MetaL — A Library for Formalised Metatheory in Agda

Robin Adams

March 24, 2017

1 Introduction

2 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]
- 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.

3 Grammar

Example 3.1 (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:

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

[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.

3.1 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
  VarKind : Set
  NonVarKind : Set

 data ExpKind : Set where
  varKind : VarKind → ExpKind
  nonVarKind : NonVarKind → ExpKind
```

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 → 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
```

Example 3.2. For the simply-typed lambda-calculus, there are two expression kinds: type, which is a non-variable kind, and term, which is a variable kind:

```
data stlcVarKind : Set where
 -term : stlcVarKind

data stlcNonVarKind : Set where
 -type : stlcNonVarKind

stlcTaxonomy : Taxonomy
stlcTaxonomy = record {
 VarKind = stlcVarKind ;
 NonVarKind = stlcNonVarKind }
```

3.2 Grammar

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.

To define these, we introduce the notion of a *simple kind*: a simple kind over sets S and T is an object of the form $s_1 \to \cdots \to s_n \to t$, where each $s_i \in S$ and $t \in T$.

```
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


AbsKind = SimpleKind VarKind ExpKind
ConKind = SimpleKind AbsKind ExpKind
```

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 : ConKind → Set
  parent : VarKind → ExpKind

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

Example 3.3. The grammar given in Example 3.1 has four constructors:

- $\bot$, of kind type;
- $\rightarrow$, of kind type $\longrightarrow$ type $\longrightarrow$ type
- app, 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 : ExpKind
type = nonVarKind -type

term : ExpKind
term = varKind -term

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

stlcParent : VarKind → ExpKind
stlcParent -term = type

stlc : Grammar
stlc = record {
 taxonomy = stlcTaxonomy ;
 isGrammar = record {
  Con = stlcCon ;
  parent = stlcParent } }

open STLCGrammar
open Grammar STLCGrammar.stlc

Type : Alphabet → Set
Type V = Expression V type

Term : Alphabet → Set
Term V = Expression V term

⊥ : ∀ V → Type V
⊥ V = app -bot []

_⇒_ : ∀ {V} → Type V → Type V → Type V
A ⇒ B = app -arrow (A :: B :: [])
```

```
appl : ∀ {V} → Term V → Term V → Term V
appl M N = app -app (M :: N :: [])

Λ : ∀ {V} → Type V → Term (V , -term) → Term V
Λ A M = app -lam (A :: M :: [])
```

4 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:

  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.