MetaL — A Library for Formalised Metatheory in Agda

Robin Adams

March 23, 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

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:

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

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

[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
  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 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 }
open Taxonomy stlcTaxonomy

type : ExpKind
type = nonVarKind -type

term : ExpKind
term = varKind -term
```

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
```