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

1.1 Design Criteria

This library was produced with the following design goals.

– The library should be *modular*. There should be a type Grammar, and results such as the Substitution Lemma should be provable 'once and for all' for all grammars.[1]

[1] For future versions of the library, we wish to have a type of reduction rules over a grammar, and a type of theories (sets of rules of deduction) over a grammar.

– It should be possible for the user to define their own operations, such as path substitution
– Operations which are defined by induction on expressions should be definable by induction in Agda. Results which are proved by induction on expressions should be proved by induction in Agda.

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:

$$\begin{array}{l} \text{Type } A ::= \bot \mid A \to A \\ \text{Term } M ::= x \mid MM \mid \lambda x : A.M \end{array}$$

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;
- $\rightarrow$, 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. Many of the results about these two operations have very similar proofs, so in order to avoid duplicating code, we make the following definition.

Definition 5 (Family of Operations). *A* family of operations $\Rightarrow$ *consists of:*

- *for any alphabets U, V, a set $U \Rightarrow V$ of* operations *from U to V;*
- *for any operation $\sigma : U \Rightarrow V$ and variable $x : \mathsf{Var}\, U\, K$, an expression $\sigma(x) : \mathsf{Expression}\, V\, K$*
- *for any alphabet V and variable kind K, an operation $\uparrow : V \Rightarrow V, K$*
- *for any alphabet V, an operation $1_V : V \Rightarrow V$*
- *for any operations $\rho : U \Rightarrow V$ and $\sigma : V \Rightarrow W$, an operation $\sigma \circ \rho : U \Rightarrow W$, the* composition *of σ and ρ;*
- *for any operation $\sigma : U \Rightarrow V$ and variable kind K, an operation $\sigma^{\uparrow} : U, K \Rightarrow V, K$, the* lifting *of σ;*

such that:

- $\uparrow (x) \equiv x$ *for any variable* x
- $1_V(x) \equiv x$ *for any variable* x
- $\sigma^{\uparrow}(x_0) \equiv x_0$
- $\sigma^{\uparrow}(x) \equiv \sigma(x)[\uparrow]$
- $(\sigma \circ \rho)(x) \equiv \rho(x)[\sigma]$

where, if $E : \mathsf{Expression}\, U\, K$ and $\sigma : U \Rightarrow V$ then $E[\sigma] : \mathsf{Expression}\, V\, K$, the action *of σ on E, is defined by*

$$\begin{aligned} x[\sigma] &\stackrel{\text{def}}{=} \sigma(x) \\ ([x_1, \ldots, x_n]E)[\sigma] &\stackrel{\text{def}}{=} E[\sigma^{\uparrow\uparrow\cdots\uparrow}] \\ (cE_1 \cdots E_n)[\sigma] &\stackrel{\text{def}}{=} c(E_1[\sigma]) \cdots (E_n[\sigma]) \end{aligned}$$

We write $\rho \sim \sigma$ iff ρ and σ are extensionally equal, i.e. $\rho(x) \equiv \sigma(x)$ for every variable x.

The way that this is formalised in Agda is described in Appendix **??**.

It is easy to see that our two examples of replacement and substitution fit this pattern.

Definition 6 (Replacement). Replacement *is the family of operations defined as follows.*

- *A* replacement *from U to V, $\rho : U \rightarrow_R V$, is a family of functions $\rho_K : \mathsf{Var}\, U\, K \rightarrow \mathsf{Var}\, V\, K$ for every variable kind K.*
- *For $x : \mathsf{Var}\, U\, K$, define $\rho(x) \stackrel{\text{def}}{=} \rho_K(x)$.*
- *Define $\uparrow : V \rightarrow_R V, K$ 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* $\sigma^{\uparrow}_L(\uparrow x) \equiv \uparrow \sigma_L(x)$.

REP : OpFamily

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

Definition 7 (Substitution). Substitution *is the family of operations defined as follows.*

– *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* $\sigma^{\uparrow}_L(\uparrow x) \equiv \sigma_L(x)\langle\uparrow\rangle$.

SUB : OpFamily

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

Results about Families of Operations. We can prove the following results about an arbitrary family of operations.

Lemma 8. *1. If* $\rho \sim \sigma$ *then* $E[\rho] \equiv E[\sigma]$.

```
ap-congl : ∀ {U} {V} {C} {K}
 {ρ σ : Op U V} → ρ ∼op σ → ∀ (E : Subexp U C K) →
 ap ρ E ≡ ap σ E
```

2. $1^{\uparrow}_V = 1_{V,K}$

```
liftOp-idOp : ∀ {V} {K} → liftOp K (idOp V) ∼op idOp (V , K)
```

3. $E[1_V] \equiv E$

```
ap-idOp : ∀ {V} {C} {K} {E : Subexp V C K} → ap (idOp V) E ≡ E
```

4. $E[\sigma \circ \rho] \equiv E[\rho][\sigma]$

```
ap-comp : ∀ {U V W C K} (E : Subexp U C K) {σ ρ} →
 ap H (_∘_ {U} {V} {W} σ ρ) E ≡ ap F σ (ap G ρ E)
```

5. $\tau \circ (\sigma \circ \rho) \sim (\tau \circ \sigma) \circ \rho$

```
assoc : ∀ {U} {V} {W} {X}
 {τ : Op W X} {σ : Op V W} {ρ : Op U V} →
 τ ∘ (σ ∘ ρ) ∼op (τ ∘ σ) ∘ ρ
```

6. *If* $\sigma : U \Rightarrow V$ *then* $1_V \circ \sigma \sim \sigma \sim \sigma \circ 1_U$

```
unitl : ∀ {U} {V} {σ : Op U V} → idOp V ∘ σ ∼op σ

unitr : ∀ {U} {V} {σ : Op U V} → σ ∘ idOp U ∼op σ
```

2.4 Substitution for the Last Variables

Given an alphabet $V \cup \{x_0, \ldots, x_n\}$ and expressions E_0, ..., E_n, we define the substitution

$$[x_0 := E_0, \ldots, x_n := E_n] : V \cup \{x_0, \ldots, x_n\} \Rightarrow V$$

```
botSub : ∀ {V} {KK} → HetsnocList (VExpression V) KK → Sub (snoc-extend V KK) V
botSub {KK = []} _ x = var x
botSub {KK = _ snoc _} (_ snoc E) x₀ = E
botSub {KK = _ snoc _} (EE snoc _) (↑ x) = botSub EE x

infix 65 x₀:=_
x₀:=_ : ∀ {V} {K} → Expression V (varKind K) → Sub (V , K) V
x₀:= E = botSub ([] snoc E)
```

We have the following results about this substitution:

Lemma 9. *1.* $E'\langle\uparrow\rangle[x_0 := E] \equiv E$

```
botSub-up : ∀ {F} {V} {K} {C} {L} {E : Expression V (varKind K)} (comp : Composition SubLF F S
 ap F (up F) E' ⟦ x₀:= E ⟧ ≡ E'
```

2. $E'[x_0 := E][\sigma] \equiv E'[\sigma^{\uparrow}][x_0 := E[\sigma]]$

```
•-botSub'' : ∀ {U} {V} {C} {K} {L}
 {E : Expression U (varKind K)} {σ : Sub U V} (F : Subexp (U , K) C L) →
  F ⟦ x₀:= E ⟧ ⟦ σ ⟧ ≡ F ⟦ liftSub K σ ⟧ ⟦ x₀:= (E ⟦ σ ⟧) ⟧
```

3 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

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

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.

4 Related Work

5 Conclusion

References