Syntax-Generic Operations, Reflectively Reified

Extended Abstract

Tzu-Chi Lin and Josh Ko

September 11, 2022

Institute of Information Science
Academia Sinica, Taiwan

Typical Language in Language

Intrinsic typing is common for λ-calculus with De Bruijn indices.

```
data _⊢_ : Context → Ty → Set where
  `_  : Γ ∋ A → Γ ⊢ A
  ƛ_  : Γ , A ⊢ B → Γ ⊢ A ⇒ B
  _·_ : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B
```

```
data Ty : Set where
  α    : Ty
  _⇒_ : Ty → Ty → Ty

data Context (A : Set) : Set where
  ∅    : Context A
  _,_  : (Γ : Context A) → A → Context A

data _∋_ : Context → Ty → Set where
  z : Γ , A ∋ A
  s : Γ ∋ A → Γ , B ∋ A
```

Scope-safe Operations

$$
\begin{aligned}
&\mathit{rename} : \forall\ \{\Gamma\ \Delta\} \to (\forall\ \{A\} \to \Gamma \ni A \to \Delta \ni A) \\
&\qquad\qquad\qquad\quad \to (\forall\ \{A\} \to \Gamma \vdash A \to \Delta \vdash A) \\
&\mathit{rename}\ \rho\ (\text{`}\ x) \quad = \text{`}\ (\rho\ x) \\
&\mathit{rename}\ \rho\ (\lambda\ N) \quad = \lambda\ (\mathit{rename}\ (\mathit{ext}\ \rho)\ N) \\
&\mathit{rename}\ \rho\ (L \cdot M) \quad = (\mathit{rename}\ \rho\ L) \cdot (\mathit{rename}\ \rho\ M)
\end{aligned}
$$

Repetition

Intrinsic typing is common for λ-calculus with De Bruijn indices.

```
data Ty : Set where
  ...
  `ℕ : Ty
data _⊢_ : Context → Ty → Set where
  ...
  `zero : Γ ⊢ `ℕ
  `suc_ : Γ ⊢ `ℕ → Γ ⊢ `ℕ
  case  : Γ ⊢ `ℕ → Γ ⊢ A → Γ , `ℕ ⊢ A → Γ ⊢ A
  μ_    : Γ , A ⊢ A → Γ ⊢ A
```

Repetition

$$
\begin{aligned}
&\mathit{rename} : \forall\ \{\Gamma\ \Delta\} \to (\forall\ \{A\} \to \Gamma \ni A \to \Delta \ni A) \\
&\qquad\qquad\qquad\qquad \to (\forall\ \{A\} \to \Gamma \vdash A \to \Delta \vdash A) \\
&\ldots \\
&\mathit{rename}\ \rho\ (\text{`}\mathit{zero}) \qquad\quad = \text{`}\mathit{zero} \\
&\mathit{rename}\ \rho\ (\text{`}\mathit{suc}\ M) \qquad = \text{`}\mathit{suc}\ (\mathit{rename}\ \rho\ M) \\
&\mathit{rename}\ \rho\ (\mathit{case}\ L\ M\ N) \;=\; \mathit{case}\ (\mathit{rename}\ \rho\ L) \\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathit{rename}\ \rho\ M) \\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathit{rename}\ (\mathit{ext}\ \rho)\ N) \\
&\mathit{rename}\ \rho\ (\mu\ N) \qquad\quad = \mu\ (\mathit{rename}\ (\mathit{ext}\ \rho)\ N)
\end{aligned}
$$

Repetition

Change/extend the object language even further:

$$
\begin{array}{l}
\textbf{data}\ \_\vdash\_ : \mathit{Context} \to \mathit{Ty} \to \mathit{Set}\ \textbf{where} \\
\quad \ldots \\
\quad \mathit{con} \quad\quad : \mathbb{N} \to \Gamma \vdash \mathit{Nat} \\
\quad \_`*\_ \quad\quad : \Gamma \vdash \mathit{Nat} \to \Gamma \vdash \mathit{Nat} \to \Gamma \vdash \mathit{Nat} \\
\quad `\mathit{let} \quad\quad : \Gamma \vdash A \to \Gamma\ ,\ A \vdash B \to \Gamma \vdash B \\
\quad `<\_\ ,\ \_> : \Gamma \vdash A \to \Gamma \vdash B \to \Gamma \vdash A\ `\times\ B \\
\quad `\mathit{proj}_1 \quad : \Gamma \vdash A\ `\times\ B \to \Gamma \vdash A \\
\quad `\mathit{proj}_2 \quad : \Gamma \vdash A\ `\times\ B \to \Gamma \vdash B \\
\quad \mathit{case}\times \quad : \Gamma \vdash A\ `\times\ B \to \Gamma\ ,\ A\ ,\ B \vdash C \to \Gamma \vdash C
\end{array}
$$

Repetition

Redefine/extend syntax operations:

$$
\begin{aligned}
&\ldots \\
&\mathit{rename}\ \rho\ (\mathit{con}\ n) &&= \mathit{con}\ n \\
&\mathit{rename}\ \rho\ (M \mathbin{`*} N) &&= \mathit{rename}\ \rho\ M \mathbin{`*} \mathit{rename}\ \rho\ N \\
&\mathit{rename}\ \rho\ (\mathit{`let}\ M\ N) &&= \mathit{`let}\ (\mathit{rename}\ \rho\ M)\ (\mathit{rename}\ (\mathit{ext}\ \rho)\ N) \\
&\mathit{rename}\ \rho\ \mathbin{`<} M\ ,\ N > &&= \mathbin{`<} \mathit{rename}\ \rho\ M\ ,\ \mathit{rename}\ \rho\ N > \\
&\mathit{rename}\ \rho\ (\mathit{`proj}_1\ L) &&= \mathit{`proj}_1\ (\mathit{rename}\ \rho\ L) \\
&\mathit{rename}\ \rho\ (\mathit{`proj}_2\ L) &&= \mathit{`proj}_2\ (\mathit{rename}\ \rho\ L) \\
&\mathit{rename}\ \rho\ (\mathit{case}\times\ L\ M) &&= \mathit{case}\times\ (\mathit{rename}\ \rho\ L) \\
&\qquad\qquad\qquad\qquad\qquad\qquad && \quad (\mathit{rename}\ (\mathit{ext}\ (\mathit{ext}\ \rho))\ M)
\end{aligned}
$$

Repetition

Other repeating operations:

$$\begin{aligned} \mathit{subst} \;:\; & \forall\, \{\Gamma\ \Delta\} \to (\forall\, \{A\} \to \Gamma \ni A \to \Delta \vdash A) \\ & \to (\forall\, \{A\} \to \Gamma \vdash A \to \Delta \vdash A) \end{aligned}$$

$$\mathit{print} \;:\; \Gamma \vdash A \to \mathit{String}$$

$$\ldots$$

Where we are going...

1. Existing Work for Syntax-generic Operations
2. Elaborator Reflection to the Rescue
3. Discussion

Existing Work for Syntax-generic Operations

Existing work

There are generic libraries for a family/families of syntaxes with binders.

We base our work on Allais et al.'s approach presented at ICFP '18 (later published in JFP '21).

Existing work by Allais et al.

Allais et al.'s *Desc*:

```
data Desc (I : Set) : Set₁ where
  `σ : (A : Set) → (A → Desc I) → Desc I
  `X : List I → I → Desc I → Desc I
  `■ : I → Desc I
```

```
data `STLC : Set where
  App Lam : Ty → Ty → `STLC
STLCD : Desc Ty
STLCD = `σ `STLC λ where
  (App i j) →
    `X [] (i ⇒ j) (`X [] i (■ j))
  (Lam i j) →
    `X (i :: []) j (■ (i ⇒ j))
```

Existing work by Allais et al.

Simply typed λ-calculus

$$
\begin{aligned}
&\textbf{data}\ \_\vdash\_ : \mathit{Context} \to \mathit{Ty} \to \mathit{Set}\ \textbf{where}\\
&\quad \text{`}\_ \ : \Gamma \ni A \to \Gamma \vdash A\\
&\quad \lambda\!\!\!\lambda\_ \ : \Gamma\ , A \vdash B \to \Gamma \vdash A \Rightarrow B\\
&\quad \_\cdot\_ \ : \Gamma \vdash A \Rightarrow B \to \Gamma \vdash A \to \Gamma \vdash B
\end{aligned}
$$

Existing work by Allais et al.

Simply typed λ-calculus

```
data _⊢_ : Context → Ty → Set where
  `_  : Γ ∋ A → Γ ⊢ A
  ƛ_  : Γ , A ⊢ B → Γ ⊢ A ⇒ B
  _·_ : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B
```

encoded in *Desc*:

```
STLCD : Desc Ty
STLCD = `σ `STLC λ where
  (App i j) →
    `X [] (i ⇒ j) (`X [] i (■ j))
  (Lam i j) →
    `X (i :: []) j (■ (i ⇒ j))
```

Existing work by Allais et al.

Simply typed λ-calculus

```
data _⊢_ : Context → Ty → Set where
  `_  : Γ ∋ A → Γ ⊢ A
  ƛ_  : Γ , A ⊢ B → Γ ⊢ A ⇒ B
  _·_ : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B
```

encoded in *Desc*:

```
STLCD : Desc Ty
STLCD = `σ `STLC λ where
  (App i j) →
    `X [] (i ⇒ j) (`X [] i (■ j))
  (Lam i j) →
    `X (i :: []) j (■ (i ⇒ j))

STLC′ = Tm STLCD
```

Existing work by Allais et al.

Simply typed λ-calculus

```
data _⊢_ : Context → Ty → Set where
  `_  : Γ ∋ A → Γ ⊢ A
  ƛ_  : Γ , A ⊢ B → Γ ⊢ A ⇒ B
  _·_ : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B
```

encoded in *Desc*:

```
STLCD : Desc Ty
STLCD = `σ `STLC λ where
  (App i j) →
    `X [] (i ⇒ j) (`X [] i (■ j))
  (Lam i j) →
    `X (i :: []) j (■ (i ⇒ j))
```

```
STLC′ = Tm STLCD
```

$$\_\vdash\_ \cong STLC'$$

Generic Functions for the Whole Universe

$$
\begin{aligned}
&\mathit{Var}\ \sigma\ \Gamma\ =\ \Gamma \ni \sigma \\
&\mathit{Renaming}\ :\ \forall\ \{d\ :\ \mathit{Desc}\ I\} \to \mathit{Semantics}\ d\ \mathit{Var}\ (\mathit{Tm}\ d) \\
&\mathit{rename}\ :\ \forall\ \{d\ :\ \mathit{Desc}\ I\} \to (\forall\ \{\sigma\} \to \mathit{Var}\ \sigma\ \Gamma\ \ \to \mathit{Var}\ \sigma\ \Delta) \\
&\quad \forall\ \{\tau\} \qquad\qquad\qquad\qquad\qquad\qquad \to \mathit{Tm}\ d\ \tau\ \Gamma \to \mathit{Tm}\ d\ \tau\ \Delta \\
&\mathit{rename}\ =\ \mathit{semantics}\ \mathit{Renaming}
\end{aligned}
$$

Generic Functions for the Whole Universe

$$
\begin{aligned}
&\mathit{Var}\ \sigma\ \Gamma\ =\ \Gamma \ni \sigma \\
&\mathit{Renaming}\ :\ \forall\ \{d\ :\ \mathit{Desc}\ I\} \to \mathit{Semantics}\ d\ \mathit{Var}\ (\mathit{Tm}\ d) \\
&\mathit{rename}\ :\ \forall\ \{d\ :\ \mathit{Desc}\ I\} \to (\forall\ \{\sigma\} \to \mathit{Var}\ \sigma\ \Gamma\ \ \to \mathit{Var}\ \sigma\ \Delta) \\
&\quad \forall\ \{\tau\} \qquad\qquad\qquad\qquad\qquad\qquad \to \mathit{Tm}\ d\ \tau\ \Gamma \to \mathit{Tm}\ d\ \tau\ \Delta \\
&\mathit{rename}\ =\ \mathit{semantics}\ \mathit{Renaming}
\end{aligned}
$$

Generic programs are *Semantics* records.

Generic Functions for the Whole Universe

$$
\begin{aligned}
&\mathit{Var}\ \sigma\ \Gamma\ =\ \Gamma \ni \sigma \\
&\mathit{Renaming}\ :\ \forall\ \{d\ :\ \mathit{Desc}\ I\} \to \mathit{Semantics}\ d\ \mathit{Var}\ (\mathit{Tm}\ d) \\
&\mathit{rename}\ :\ \forall\ \{d\ :\ \mathit{Desc}\ I\} \to (\forall\ \{\sigma\} \to \mathit{Var}\ \sigma\ \Gamma\ \ \to \mathit{Var}\ \sigma\ \Delta) \\
&\quad \forall\ \{\tau\} \qquad\qquad\qquad\qquad\qquad\qquad \to \mathit{Tm}\ d\ \tau\ \Gamma \to \mathit{Tm}\ d\ \tau\ \Delta \\
&\mathit{rename}\ =\ \mathit{semantics}\ \mathit{Renaming}
\end{aligned}
$$

Generic programs are *Semantics* records.

Functions are realized on fixpoints *Tm* via *semantics*.

Generic Functions for the Whole Universe

$$
\begin{aligned}
&\mathit{Var}\ \sigma\ \Gamma\ =\ \Gamma \ni \sigma \\
&\mathit{Renaming}\ :\ \forall\ \{d\ :\ \mathit{Desc}\ I\} \to \mathit{Semantics}\ d\ \mathit{Var}\ (\mathit{Tm}\ d) \\
&\mathit{rename}\ :\ \forall\ \{d\ :\ \mathit{Desc}\ I\} \to (\forall\ \{\sigma\} \to \mathit{Var}\ \sigma\ \Gamma\ \ \to \mathit{Var}\ \sigma\ \Delta) \\
&\quad \forall\ \{\tau\} \qquad\qquad\qquad\qquad\qquad\qquad \to \mathit{Tm}\ d\ \tau\ \Gamma \to \mathit{Tm}\ d\ \tau\ \Delta \\
&\mathit{rename}\ =\ \mathit{semantics}\ \mathit{Renaming}
\end{aligned}
$$

Generic programs are *Semantics* records.

Functions are realized on fixpoints *Tm* via *semantics*.

rename can be applied to fixpoints of any description (e.g. *Tm STLCD*).

Elaborator Reflection to the Rescue

We Want Native Definitions!

“Datatype-Generic Programming Meets Elaborator Reflection” by Josh Ko, Liang-Ting Chen, and Tzu-Chi Lin at 15:50, Tuesday.

We Want Native Definitions!

“Datatype-Generic Programming Meets Elaborator Reflection” by Josh Ko, Liang-Ting Chen, and Tzu-Chi Lin at 15:50, Tuesday.

Data-generic programs are functions that work for a family of datatypes.

We Want Native Definitions!

“Datatype-Generic Programming Meets Elaborator Reflection” by Josh Ko, Liang-Ting Chen, and Tzu-Chi Lin at 15:50, Tuesday.

Data-generic programs are functions that work for a family of datatypes.

Syntax-generic operations *are* Datatype-generic programs with constraints.

The process - Incomplete

1. The programmer defines a native datatype T.

The process - Incomplete

1. The programmer defines a native datatype T.
2. A metaprogram generates the description D of T.

The process - Incomplete

1. The programmer defines a native datatype T.
2. A metaprogram generates the description D of T.
3. The programmer chooses a description P from a set of pre-defined generic programs.

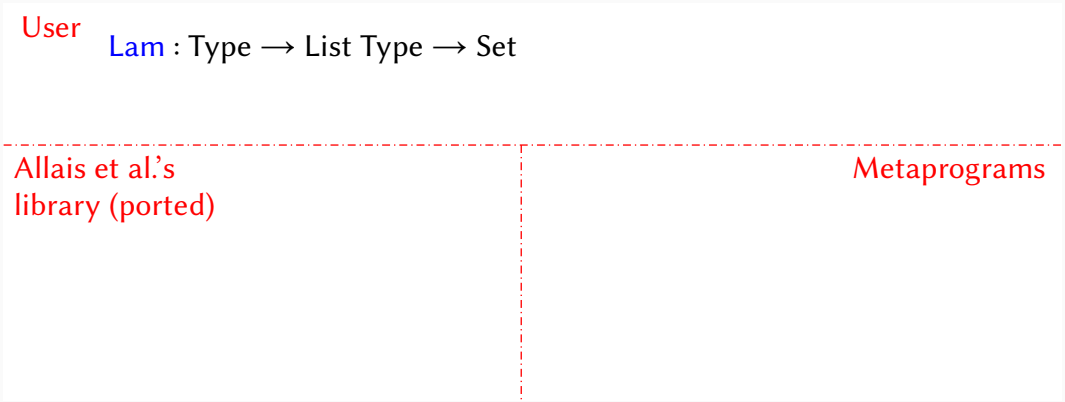
The process - Incomplete

1. The programmer defines a native datatype T.
2. A metaprogram generates the description D of T.
3. The programmer chooses a description P from a set of pre-defined generic programs.
4. A metaprogram takes D and P, generates a native function accordingly.

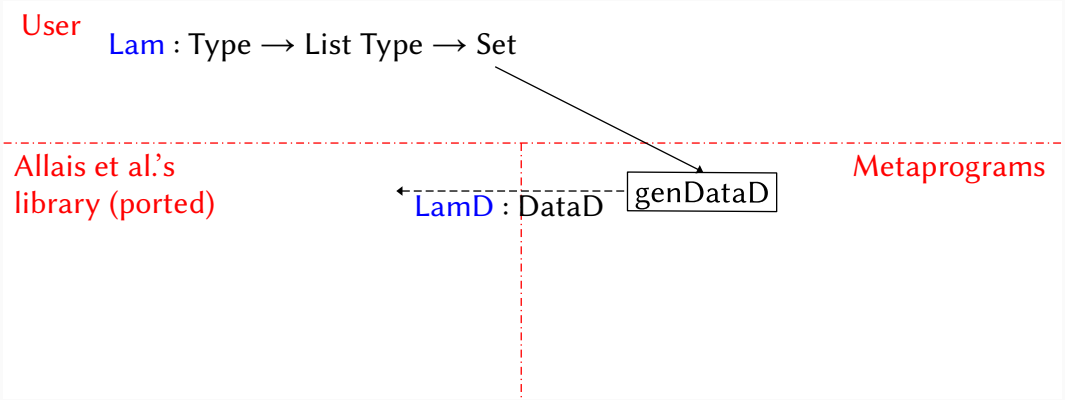
The process - Complete

1. The programmer defines a native datatype T.
2. A metaprogram generates the description D of T.
3. The programmer provides a proof S of D that says T is indeed a syntax.
4. The programmer chooses a description P from a set of pre-defined generic programs.
5. A metaprogram takes D, S, and P, generates a native function accordingly.

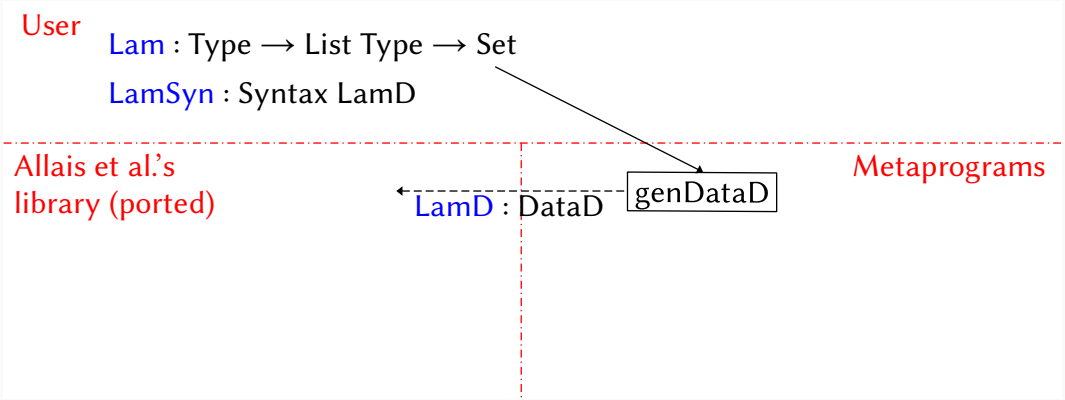
Flow Chart


User
Lam : Type ⟶ List Type ⟶ Set
Allais et al.'s
library (ported)
Metaprograms

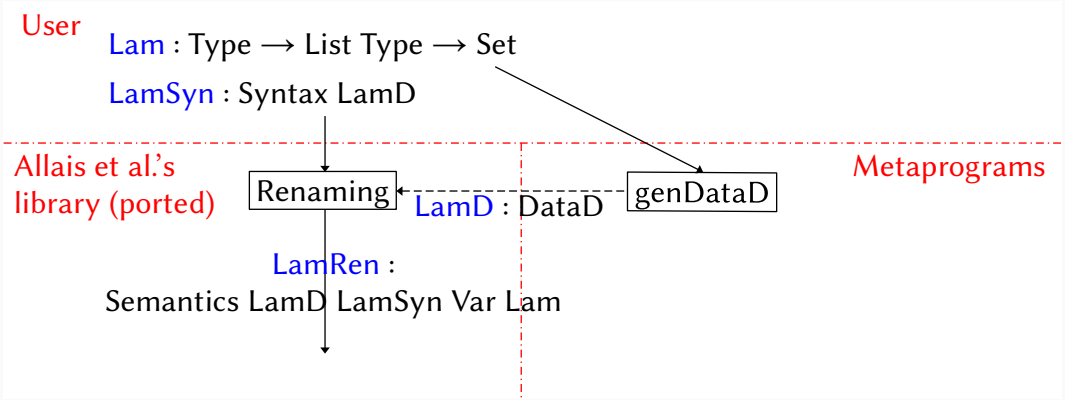
Flow Chart


User
Lam : Type ⟶ List Type ⟶ Set
Allais et al.'s
library (ported)
LamD : DataD
genDataD
Metaprograms

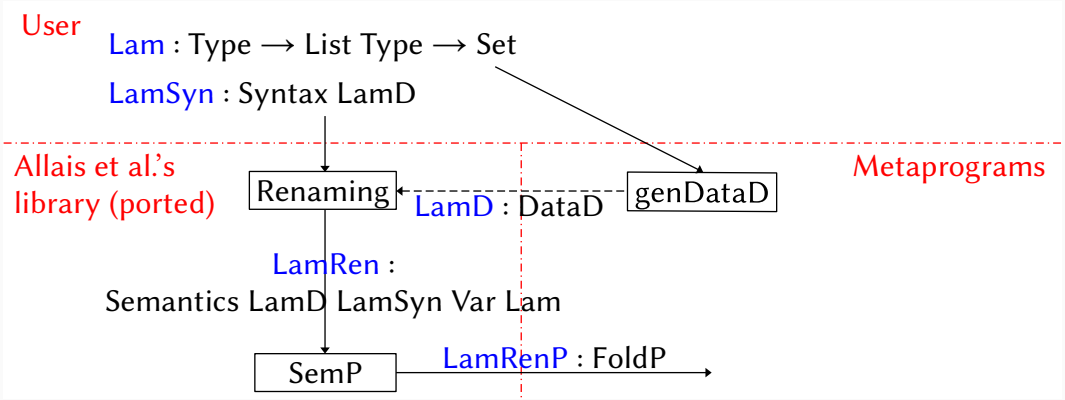
Flow Chart


User
Lam : Type ⟶ List Type ⟶ Set
LamSyn : Syntax LamD
Allais et al.'s
library (ported)
LamD : DataD
genDataD
Metaprograms

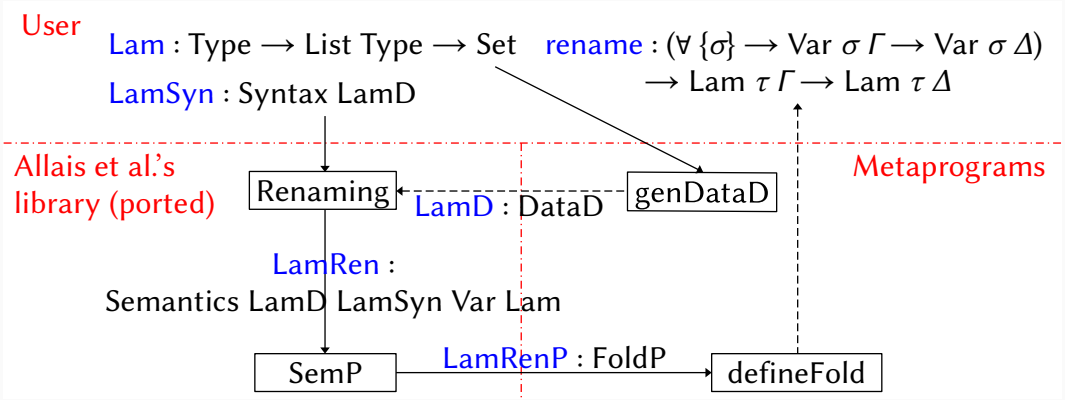
Flow Chart


User
Lam : Type ⟶ List Type ⟶ Set
LamSyn : Syntax LamD
Allais et al.'s
library (ported)
Renaming
LamD : DataD
genDataD
Metaprograms
LamRen :
Semantics LamD LamSyn Var Lam

Flow Chart


User
Lam : Type ⟶ List Type ⟶ Set
LamSyn : Syntax LamD
Allais et al.'s
library (ported)
Metaprograms
Renaming
LamD : DataD
genDataD
LamRen :
Semantics LamD LamSyn Var Lam
SemP
LamRenP : FoldP

Flow Chart


User
Lam : Type ⟶ List Type ⟶ Set
rename : (∀ {σ} ⟶ Var σ Γ ⟶ Var σ Δ)
⟶ Lam τ Γ ⟶ Lam τ Δ
LamSyn : Syntax LamD
Allais et al.'s
library (ported)
Metaprograms
Renaming
LamD : DataD
genDataD
LamRen :
Semantics LamD LamSyn Var Lam
SemP
LamRenP : FoldP
defineFold

Some of Our Contributions

In “Datatype-generic Programming Meets Elaborator Reflection”:

- *DataD*
- *FoldP* for folds (and *IndP* for inductions)
- metaprogram *genDataD*
- metaprogram *defineFold*

Some of Our Contributions

In “Datatype-generic Programming Meets Elaborator Reflection”:

- *DataD*
- *FoldP* for folds (and *IndP* for inductions)
- metaprogram *genDataD*
- metaprogram *defineFold*

In this work:

- predicate *Syntax* on *DataD* that captures *Desc*.
- function *SemP* that generates *FoldP* from *Syntax* proofs.

The *Syntax* Predicate

$$Syntax \ : \ Set \ \ell \to DataD \to Set\omega$$

Desc are captured by *Syntax* as each:

The *Syntax* Predicate

$$Syntax \ : \ Set \ \ell \to DataD \to Set\omega$$

Desc are captured by *Syntax* as each:

- has a variable rule,

THE *Syntax* PREDICATE

$$Syntax \ : \ Set \ \ell \to DataD \to Set\omega$$

Desc are captured by *Syntax* as each:

- has a variable rule,
- is not universe polymorphic,

The *Syntax* Predicate

$$Syntax \ : \ Set \ \ell \to DataD \to Set\omega$$

Desc are captured by *Syntax* as each:

- has a variable rule,
- is not universe polymorphic,
- has two indices, I and $List\ I$, and

The *Syntax* Predicate

$$Syntax \ : \ Set \ \ell \to DataD \to Set\omega$$

Desc are captured by *Syntax* as each:

- has a variable rule,
- is not universe polymorphic,
- has two indices, I and $List\ I$, and
- supports context extensions.

The *Syntax* Predicate

Does *PCF* satisfies *Syntax*?

$$
\begin{aligned}
&\textbf{data}\ \mathit{PCF}\ :\ \mathit{Ty} \to \mathit{Context} \to \mathit{Set}\ \textbf{where}\\
&\quad \text{`}\mathit{var}\ \ :\ \mathit{Var}\ \sigma\ \Gamma \to \mathit{PCF}\ \sigma\ \Gamma\\
&\quad \text{`}\mathit{app}\ \ :\ \mathit{PCF}\ (\sigma \Rightarrow \tau)\ \Gamma \to \mathit{PCF}\ \sigma\ \Gamma \to \mathit{PCF}\ \tau\ \Gamma\\
&\quad \text{`}\mathit{lam}\ \ :\ \mathit{PCF}\ \tau\ (\sigma :: \Gamma) \to \mathit{PCF}\ (\sigma \Rightarrow \tau)\ \Gamma\\
&\quad \text{`}\mathit{zero}\ :\ \mathit{PCF}\ \text{`}\mathbb{N}\ \Gamma\\
&\quad \text{`}\mathit{suc}\_\ :\ \mathit{PCF}\ \text{`}\mathbb{N}\ \Gamma \to \mathit{PCF}\ \text{`}\mathbb{N}\ \Gamma
\end{aligned}
$$

The *Syntax* Predicate

Does *PCF* satisfies *Syntax*?

```
data PCF : Ty → Context → Set where
  `var  : Var σ Γ → PCF σ Γ
  `app  : PCF (σ ⇒ τ) Γ → PCF σ Γ → PCF τ Γ
  `lam  : PCF τ (σ :: Γ) → PCF (σ ⇒ τ) Γ
  `zero : PCF `ℕ Γ
  `suc_ : PCF `ℕ Γ → PCF `ℕ Γ
```

Proof of PCF being Syntax:

```
SyntaxPCF : Syntax Ty (genDataD PCF)
SyntaxPCF = _
  , refl
  , (refl , refl)
  , _
  , refl
  , refl
  , (_ , _ , _ , refl , (λ _ → refl))
  , (_ , _ , _ , refl , (λ _ → refl))
  , (_ , _ , _ , refl , (λ _ → refl))
  , (_ , _ , _ , refl , (λ _ → refl))
  , tt
```

Discussion

Problems with Syntax Universes: Readability

```
data _⊢_ : Context → Ty → Set where
  `_  : Γ ∋ A → Γ ⊢ A
  ƛ_  : Γ , A ⊢ B → Γ ⊢ A ⇒ B
  _·_ : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B
```

```
STLCD : Desc Ty
STLCD = `σ `STLC λ where
  (App i j) →
    `X [] (i ⇒ j) (`X [] i (■ j))
  (Lam i j) →
    `X (i :: []) j (■ (i ⇒ j))
STLC′ = Tm STLCD
```

Problems with Syntax Universes: Burden on Programmers

$$
\begin{aligned}
&\textbf{data}\ \mathit{Desc}\ (I\ :\ \mathit{Set})\ :\ \mathit{Set}_1\ \textbf{where}\\
&\quad \text{`}\sigma\ :\ (A\ :\ \mathit{Set}) \to (A \to \mathit{Desc}\ I) \to \mathit{Desc}\ I\\
&\quad \text{`}X\ :\ \mathit{List}\ I \to I \to \mathit{Desc}\ I \to \mathit{Desc}\ I\\
&\quad \text{`}\blacksquare\ :\ I \to \mathit{Desc}\ I
\end{aligned}
$$

Problems with Syntax Universes: Interoperability

$STLCD : Desc\ Ty$
$STLCD = \ldots$

$STLCD' : Desc'\ ???$
$STLCD' = ???$

Towards Datatype-generic Libraries for Syntaxes?

- Do we really need syntax-generic libraries?