Introduction to Rocq — Arthur Azevedo de Amorim

Lecture 3 - *June 23, 2026*

1 Modelling an Imperative Language

We are going to define the semantics of a simple imperative language, called `Imp`. The language only has if and while statements as control flow, global variables, and arithmetic over natural numbers. Expressions are given by variables, literals, and binary operations. Commands are given by `skip`, `if-then-else`, `while`, sequencing, and variable assignment. The global state of the program is given by a finite map from strings (variable names) to integers.

2 Interpreting the Language

The result of the computation is given by an inductive type:

```
Inductive result (T : Type) :=
| Done (x : T)
| Error
| NotYet.
```

The `Error` result is required because the global state is given by a *finite* map. This implies that variables need not be defined, and so a call to an unused variable may lead to the `Error` result. The result `NotYet` will be required to assign results to intermediate computations, since the presence of while loops imply that the `Imp` language is partial.

This type can be given monadic structure, where the bind function is called `mbind` [1], and the map `return : T -> result T` is given by `Done`. We can prove in Rocq that these definitions do indeed form a monad; they satisfy the following laws for Kleisli triples:

- `mbind return x = return x`

[1] https://plv.mpi-sws.org/coqdoc/stdpp/stdpp.base.html

Compiled By:
Bas Laarakker, Songlin Jia, Ulises Torrella

- `mbind f (return x) = f x`
- `mbind g (mbind f x) = mbind (g . f) x`

This also allows us to use Haskell-like notation when evaluating monadic functions. The following is notation for `mbind (\y -> g y) x`:

```
n ← x;
g
```

To evaluate commands in the language, we have to take care of the fact that the language `Imp` is partial due to the presence of while loops, whereas Rocq requires all functions to be total. To circumvent this, we use a fuel parameter to define a function which iterates a function `f`, given an initial value `x`.

```
Fixpoint iter {T}
    (f : (T -> result T) -> T -> result T)
    (k : nat) (x : T) : result T :=
  f (fun x' =>
      match k with
      | 0 => NotYet
      | S k' => iter f k' x'
      end) x.
```

The interesting case of the evaluation of commands is the case for `while`:

```
Fixpoint eval_com (c : com) (k : nat) (s : state) : result state :=
  match c with
  ...
| While e c =>
      let f eval_while s' : result state :=
        n ← eval_expr e s';
        if bool_decide (n = 0) then
          Done s'
        else
          s'' ← eval_com c k s';
          eval_while s''
      in
      iter f k s
```

Here, `eval_while` is the *continuation* passed to `f`, which is in turn passed to `iter`. That is, in evaluating `While e c`, if `e` does not evaluate to `0` (false), the command `c` is evaluated a single

time in state `s'`, resulting in a new state `s''`. The recursive call is the given by the continuation `eval_while`, which evaluates the remaining loop in `s''`. Giving the function `f` to `iter` with some amount of fuel `k` results in a term `Done s'` if the loop finishes within `k` iterations, and `NotYet` otherwise.

3 Relational Reasoning

Because `Imp` programs need not terminate, we define an ordering on the type `Result T` that signifies how well-defined a result of a program is. Intuitively, we have the following

```
x ⊑ y <-> x = NotYet \/ x = y.
```

We can prove that `iter` is monotone with respect to this ordering, which allows us to show that the evaluation of programs (`eval_com`) is monotone with respect to this ordering. To do so, we need a *generalized* induction hypothesis, which is parametrized by the state the computation is performed in.