Why Is It Still Hard to Formalise Metatheory?

Robin Adams
University of Bergen
robin.adams.78@gmail.com

27 April 2017

Introduction

My Motivation

I wanted to formalise a result about PHOML:

My Motivation

I wanted to formalise a result about PHOML:

- Three classes of expression:
  - Types $A, B, \ldots$
  - Terms $M, N, \ldots$
  - Paths $P, Q, \ldots$

My Motivation

I wanted to formalise a result about PHOML:

- Three classes of expression:
  - Types $A, B, \ldots$
  - Terms $M, N, \ldots$
  - Paths $P, Q, \ldots$
- Judgement forms $\Gamma \vdash M : A, \Gamma \vdash P : M =_A N$

My Motivation

I wanted to formalise a result about PHOML:

- Three classes of expression:
  - Types $A, B, \ldots$
  - Terms $M, N, \ldots$
  - Paths $P, Q, \ldots$
- Judgement forms $\Gamma \vdash M : A, \Gamma \vdash P : M =_A N$
- An operation of *path substitution*:

$$\frac{\Gamma, x : A \vdash N : B \qquad \Gamma \vdash P : M =_A M'}{\Gamma \vdash N\{x := Q : M = M'\} : N[x := M] =_B N[x := M']}$$

defined by induction on N

My Motivation

I wanted to formalise a result about PHOML:

- Three classes of expression:
  - Types $A, B, \ldots$
  - Terms $M, N, \ldots$
  - Paths $P, Q, \ldots$
- Judgement forms $\Gamma \vdash M : A, \Gamma \vdash P : M =_A N$
- An operation of *path substitution*:

$$\frac{\Gamma, x : A \vdash N : B \qquad \Gamma \vdash P : M =_A M'}{\Gamma \vdash N\{x := Q : M = M'\} : N[x := M] =_B N[x := M']}$$

  defined by induction on N
- Lemmas proved by induction on expressions

My Motivation

I wanted to formalise a result about PHOML:

- Three classes of expression:
  - Types $A, B, \ldots$
  - Terms $M, N, \ldots$
  - Paths $P, Q, \ldots$
- Judgement forms $\Gamma \vdash M : A, \Gamma \vdash P : M =_A N$
- An operation of *path substitution*:

$$\frac{\Gamma, x : A \vdash N : B \qquad \Gamma \vdash P : M =_A M'}{\Gamma \vdash N\{x := Q : M = M'\} : N[x := M] =_B N[x := M']}$$

  defined by induction on N
- Lemmas proved by induction on expressions
- Lemmas proved by induction on derivations

Syntax with Variables

Many formalizations use named variables, de Bruijn indices, or both:

- [Pol94] — two types of bound variables and free variables
- [MM04] — a type of free variables, de Bruijn indices for the bound variables
- [Ada06] — de Bruijn indices for both
- [ACP+08] — cofinite quantification over free variables, de Bruijn indices for bound variables

All begin by definining an inductive datatype, e.g.

$$\frac{x : V}{x : \mathsf{Term}\, V} \qquad \frac{M : \mathsf{Term}\, V \qquad N : \mathsf{Term}\, V}{\mathsf{app}\, M\, N : \mathsf{Term}\, V} \qquad \frac{M : \mathsf{Term}\, (V + 1)}{\lambda M : \mathsf{Term}\, V}$$

If I wish to work with a different language, I must:

- Change definition of Term
- Change all proofs by induction over Term
- Not much less work than starting from scratch

Higher-Order Abstract Syntax

Introduced in [PE88].

Hijack the abstraction mechanism from the metalanguage:

$$\frac{M : \mathsf{Term} \quad N : \mathsf{Term}}{\mathsf{app}\ M\ N : \mathsf{Term}} \qquad \frac{M : \mathsf{Term} \to \mathsf{Term}}{\Lambda M : \mathsf{Term}}$$

We represent $\lambda x : A.M$ by $\Lambda(\lambda x : A.M)$.
We represent $M[x := N]$ by $(\lambda x : A.M)N$ which is definitionally equal to $M[x := N]$.
We represent $M\{x := P : N = N'\}$ by ...?

- No need to worry about freshness, renumbering de Bruijn indices, etc.
- Properties such as Substitution Lemma hold up to definitional equality.
- Cannot define new operations that do not exist in the metalanguage.

Cannot define path substitution

Nominal Sets or Nominal Syntax

Example: [CATS^{+}16]

Type of variables (atoms), and type of permutations on atoms.

Define relation of α-conversion on terms.

- Closer to what we do on paper
- Need to prove everything respects α-conversion

Would need to start from scratch.

Kipling

Introduced in [McB10].

Use semantics on the right of the colon.

Define $\Gamma \vdash M : A$ where $A : [\![\Gamma]\!] \to \mathsf{Set}$.

$$\frac{\Gamma, A \vdash M : B}{\Gamma \vdash \Lambda M : \lambda\gamma : [\![\Gamma]\!].\Pi x : A\gamma.B(\gamma, a)}$$

If $M = N$ is derivable, then $([\![M]\!]$ and $[\![N]\!]$ are definitionally equal in the metatheory(!)

Can only represent features that already exist in the metalanguage.

Why Is It Still So Hard?

The following are *coupled*:

- the *implementation* of syntax with binding
- the *interface* to syntax with binding
- the syntax of the object theory

Why Is It Still So Hard To Sort A List?

- Data types:
  - arrays, singly-linked lists, doubly-linked lists, heaps, trees, ...
- Algorithms:
  - bubble sort, quicksort, merge sort, ...
- Theory:
  - free monoid, monads, commutative monads, ...

There is no best way.

Separation of Concerns

A list is a complicated thing.

Separation of Concerns

A list is a complicated thing.

Separation of concerns:

- Implementation details (array, linked list, etc.)
- API
- Standard library

Separation of Concerns

A list is a complicated thing.

Separation of concerns:

- Implementation details (array, linked list, etc.)
- API
- Standard library

Syntax-with-binding, α-conversion, capture-avoiding substitution is a complicated thing.

- The λ-calculus is Turing complete.

Separation of Concerns

A list is a complicated thing.

Separation of concerns:

- Implementation details (array, linked list, etc.)
- API
- Standard library

Syntax-with-binding, α-conversion, capture-avoiding substitution is a complicated thing.

- The λ-calculus is Turing complete.

Let us separate concerns here.

MetaL

Design Criteria

There should be:

- a type Grammar
- a datatype Expression : Grammar $\to \cdots \to$ Set

- The definition of an object of type Grammar should be readable (look like the syntax on paper)
- It should be possible to define functions and prove theorems by induction over the objects of type Expression $G \cdots$
- It should be possible to prove theorems by induction on derivations

Running Example

The syntax of the simply-typed lambda calculus:

$$\begin{array}{lccl} \text{Type} & A & ::= & * \mid A \to A \\ \text{Term} & M & ::= & x \mid \lambda x : A.M \mid M M \end{array}$$

Taxonomy

A *taxonomy* consists of a set of *expression kinds*, divided into *variable kinds* and *non-variable kinds*.

```
record Taxonomy : Set₁ where
  field
    VariableKind : Set
    NonVariableKind : Set

  data ExpressionKind : Set where
    varKind : VariableKind → ExpressionKind
    nonVariableKind : NonVariableKind → ExpressionKind
```

Taxonomy — Example


```
data stlcVariableKind : Set where
  -term : stlcVariableKind

data stlcNonVariableKind : Set where
  -type : stlcNonVariableKind

stlcTaxonomy : Taxonomy
stlcTaxonomy = record {
  VariableKind = stlcVariableKind ;
  NonVariableKind = stlcNonVariableKind }
```

Alphabets

An *alphabet* V is a finite set of *variables*, each with a variable kind K.
We write Var $V\ K$ for the set of all variables in V of kind K.

```
infixl 55 _,_
data Alphabet : Set where
  ∅ : Alphabet
  _,_ : Alphabet → VarKind → Alphabet

data Var : Alphabet → VarKind → Set where
  x₀ : ∀ {V} {K} → Var (V , K) K
  ↑ : ∀ {V} {K} {L} → Var V L → Var (V , K) L
```

Grammar

A grammar is given by a set of *constructors*, each with a *constructor kind* that shows what arguments it takes, and which variables are bound.

In our example, there will be four constructors:

$$
\begin{array}{rcl}
* & : & \mathbf{Type} \\
\rightarrow & : & \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \mathbf{Type} \\
\Lambda & : & \mathbf{Type} \rightarrow (\mathbf{Term} \rightarrow \mathbf{Term}) \rightarrow \mathbf{Term} \\
\mathsf{app} & : & \mathbf{Term} \rightarrow \mathbf{Term} \rightarrow \mathbf{Term}
\end{array}
$$

Constructor Kinds

An expression like $(\mathbf{Term} \to \mathbf{Term})$ is an *abstraction kind*. An expression like $(\mathbf{Term} \to \mathbf{Term}) \to \mathbf{Term}$ is a *constructor kind*.

```
record SimpleKind (A B : Set) : Set where
  constructor SK
  field
    dom : List A
    cod : B

infix 71 _◊
_◊ : ∀ {A} {B} → B → SimpleKind A B
b ◊ = SK [] b

infixr 70 _⟶_
_⟶_ : ∀ {A} {B} → A → SimpleKind A B →
  SimpleKind A B
a ⟶ SK dom cod = SK (a :: dom) cod
```

Constructor Kinds

AbstractionKind = SimpleKind VariableKind ExpressionKind

ConstructorKind = SimpleKind AbstractionKind ExpressionKind

Grammar

A *grammar* is a set of *constructors*, each with a constructor kind.

```
record IsGrammar (T : Taxonomy) : Set₁ where
  open Taxonomy T
  field
    Constructor  : ConstructorKind → Set
    parent : VariableKind → ExpressionKind

record Grammar : Set₁ where
  field
    taxonomy : Taxonomy
    isGrammar : IsGrammar taxonomy
  open Taxonomy taxonomy public
  open IsGrammar isGrammar public
```

Grammar — Example

Example:

data stlcCon : ConstructorKind → Set where
  -bot : stlcCon (type ◊)
  -arrow : stlcCon (type ◊ ⟶ type ◊ ⟶ type ◊)
  -app : stlcCon (term ◊ ⟶ term ◊ ⟶ term ◊)
  -lam : stlcCon (type ◊ ⟶ (-term ⟶ term ◊) ⟶
    term ◊)

Expressions

Define simultaneously:

- If x is a variable of kind K in V then x is an expression of kind K over V
- If c is a constructor of kind $\vec{A} \longrightarrow K$ over V, and $\vec{E}$ an abstraction list of kind $\vec{A}$ over V, then $c\vec{E}$ is an expression of kind K over V.
- An abstraction of kind $\vec{A} \rightarrow K$ over V is an expression of kind K over $(V, \vec{A})$.
- An abstraction list of kind $A_1, \ldots, A_n$ is an abstraction of kind A_1, ..., an abstraction of kind A_n.

Expressions

$$
\begin{aligned}
&\text{Expression } V\, K = \text{Subexpression } V \text{ -Expression } K\\
&\text{VExpression } V\, K = \text{Expression } V\, (\text{varKind } K)\\
&\text{Abstraction } V\, (\text{SK } KK\, L) = \text{Expression } (\text{extend } V\, KK)\, L\\
&\text{ListAbstraction } V\, AA = \text{Subexpression } V \text{ -ListAbstraction } AA
\end{aligned}
$$

```
infixr 5 _::_
data Subexpression V where
  var : ∀ {K} → Var V K → VExpression V K
  app : ∀ {AA} {K} → Constructor (SK AA K) →
    ListAbstraction V AA → Expression V K
  [] : ListAbstraction V []
  _::_ : ∀ {A} {AA} → Abstraction V A →
    ListAbstraction V AA → ListAbstraction V (A :: AA)
```

Replacement and Substitution

A *replacement* is a mapping from variables to variables:

$$\mathsf{Rep} : \mathsf{Alphabet} \longrightarrow \mathsf{Alphabet} \longrightarrow \mathsf{Set}$$
$$\mathsf{Rep}\ U\ V = \forall\ \{K\} \longrightarrow \mathsf{Var}\ U\ K \longrightarrow \mathsf{Var}\ V\ K$$

A *substitution* is a mapping from variables to expressions:

$$\mathsf{Sub} : \mathsf{Alphabet} \longrightarrow \mathsf{Alphabet} \longrightarrow \mathsf{Set}$$
$$\mathsf{Sub}\ U\ V = \forall\ \{K\} \longrightarrow \mathsf{Var}\ U\ K \longrightarrow \mathsf{VExpression}\ V\ K$$

Fusion Laws

With these definitions, we can prove the fusion laws:

$$\text{sub-}\bullet : \forall \{U\ V\ W\ C\ K\} \\ (E : \text{Subexpression}\ U\ C\ K)\ \{\sigma : \text{Sub}\ V\ W\}\ \{\rho : \text{Sub}\ U\ V\} \to \\ E\,[\,\sigma \bullet \rho\,] \equiv E\,[\,\rho\,]\,[\,\sigma\,]$$

$$\text{sub-ref} : \forall \{V\ C\ K\} \\ (E : \text{Subexpression}\ V\ C\ K) \to E\,[\,\text{var}\,] \equiv E$$

Example — The π Calculus


```
data PiCalcConstructor : ConstructorKind → Set where
  -receive : PiCalcConstructor
     (channel ◊ ⟶ (-channel ⟶ program ◊) ⟶
        program ◊)
  -send : PiCalcConstructor
     (channel ◊ ⟶ channel ◊ ⟶ program ◊ ⟶
        program ◊)
  -simul : PiCalcConstructor
     (program ◊ ⟶ program ◊ ⟶ program ◊)
  -new : PiCalcConstructor
     ((-channel ⟶ program ◊) ⟶ program ◊)
  -spawn : PiCalcConstructor
     (program ◊ ⟶ program ◊)
  -term : PiCalcConstructor
     (program ◊)
```

Contexts

A *context* over $\{x_1, \ldots, x_n\}$ is a sequence $x_1 : A_1, \ldots, x_n : A_n$ where, for each i, if x_i has kind K_i, then A_i is an expression of kind parent K_i over $\{x_1, \ldots, x_{i-1}\}$.
Define

$$\text{typeof} : \text{Var}\, V\ K \to \text{Context}\, V \to \text{Expression}\, V\ (\text{parent}\ K)$$

ρ is a replacement from Γ to Δ, $\rho : \Gamma \to \Delta$ iff, for every x,

$$\text{typeof}(\rho\, x)\, \Delta \equiv (\text{typeof}\, x\, \Gamma)\langle \rho \rangle$$

Prove lemmas such as:
If $\rho : \Gamma \to \Delta$ then $\rho^{\uparrow} : \Gamma, x : A \to \Delta, x : A\langle\rho\rangle$.

Reduction Relations (Work in Progress)

Current Version A *reduction relation* is a relation R between expressions of the same kind such that, if ERF, then E is not a variable. We can define $\rightarrow_R$, $\twoheadrightarrow_R$, $\simeq_R$ and prove results like:

$$E \;\text{Red.}\!\Rightarrow\; F \rightarrow E\,[\,\sigma\,]\;\text{Red.}\!\Rightarrow\; F\,[\,\sigma\,]$$

Reduction Relations (Work in Progress)

Next Version

- A *second-order alphabet* is a finite set of *metavariables*, to each of which is associated an *abstraction kind*.
- The set of *patterns* over a second-order alphabet is defined as follows:
  - If X is a second-order variable of kind $K_1 \to \cdots \to K_n \to L$, and P_1 is a pattern of kind $K_1, \ldots,$ P_n is a pattern of kind K_n over V, then $X[P_1, \ldots, P_n]$ is a pattern of kind L over V.
  - If c is a constructor of kind $K_1 \to \cdots \to K_n \to L$ and, for every i, P_i is a pattern of kind B_i over $V \cup \{x_1 : A_{i1}, \ldots, x_{r_i} : A_{ir_i}\}$, where $K_i \equiv A_{i1} \to \cdots \to A_{ir_i} \to B_i$, then $c([x_{11}, \ldots, x_{1r_1}]P_1, \ldots, [x_{n1}, \ldots, x_{nr_n}]P_n)$ is a pattern of kind L over V.

Reduction Relations (Work in Progress)

Next Version Define:

- an *instantiation* from a second-order alphabet U to a first-order alphabet V is a function mapping every metavariable of kind A to an abstraction of kind A
- given a pattern P of kind K and instantiation τ, define the expression $P[\tau]$.

A *reduction relation* R is a set of pairs of patterns of the same kind. M is a redex that contracts to N iff there exists a pair (P, Q) in R and τ such that $M \equiv P[\tau]$, $N \equiv Q[\tau]$.

Example β-reduction is the reduction relation consisting of one pair:

$$(\mathsf{app}(\Lambda(A[], [x]M[x]), N[]), M[N[]])$$

over the alphabet
$\{A : \mathbf{Type}, M : \mathbf{Term} \to \mathbf{Term}, N : \mathbf{Term}\}$.

Rules of Deduction (Work in Progress)

Given a list of patterns $((\Delta_1, P_1), \ldots, (\Delta_n, P_n), C)$, define the rule of deduction

$$\frac{\Gamma, \Delta_1[\tau] \vdash P_1[\tau] \qquad \Gamma, \Delta_n[\tau] \vdash P_n[\tau]}{\Gamma \vdash C[\tau]}$$

Prove general results:

- If $\vdash$ is defined by the variable rule and rules of deduction all of the form above, then the Weakening and Substitution Lemmas hold.

Conclusion

Conclusion

Source code available at:
`http://www.github.com/radams78/MetaL`
For the future:

- Reduction relation and rules as instantiations of *patterns* with second-order variables.
- Interface for representation of syntax.
- Translation between two grammars.
- The POPLMark challenge.

Bibliography

- Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. *SIGPLAN Not.*, 43(1):3–15, January 2008
- Robin Adams. Formalized metatheory with terms represented by an indexed family of types. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004*, volume 3839 of *LNCS*, pages 1–16. Springer, 2006
- Ernesto Copello, Álvaro Tasistro, Nora Szasz, Ana Bove, and Maribel Fernández. Alpha-structural induction and recursion for the lambda calculus in constructive type theory. *Electronic Notes in Theoretical Computer Science*, 323:109 – 124, 2016

Bibliography

- Conor McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP '10, pages 1–12, New York, NY, USA, 2010. ACM
- Conor Mcbride and James Mckinna. Functional pearl: I am not a number–i am a free variable. In *In Proc. Haskell workshop*, pages 1–9. ACM, 2004
- F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM Press
- Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994