  (‘lam σ τ) → ‘X τ (σ :: []) (‘■ (σ⇒τ))
```

Example: Code for STLC

```
data ‘STLC : Set where
  ‘app : (σ τ : Type) → ‘STLC
  ‘lam : (σ τ : Type) → ‘STLC

STLC : Desc Type
STLC = ‘σ ‘STLC $ λ where
  (‘app σ τ) → ‘X (σ ⇒ τ) [] ( ‘X σ [] (‘■ τ))
  (‘lam σ τ) → ‘X τ (σ :: []) (‘■ (σ⇒τ))
```

Example: Code for STLC

```
data ‘STLC : Set where
    ‘app : (σ τ : Type) → ‘STLC
    ‘lam : (σ τ : Type) → ‘STLC

STLC : Desc Type
STLC = ‘σ ‘STLC $ λ where
    (‘app σ τ) → ‘X (σ ⇒ τ) [] (‘X σ [] (‘■ τ))
    (‘lam σ τ) → ‘X τ (σ :: []) (‘■ (σ⇒τ))
```

Example: Code for STLC

```
data ‘STLC : Set where
  ‘app : (σ τ : Type) → ‘STLC
  ‘lam : (σ τ : Type) → ‘STLC

STLC : Desc Type
STLC = ‘σ ‘STLC $ λ where
  (‘app σ τ) → ‘X (σ ⇒ τ) [] (‘X σ [] (‘■ τ))
  (‘lam σ τ) → ‘X τ (σ :: []) (‘■ (σ⇒τ))
```

Example: Code for STLC

```
data ‘STLC : Set where
    ‘app : (σ τ : Type) → ‘STLC
    ‘lam : (σ τ : Type) → ‘STLC

STLC : Desc Type
STLC = ‘σ ‘STLC $ λ where
    (‘app σ τ) → ‘X (σ ⇒ τ) [] (‘X σ [] (‘■ τ))
    (‘lam σ τ) → ‘X τ (σ :: []) (‘■ (σ⇒τ) )
```

Terms as Free Relative Monads

```
data Tm (d : Desc I) (i : I) (Γ : List I) : Set where
  ‘var : Var i Γ → Tm d i Γ
  ‘con : ⟦ d ⟧ (Tm d) i Γ → Tm d i Γ
```

Terms as Free Relative Monads

```
data Tm (d : Desc I) (i : I) (Γ : List I) : Set where
  ‘var : Var i Γ → Tm d i Γ
  ‘con : ⟦ d ⟧ (Tm d) i Γ → Tm d i Γ
```

Terms as Free Relative Monads

```
data Tm (d : Desc I) (i : I) (Γ : List I) : Set where
  'var : Var i Γ → Tm d i Γ
  'con : ⟦ d ⟧ (Tm d) i Γ → Tm d i Γ
```

Terms as Free Relative Monads

```
data Tm (d : Desc I) (i : I) (Γ : List I) : Set where
  'var : Var i Γ → Tm d i Γ
  'con : ⟦ d ⟧ (Tm d) i Γ → Tm d i Γ
```

Terms as Free Relative Monads

data Tm (d : Desc I) (i : I) (Γ : List I) : Set where

 ‘var : Var i $\Gamma \rightarrow$ Tm d i Γ

 ‘con : $[\![\ d\ ]\!]$ (Tm d) i $\Gamma \rightarrow$ Tm d i Γ

$$\text{record Sem } (d : \text{Desc } I)\ (\mathcal{V}\ \mathcal{C} : I \to \text{List } I \to \text{Set}) : \text{Set where}$$

$$\vdots$$

$$\begin{aligned} \text{sem} : \ & \text{Sem } d\ \mathcal{V}\ \mathcal{C} \to \forall\{\Gamma\ \Delta\} \to \\ & (\forall\{i\} \to \text{Var } i\ \Gamma \to \mathcal{V}\ i\ \Delta) \to \\ & \forall\{i\} \to \text{Tm } d\ i\ \Gamma \to \mathcal{C}\ i\ \Delta \end{aligned}$$

record Sem (d : Desc I) ($\mathcal{V}\ \mathcal{C} : I \to$ List $I \to$ Set) : Set where

$$\vdots$$

$$
\begin{aligned}
\text{sem} : \ & \text{Sem}\ d\ \mathcal{V}\ \mathcal{C} \to \forall\{\Gamma\ \Delta\} \to \\
& (\forall\{i\} \to \text{Var}\ i\ \Gamma \to \mathcal{V}\ i\ \Delta) \to \\
& \forall\{i\} \to \text{Tm}\ d\ i\ \Gamma \to \mathcal{C}\ i\ \Delta
\end{aligned}
$$

$$\text{record Sem } (d : \text{Desc } I)\ (\mathcal{V}\ \mathcal{C} : I \to \text{List } I \to \text{Set}) : \text{Set where}$$

$$\vdots$$

$$\begin{aligned}\text{sem} :\ & \text{Sem } d\ \mathcal{V}\ \mathcal{C} \to \forall\{\Gamma\ \Delta\} \to \\ & (\forall\{i\} \to \text{Var } i\ \Gamma \to \mathcal{V}\ i\ \Delta) \to \\ & \forall\{i\} \to \text{Tm } d\ i\ \Gamma \to \mathcal{C}\ i\ \Delta\end{aligned}$$

$$\textsf{record Sem } (d : \textsf{Desc } I)\ (\mathcal{V}\ \mathcal{C} : I \to \textsf{List } I \to \textsf{Set}) : \textsf{Set where}$$

$$\vdots$$

$$\begin{aligned}\textsf{sem} : \ & \textsf{Sem } d\ \mathcal{V}\ \mathcal{C} \to \forall\{\Gamma\ \Delta\} \to \\ & (\forall\{i\} \to \textsf{Var } i\ \Gamma \to \mathcal{V}\ i\ \Delta) \to \\ & \forall\{i\} \to \textsf{Tm } d\ i\ \Gamma \to \mathcal{C}\ i\ \Delta\end{aligned}$$

$$\textsf{record Sem } (d : \textsf{Desc } I)\ (\mathcal{V}\ \mathcal{C} : I \to \textsf{List } I \to \textsf{Set}) : \textsf{Set where}$$

$$\vdots$$

$$\begin{aligned} \textsf{sem} :\ & \textsf{Sem } d\ \mathcal{V}\ \mathcal{C} \to \forall\{\Gamma\ \Delta\} \to \\ & (\forall\{i\} \to \textsf{Var } i\ \Gamma \to \mathcal{V}\ i\ \Delta) \to \\ & \forall\{i\} \to \textsf{Tm } d\ i\ \Gamma \to \mathcal{C}\ i\ \Delta \end{aligned}$$

$$\textsf{record Sem } (d : \textsf{Desc } I)\ (\mathcal{V}\ \mathcal{C} : I \to \textsf{List } I \to \textsf{Set}) : \textsf{Set where}$$

$$\vdots$$

$$\begin{aligned}\textsf{sem} :\ & \textsf{Sem } d\ \mathcal{V}\ \mathcal{C} \to \forall\{\Gamma\ \Delta\} \to \\ & (\forall\{i\} \to \textsf{Var } i\ \Gamma \to \mathcal{V}\ i\ \Delta) \to \\ & \forall\{i\} \to \textsf{Tm } d\ i\ \Gamma \to \mathcal{C}\ i\ \Delta\end{aligned}$$

$$\mathsf{record}\ \mathsf{Sem}\ (d : \mathsf{Desc}\ I)\ (\mathcal{V}\ \mathcal{C} : I \to \mathsf{List}\ I \to \mathsf{Set}) : \mathsf{Set}\ \mathsf{where}$$

$$\vdots$$

$$\begin{aligned}\mathsf{sem} :\ & \mathsf{Sem}\ d\ \mathcal{V}\ \mathcal{C} \to \forall\{\Gamma\ \Delta\} \to \\ & (\forall\{i\} \to \mathsf{Var}\ i\ \Gamma \to \mathcal{V}\ i\ \Delta) \to \\ & \forall\{i\} \to \mathsf{Tm}\ d\ i\ \Gamma \to \mathcal{C}\ i\ \Delta\end{aligned}$$

A Catalogue of Scope-and-Kind Preserving Programs

- Generic:
  - Renaming
  - Substitution
  - Let-elaboration
  - Printing
  - Scope-checking
  - (Unsafe) Normalization by Evaluation
- Specific to a given language:
  - CPS translation
  - Typechecking
  - Elaboration to a typed language
  - (Safe) Normalization by Evaluation

Proof Frameworks

Observations:

- Traversals defined using Sem have a constrained shape
- We should get something for free out of it!

Proof Frameworks

Observations:

- Traversals defined using Sem have a constrained shape
- We should get something for free out of it!

Results:

- Simulation lemma between two Semantics

Proof Frameworks

Observations:

- Traversals defined using Sem have a constrained shape
- We should get something for free out of it!

Results:

- Simulation lemma between two Semantics
- Fusion lemma between three Semantics

Proof Frameworks

Observations:

- Traversals defined using Sem have a constrained shape
- We should get something for free out of it!

Results:

- Simulation lemma between two Semantics
- Fusion lemma between three Semantics
- Instances for common traversals defined generically

Summary

Motivation

- A Program Transformation
- A Soundness Lemma

A Universe of Syntaxes with Binding

- Anatomy of a Language's Syntax
- Codes for Syntaxes

Scope-and-Kind Aware Traversals

- A Generic Notion of Semantics
- A Catalogue of Scope-and-Kind Preserving Programs

Proof Frameworks

Thank you for your attention

You can find all of this (and more) at
`https://github.com/gallais/generic-syntax`

Avenues for future research:

- Which compilation passes can be implemented generically?
- Which syntaxes can be safely normalized?
- Can we have a theory of refinement between various syntaxes?
- Can we define a subset of well-behaved typing judgments for syntaxes?