MetaL — A Library for Formalised Metatheory in Agda

Robin Adams

March 28, 2017

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]
- 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 2.1 (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:

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

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

[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.

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
```

Example 2.2. 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 stlcVariableKind : Set where
 -term : stlcVariableKind

data stlcNonVariableKind : Set where
 -type : stlcNonVariableKind

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

2.2 Grammar

Definition 2.3. 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 $t \in T$.

We implement this by saying a simple kind over S and T consists of a list of objects of S, and one object of 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
```

We can construct an object of type SK S T by writing

$$s_1 \longrightarrow \cdots \longrightarrow s_n \longrightarrow t\Diamond \ .$$

(The '$\Diamond$' symbol marks the end of the simple kind. It is needed to help Agda disambiguate the syntax.)

We are now able to write Definition 2.3 like this:

```
AbstractionKind = SimpleKind VariableKind ExpressionKind
ConstructorKind = SimpleKind AbstractionKind ExpressionKind
```

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 2.4. 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 2.5. The grammar given in Example 2.1 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 ◊)

stlcParent : VariableKind → ExpressionKind
stlcParent -term = type

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

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 :: [])
```

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.