Coq projects for type theory 2010

Herman Geuvers, James McKinna, Freek Wiedijk

December 21, 2010

Here are five projects for the type theory course to choose from. Each student has to choose one of these projects. More than one student can choose the same project, but they should not collaborate on it.

At the end of the course each student needs to submit both a Coq formalization, plus a short report (up to ten pages) that describes the formalization and discusses the choices made while formalizing.

For each project we first give a high level description of what should be done, followed by details of one specific way to do that. However, students are free not to follow these specifics.

1 Type check the simply typed lambda calculus

Formalize the typing rules of the simply typed lambda calculus, then formalize a type checker *for this system, and finally prove that the type checker will produce a correct type judgment if it succeeds.*

This project is an instance of *reflection*. Although the type theory of Coq contains simply typed lambda calculus as a subsystem, the terms and types that the formalization talks about will not be *those* Coq terms and types, but syntactic objects *modelled* in Coq.

You do not need to prove the *completeness* of your type checker.

Here are specifics of one possible solution, which takes 97 lines of Coq:

- Define inductive types `type` and `term` for the types and terms of the simply typed lambda calculus. For example the definition of `type` might look like

  ```
  Inductive type : Set :=
  | var_type : string -> type
  | fun_type : type -> type -> type
  ```

  With this definition the type $(A \to B) \to C$ would be represented by

  ```
  fun_type (fun_type (var_type "A") (var_type "B"))
           (var_type "C")
  ```

To get string notation in Coq, put

```
Require Import String.
Open Local Scope string_scope.
```

at the start of your file.

- Define an inductive predicate `has_type`, such that the Coq formula

  ```
  has_type Gamma M A
  ```

  corresponds to the derivability of the judgment

  $$\Gamma \vdash M : A$$

- Write a recursive function `type_check` that (in a given context) returns the type of an element of `term`. A possible Coq type for this function might be

  ```
  type_check
       : list (string * type) -> term -> option type
  ```

  If the input term (the second argument) is not type correct, the function will have to return `None`. For this reason the output type is not `type` but `option type`.

- You will need to look up variables in the context. For this define an inductive predicate `assoc` (to be used in the variable case of `has_type`) and a recursive function `lookup` (to be used in the variable case of `type_check`):

  ```
  assoc
       : forall A B : Set, list (A * B) -> A -> B -> Prop

  lookup
       : forall A B : Set,
         (forall x y : A, {x = y} + {x <> y}) ->
         list (A * B) -> A -> option B
  ```

  The third argument of `lookup` is a decision procedure for equality on `A`. If the keys are `string`s this argument should be `string_dec`.

  The functions `assoc` and `lookup` correspond to each other in exactly the same way that `has_type` and `type_check` do.

- The type checker needs to be able to decide equality of types. (If you apply a function to an argument, the type of the argument needs to match the type of the domain of the function.) For this prove the lemma

```
type_dec
     : forall A B : type, {A = B} + {A <> B}
```

A convenient tactic for proving this is `decide equality.`

- Next we will need to prove our type checker correct. A nice way to do this is by changing the types of `lookup` and `type_check` to have them also return 'proof objects' for the properties of the objects they return:

```
lookup
     : forall A B : Set,
       (forall x y : A, {x = y} + {x <> y}) ->
       forall (l : list (A * B)) (a : A),
       option {b : B | assoc l a b}

type_check
     : forall (Gamma : list (string * type)) (M : term),
       option {A : type | has_type Gamma M A}
```

To find out about the meaning of the set notation `{ ... | ... }` do `Check exist` or `Print sig`.

The function `exist` has implicit arguments. If you want to give those arguments explicitly because Coq cannot figure them out, write `@exist`: then all four arguments can and should be given.

- Often Coq will complain if you just use a simple `match ... with`. In that case using `match ... in ... return ... with` can improve things. For example, the `match` the we used in our definition of `lookup` looks like

```
match l return option {b : B | assoc l a b} with ...
```

2 The pigeon hole principle

Use Coq to formally prove the pigeon hole principle *which says that if you put n pigeons in m holes, with $m < n$, then at least one hole with have more than one pigeon in it.*

Here are specifics of one possible solution, which takes 61 lines of Coq (but with many tactics per line!):

- A possible way to write the statement is:

```
Lemma pigeon_hole :
  forall m n, m < n ->
    forall f, (forall i, i < n -> f i < m) ->
      exists i, i < n /\
        exists j, j < n /\ i <> j /\ f i = f j.
```

Here `f` is the function that maps the number of a pigeon in $\{0 \ldots n-1\}$ to the number of its hole in $\{0 \ldots m-1\}$. The fact that `f` also will map numbers $\geq n$ to something will not hurt.

- A useful tactic to automatically prove equalities and inequalities between natural numbers is

  ```
  omega.
  ```

  To make it available put

  ```
  Require Import Omega.
  ```

  at the start of your file.

- If for natural numbers x and y in a term you want to make a distinction between whether $x \leq y$ or $y < x$, you can write:

  ```
  if le_lt_dec x y then ... else ...
  ```

  Then to do a case split between those two cases in the proof, one can use:

  ```
  elim (le_lt_dec x y).
  ```

3 Proving an expression compiler correct

Formalize both an interpreter and a compiler for a simple language of arithmetical expressions, and show that both give the same results. Compile the expressions to code for a simple stack machine. Use dependent types to make Coq aware of the fact that the compiled code will never lead to a run time error.

Here are specifics of one possible solution, which takes 78 lines of Coq:

- Consider the following expression language:

  $$\begin{array}{rcl} \langle \text{exp} \rangle & ::= & \langle \text{literal} \rangle \mid \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \ldots \\ \langle \text{literal} \rangle & ::= & 0 \mid 1 \mid 2 \mid \ldots \end{array}$$

  Give an `Inductive` definition of a datatype `Exp` of (the abstract syntax for) $\langle$exp$\rangle$s.

- Define a function

  ```
  eval: Exp -> nat
  ```

  giving a semantics for $\langle$exp$\rangle$s.

- Give an `Inductive` definition of a datatype `RPN` of Reverse Polish Notation for ⟨exp⟩s.
- Write a compiler

  ```
  rpn : Exp -> RPN
  ```

- Write an evaluator `rpn_eval` for `RPN`, returning an `option nat`.
- Prove that

  ```
  forall e:Exp, Some (eval e) = rpn_eval (rpn e)
  ```

- Generalize the above to `Exp`ressions containing variables, and evaluation with respect to an environment of bindings of variables to `nat`s.
- Discuss how you might avoid explicit consideration of `None` terms in the definition of `rpn_eval`, and explain how you need to modify your formalization in Coq.

4 Satisfiability of propositional formulas

Implement in Coq a function that searches for a variable assignment that makes a given propositional formula true, and prove that this function is correct.

You do not need to prove the *completeness* of your satisfiability checker.

Here are specifics of one possible solution, which takes 79 lines of Coq:

- We study propositional formulas and check whether they are 'satisfiable'. A formula f is satisfiable if there is a valuation ρ (a valuation is a map that assigns 0 or 1 to each of the proposition variables) such that $\rho(f) = 1$.

  Here, $\rho(f)$ is computed using the well-known 'truth table semantics'.
- Define the inductive type of 'propositional expressions' `form` with the following constructors.

  ```
  f_var : nat -> form
  f_and : form -> form -> form
  f_or : form -> form -> form
  f_imp : form -> form -> form
  f_neg : form -> form
  ```

  `f_var` gives us infinitely many propositional variables, that are all indexed by a natural number.
- Define the notion of a 'model' as a valuation ρ that assigns a boolean to each natural number. This can be done in various ways:

```
model : nat -> bool
model : list (nat * bool)
model : list bool
```

The last two assign a boolean to only finitely many numbers, but a proposition contains only finitely many variables anyway, so that's no problem. each of these choices has pros and cons; probably the second is easiest to work with.

- Define a function `find_model` that, given an `e:form`, computes a model ρ in which `e` is true (i.e. in which $\rho($`e`$) =$ `true`).

  Let `find_model` give an 'error' message if no such ρ exists, by making it of type `form -> option model`. Check the definition of `option` by doing `Print option`

  NB. To define `find_model`, you will probably have to:

  - First collect the list of proposition variables that occur in `e`.
  - Then, by recursion over this list, try out all different valuations of $\{$`true`$,$`false`$\}$ to the proposition variables occurring in `e`.

- Prove that `find_model` 'works'. Specifically: prove that if `find_model e` $\neq$ `None _` , then `find_model e` produces a model of `e`.

5 The unary versus the binary natural numbers

Define both the unary and the binary natural numbers, define addition on both types, define mappings in both directions, show that those mappings are inverse to each other, and finally show that the two addition functions correspond to each other under these mappings.

This exercise is more difficult than it looks, but it is a nice challenge. There are various approaches possible:

- Define the binary numbers with the possibility of leading zeroes. This amounts to 'lists of bits'.

  In the case the mappings between the two types are not simply inverse to each other, as different binary representations might represent the same number. One can define a predicate of two representations being 'equal' (= representing the same number), a predicate of a representation being in normal form (= having no leading zeroes), and a function to normalize a representation (= remove the leading zeroes).

  Our solution using this approach (with many tactics per line!) takes 347 lines and has 34 lemmas.

- Alternatively one can use a binary representation that is unique. There are at least two possibilities for this:

– Define a type of *positive* binary natural numbers first, and then use that to define a type of *all* binary natural number. This is similar to the way the type of binary integers `Z` is defined in the Coq standard library.

  Our solution using this approach takes 155 lines and has 11 lemmas.

– Define *mutual* types of positive and non-negative binary natural numbers:

```
Inductive bnat : Set :=
| b0 : bnat
| i : pnat -> bnat

with pnat : Set :=
| bit0 : pnat -> pnat
| bit1 : bnat -> pnat.
```

  Here the functions `bit0` and `bit1` add a 0 or 1 at the right end of the number, so they amount to doubling the number respectively doubling it and adding one. Note that the type `bnat` has unique representations for all natural numbers.

  Our solution using this approach takes 143 lines and has 10 lemmas.

Both of these approaches are quite hairy with most definitions and lemmas appearing twice for the two different types for binary numbers.