MetaL — A Library for Formalised Metatheory in Agda

Robin Adams[1]

University of Bergen r.adams@ii.uib.no

Abstract. There are now many techniques for formalising metatheory (nominal sets, higher-order abstract syntax, etc.) but, in general, each requires the syntax and rules of deduction for a system to be defined afresh, and so all the proofs of basic lemmas must be written anew when we work with a new system, and modified every time we modify the system.
In this rough diamond, we present an early version of MetaL ("Metatheory Library"), a library for formalised metatheory in Agda. There is a type Grammar of grammars with binding, and types Red G of reduction relations and Rule G of sets of rules of deduction over G : Grammar. A grammar is given by a set of constructors, whose type specifies how many arguments it takes, and how many variables are bound in each argument. Reduction relations and rules of deduction are given by patterns, or expressions involving second-order variables.
The library includes a general proof of the substitution lemma. The final version is planned to include proofs of Church-Rosser for reductions with no critical pairs, and Weakening and Substitution lemmas for appropriate sets of rules of deduction.
MetaL has been designed with the following criteria in mind. It is easy to specify a grammar, reduction rule or set of rules of deduction: the Agda definition is the same length as the definition on paper. The general results are immediately applicable. When working within a grammar G, it should be possible to define functions by induction on expressions, and prove results by induction on expressions or induction on derivations, using only Agda's built-in pattern matching.

1 Introduction

A large number of techniques for representing syntax with binding and formalizing metatheory have been invented (see Section 6). Typically, however, whatever representation of syntax we are using, we begin by defining an inductive datatype for the expressions of the system; then inductive datatypes for the derivability relation. We then define operations such as substitution, and prove lemmas, by induction over these datatypes.

This causes some problems. As an example, for most systems we want to prove the Substitution Lemma, which on paper reads

$$M[x := N][y := P] \equiv M[y := P][x := N[y := P]] \ .$$

Quite naturally, we prove this by induction on M.

There is no direct way to re-use the proof of the Substitution Lemma from a previous formalisation of a different system; whenever we treat a new formal system, we must prove all the metatheorems starting from nothing. If, in the course of the formalization, we decide to change the system, then we must edit the proof of every one of these lemmas. The proofs typically involve a lot of duplicated code: if there are (say) 20 constructors with binding in the object language, then there will be 20 very similar clauses in the proof of this lemma.

We can think of several different ways to handle these problems in individual cases; for example, we may write a tactic in Coq to eliminate duplicated code, or look for a representation of syntax in which the Substitution Lemma becomes a definitional equality (e.g. [2]). One solution is to create a library of metatheorems about formal systems in general. For this library to be useful in practice, it should satisfy these three criteria:

- When defining a term of type Grammar, the definition should be comparable in length to the definition of the system on paper.
- Results such as the Substitution Lemma should be provable 'once and for all' for all grammars.
- It should be possible to define functions by induction on the syntax, prove lemmas by induction on syntax, and prove lemmas by induction on derivations.

We present here an Agda library called MetaL ('Metatheory Library') which is being designed with these criteria as goals. It contains a definition of an arbitrary grammar with binding, and proofs of results about the operations of *replacement* (mapping variables to variables) and substitution. General notions of reduction rules and rules of deduction are being added to the library as a work-in-progress. This library has been used to prove the strong normalization of the simply-typed lambda calculus, and used for a metatheoretic result for a more complex system known as Predicative Higher-Order Minimal Logic (PHOML) [1].

2 Grammar

Example 1 (Simply Typed Lambda Calculus). For a running example, we will construct the grammar of the simply-typed lambda-calculus, with Church-typing and one constant ground type $\bot$. On paper, in BNF-style, we write the grammar as follows:

$$\begin{aligned} &\text{Type} \quad A ::= \bot \mid A \to A \\ &\text{Term} \; M ::= x \mid MM \mid \lambda x : A.M \end{aligned}$$

2.1 Taxonomy

A *taxonomy* is a set of *expression kinds*, divided into *variable kinds* and *non-variable kinds*. The intention is that the expressions of the grammar are divided

into expression kinds. Every variable ranges over the expressions of one (and only one) variable kind.

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

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

An *alphabet* is a finite set of *variables*, to each of which is associated a variable kind. We write Var V K for the set of all variables in the alphabet V of kind K.

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


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

2.2 Grammar

Definition 2. *An* abstraction kind *has the form* $K_1 \to \cdots \to K_n \to L$*, where each* K_i *is an abstraction kind, and* L *is an expression kind.*

A constructor kind *has the form* $A_1 \to \cdots \to A_n \to K$*, where each* A_i *is an abstraction kind, and* K *is an expression kind.*

A *grammar* over a taxonomy consists of:

- a set of *constructors*, each with an associated constructor kind;
- a function assigning, to each variable kind, an expression kind, called its *parent*. (The intention is that, when a declaration $x : A$ occurs in a context, if x has kind K, then the kind of A is the parent of K.)

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

 record Grammar : Set₁ where
```

```
field
 taxonomy : Taxonomy
 isGrammar : IsGrammar taxonomy
open Taxonomy taxonomy public
open IsGrammar isGrammar public
```

Definition 3. *We define simultaneously the set of* expressions *of kind K over V for every expression kind K and alphabet V; and the set of* abstractions *of kind A over V for every abstraction kind A and alphabet V.*

- *Every variable of kind K in V is an expression of kind K over V.*
- *If c is a constructor of kind $A_1 \to \cdots \to A_n \to K$, and M_1 is an abstraction of kind A_1, ..., M_n is an abstraction of kind A_n (all over V), then*

  $$cM_1 \cdots M_n$$

  is an expression of kind K over V.
- *An abstraction of kind $K_1 \to \cdots \to K_n \to L$ over V is an expression of the form*

  $$[x_1, \ldots, x_n]M$$

  where each x_i is a variable of kind K_i, and M is an expression of kind L over $V \cup \{x_1, \ldots, x_n\}$.

In the Agda code, we define simultaneously the following four types:

- Expression $V\,K$ = Subexp V-Expression K, the type of expressions of kind K;
- VExpression $V\,K$ = Expression V (varKind K), a convenient shorthand when K is a variable kind;
- Abstraction $V\,A$, the type of abstractions of kind A over V
- ListAbstraction $V\,AA$: if $AA \equiv [A_1, \ldots, A_n]$, then ListAbstraction $V\,AA$ is the type of lists of abstractions $[M_1, \ldots, M_n]$ such that each M_i is of kind A_i.

```
data Subexp (V : Alphabet) : ∀ C → Kind C → Set
Expression : Alphabet → ExpressionKind → Set
VExpression : Alphabet → VariableKind → Set
Abstraction : Alphabet → AbstractionKind → Set
ListAbstraction : Alphabet → List AbstractionKind → Set

Expression V K = Subexp V -Expression K
VExpression V K = Expression V (varKind K)
Abstraction V (SK KK L) = Expression (extend V KK) L
ListAbstraction V AA = Subexp V -ListAbstraction AA

infixr 5 _::_
data Subexp V where
 var : ∀ {K} → Var V K → VExpression V K
```

```
app : ∀ {AA} {K} → Con (SK AA K) → ListAbstraction V AA → Expression V K
[] : ListAbstraction V []
_::_ : ∀ {A} {AA} → Abstraction V A → ListAbstraction V AA → ListAbstraction V (A :: AA)
```

Example 4. The grammar given in Example 1 has four constructors:

- $\bot$, of kind type;
- $\to$, of kind type $\longrightarrow$ type $\longrightarrow$ type
- appl, of kind term $\longrightarrow$ term $\longrightarrow$ term
- λ, of kind type $\longrightarrow$ (term $\longrightarrow$ term) $\longrightarrow$ term

The kind of the final constructor λ should be read like this: λ takes a type A and a term M, binds a term variable x within M, and returns a term $\lambda x : A.M$

```
type : ExpressionKind
type = nonVariableKind -type

term : ExpressionKind
term = varKind -term

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

2.3 Families of Operations

Our next aim is to define replacement and substitution.

Definition 5 (Replacement).

- *A* replacement *from* U *to* V, $\rho : U \to_R V$, *is a family of functions* ρ_K : *Var*$U\ K \to$ *Var*$V\ K$ *for every variable kind* K.
- *For* x : *Var*$U\ K$, *define* $\rho(x) \stackrel{\text{def}}{=} \rho_K(x)$.
- *Define* $\uparrow : V \to_R V, K$ *by* $\uparrow_L (x) \equiv x$.
- *Define* $(1_V)_K(x) \equiv x$
- *Define* $(\sigma \circ \rho)_K(x) \equiv \sigma_K(\rho_K(x))$
- *Define* $\sigma^{\uparrow}_K(x_0) \equiv x_0$, *and* $\sigma^{\uparrow}_L(\uparrow x) \equiv \uparrow \sigma_L(x)$.

We write $E\langle\rho\rangle$ for the action of a replacement ρ on a subexpression E.

Definition 6 (Substitution).

- *A* substitution *from* U *to* V, $\sigma : U \Rightarrow V$, *is a family of functions* σ_K : *Var*$U\ K \to$ *Expression*$V\ K$ *for every variable kind* K.
- *For* x : *Var*$U\ K$, *define* $\sigma(x) \stackrel{\text{def}}{=} \sigma_K(x)$

– *Define* $\uparrow : V \to_R V, K$ *by* $\uparrow_L (x) \equiv x$.
– *Define* $(1_V)_K(x) \equiv x$
– *Define* $(\sigma \circ \rho)_K(x) \equiv \rho_K(x)[\sigma]$
– *Define* $\sigma^{\uparrow}_K(x_0) \equiv x_0$ *and* $\sigma^{\uparrow}_L(\uparrow x) \equiv \sigma_L(x)\langle\uparrow\rangle$.

We write $E[\sigma]$ for the action of a substitution σ on a subexpression E.

We can prove the following results about substitution, and analagous results about replacement.

Lemma 7. *1. If* $\rho \sim \sigma$ *then* $E[\rho] \equiv E[\sigma]$.
2. $1^{\uparrow}_V = 1_{V,K}$
3. $E[1_V] \equiv E$
4. $E[\sigma \circ \rho] \equiv E[\rho][\sigma]$
5. $\tau \circ (\sigma \circ \rho) \sim (\tau \circ \sigma) \circ \rho$
6. If $\sigma : U \Rightarrow V$ *then* $1_V \circ \sigma \sim \sigma \sim \sigma \circ 1_U$

3 Examples

3.1 Simply-Typed Lambda Calculus

We have formalized the Tait proof that the simply- We give the syntax and rules of deduction of the simply-typed lambda calculus.

We define the set of *computable* terms of each type A:

$$C_\Gamma(\bot) = \{\delta \mid \Gamma \vdash \delta : \bot \text{ and } \delta \in SN\}$$
$$C_\Gamma(\phi \to \psi) = \{\delta \mid \Gamma : \delta : \phi \to \psi \text{ and } \forall \Delta \supseteq \Gamma . \forall \epsilon \in C_\Delta(\phi) . \delta\epsilon \in C_\Delta(\psi)\}$$

Note that this can be done by induction on the expression of type Prop. There is no need, for example, to handle the case where A is a term.

We go on to prove that every computable term is strongly normalizing, and every typable term is computable.

3.2 Predicative Higher-Order Minimal Logic

This library has been used for the proof of the canonicity of predicative higher-order minimal logic (PHOML). The proof is described in the paper [1]. Here, we note only that this formalization involved defining the notion of *path substitution*.

The syntax has kinds of *types*, *terms* and *paths*. The operation of *path substitution* takes terms M, N, N' and a path P, and returns a path $M\{x := P : N = N'\}$. The intuition is that, if P is a proof that $N = N'$, then $M\{x := P : N = N'\}$ is a proof that $M[x := N] = M[x := N']$.

This is defined in Agda as follows

Note again that this can be defined by induction on M, and only the cases where M is a term need to be defined.

4 Reduction Relations and Rules of Deduction (Work in Progress)

In the above examples, the reduction relation and rules of deduction are given as inductive datatypes. This has all the disadvantages discussed in the introduction. We want a general type of reduction relations and a type of sets of rules of deduction, and we want to prove general results about them. The following progress has been made towards this goal.

We look at one of the clauses in the definition of β-reduction: $(\lambda x : A.M)N \rhd M[x := N]$. We shall call A, M and N *metavariables* or *second-order variables*, and the expressions $(\lambda x : A.M)N$ and $M[x := N]$ we shall call *patterns* over the *second-order alphabet* $\{A, M, N\}$. The definition is as follows.

Definition 8. *1. A* second-order alphabet *is a set of* metavariables*, to each of which is associated an abstraction kind. To distinguish, the alphabets defined in Definition* **??** *are now called* first-order alphabets

2. *Given a second-order alphabet U, a first-order alphabet V, and an expression kind K, the set of* patterns *of kind K over U and V are defined as follows.*
   - *If $x \in V$ is a variable of kind K, then x is a pattern of kind K.*
   - *If $X \in U$ is a metavariable of kind $K_1 \to \cdots \to K_n \to L$, and P_i is a pattern of kind K_i over U and V, then $X[P_1, \ldots, P_n]$ is a pattern of kind L over U and V.*
   - *If c is a constructor of kind $A_1 \to A_n \to K$ and, for each i, we have*

     $$A_i \equiv L_{i1} \to \cdots \to L_{ir_i} \to M_i$$

     and P_i is a pattern of kind M_i over U and $V \cup \{x_{i1}, \ldots, x_{ir_i}\}$, where each x_{ij} has kind L_{ij}, then

     $$c([x_{11}, \ldots, x_{1r_1}]P_1, \ldots, [x_{n1}, \ldots, x_{nr_n}]P_n]$$

     is a pattern of kind K over U and V.

3. *Let U be a second-order alphabet and V a first-order alphabet. An* instantiation *$\tau : U \to V$ is a function mapping every metavariable $X \in U$ of abstraction kind K to an abstraction $\tau(X)$ of kind K over V.*

4. *Given a pattern P of kind K over U and V; an instantiation $\tau : U \to W$; and a replacement $\rho : V \to W$; define the expression $P[\tau, \rho]$ of kind K over W as follows.*

   $$x[\tau, \rho] \equiv \rho(x)$$

   If $\tau(X) \equiv [x_1, \ldots, x_n]E$ then

   $$X[P_1, \ldots, P_n][\tau, \rho] \equiv E[x_1 := P_1[\tau, \rho], \ldots, x_n := P_n[\tau, \rho]]$$
   $$c([\boldsymbol{x_1}]P_1, \ldots, [\boldsymbol{x_n}]P_n)[\tau, \rho] \equiv c([\boldsymbol{x_1}]P_1[\tau, \rho^{\uparrow\cdots\uparrow}], \ldots, P_n[\tau, \rho^{\uparrow\cdots\uparrow}])$$

   The expression $P[\tau, \rho]$ is called an instance *of the pattern P.*

5. *A* reduction relation *is a pair* (P, Q) *of expressions of the same kind over the same alphabet, such that* P *has the form* $c\boldsymbol{R}$ *for some constructor* c*.*

The reduction relation β is the reduction relation consisting of one pair:

$$(\lambda x : A.M[x])N \rhd M[N] \text{ or } (\mathsf{appl}(\lambda(A, [x]M[x]), N), M[N]) \ .$$

Observe that the instances $(P[\tau, \rho], Q[\tau, \rho])$ of this pair are exactly the pairs of expressions (M, N) such that M is a β-redex and N its contractum.

A rule of deduction can similarly be represented as a sequence of patterns over U and V, each with a context over U and V. For example, the introduction rule for the $\rightarrow$-type is represented by the pair $((x : A[], M[x], B[]), (\epsilon, \lambda x : A[].M[x], A[] \rightarrow B[]))$.

5 Limitations

– There is no way to express that an expression depends on some variable kinds but not others. (E.g. in our simply-typed lambda calculus example: the types do not depend on the term variables.) This leads to some boilerplate that is needed, proving lemmas of the form

$$(\bot\, U)[\sigma] \equiv \bot\, V \tag{1}$$

There is a workaround for this special case. We can declare all the types as constants: This is what we used for the project PHOML.
For a general solution, we would need to parametrise alphabets by the set of variable kinds that may occur in them, and then prove results about mappings from one type of alphabet to another. We could then prove once-and-for-all versions of the lemmas like (1). It remains to be seen whether this would still be unwieldy in practice.

6 Related Work

Higher-order abstract syntax and Kipling both restrict the operations that can be defined on the syntax.

7 Conclusion

References

1. Adams, R., Bezem, M., Coquand, T.: A strongly normalizing computation rule for univalence in higher-order minimal logic. CoRR abs/1610.00026 (2016), http://arxiv.org/abs/1610.00026, under review for TYPES 2016
2. McBride, C.: Outrageous but meaningful coincidences (June 2010)