Formal Verification of Monadic Computations: Lecture 1

Steve Zdancewic

27 June 2022

Motivation and Challenges

Formal verification and Coq can be applied to real-life code that are safety-critical. However, these systems involve complex specifications and behaviors that are difficult to model and verify. Verifying these systems at increased scale and low cost is also a challenge.

However there are several success stories in applying formal verification to software including `CompCert`, `CakeML`, `CertiKOS`, `Bedrock2`, `sel4`, `IronFleet`, `FSCQ`, and `DeepSpec`.

In these lectures, we will cover verifying monadic programs in Coq, which touches on many recurring themes in OPLSS and dependent type theory.

`Imp`: A simple imperative language

Has simple BNF grammar

```
c := skip | x := a | c;c | if b then c else c end | while b do c end
```

We can express this in Coq as an inductive type:

```
Inductive com : Type :=
  | CSkip
  | CAsgn (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).
```

Factorials as a simple example

We can implement the factorial function within `Imp`, as shown below. We observe that (after taking out junk code) we get that the program outputs the correct factorial; and provably so via Coq.

```
Definition fact_in_coq : com :=
  <{ Z := X;
     Y := 1;
     while Z  0 do
       Y := Y × Z;
       Z := Z - 1
     end }>.
```

We would then like to evaluate functions written in `Imp`. We want to do this semantically. If we want to write this denotationally, we can make each command in our syntax correspond to some mathematical function.

```
Fixpoint ceval_fun_no_while (st : state) (c : com) : state :=
  match c with
    | <{ skip }>
        st
    | <{ x := a }>
        (x !-> (aeval st a) ; st)
    | <{ c1 ; c2 }>
        let st' := ceval_fun_no_while st c1 in
        ceval_fun_no_while st' c2
    | <{ if b then c1 else c2 end}>
        if (beval st b)
          then ceval_fun_no_while st c1
          else ceval_fun_no_while st c2
    | <{ while b do c end }>
        st (* bogus *)
  end.
```

However, here we've run into an issue with the denotational approach because the `while` command can have infinite loops, but Coq wants a pure and total semantics! So instead we make a relational operational semantics using the rules below. The type of our semantics in Coq is `com x mem x mem`, a triple relation between a command, input memory, and final memory.

$$\frac{}{st = [skip] => st}\ E\_Skip$$

$$\frac{aeval\ st\ a = n}{st = [x := a] => (x!- > n; st)}\ E\_Asgn$$

$$\frac{st = [c_1] => st' \quad st' = [c_2] => st''}{st = [c_1; c_2] => st''}\ E\_Seq$$

$$\frac{beval\ st\ b = true \quad st = [c_1] => st'}{st = [if\ b\ then\ c_1\ else\ c_2\ end] => st'}\ E\_IfTrue$$

$$\frac{beval\ st\ b = false \quad st = [c_2] => st'}{st = [if\ b\ then\ c_1\ else\ c_2\ end] => st'}\ E\_IfFalse$$

$$\frac{beval\ st\ b = false}{st = [while\ b\ do\ c\ end] => st}\ E\_WhileFalse$$

$$\frac{beval\ st\ b = true \quad st = [c] => st' \quad st' = [while\ b\ do\ c\ end] => st''}{st = [while\ b\ do\ c\ end] => st''}\ E\_WhileTrue$$

```
Inductive ceval : com → state → state → Prop :=
  | E_Skip : forall st,
```

```
    st =[ skip ]=> st
| E_Asgn : forall st a n x,
    aeval st a = n →
    st =[ x := a ]=> (x !-> n ; st)
| E_Seq : forall c1 c2 st st' st'',
    st =[ c1 ]=> st' →
    st' =[ c2 ]=> st'' →
    st =[ c1 ; c2 ]=> st''
| E_IfTrue : forall st st' b c1 c2,
    beval st b = true →
    st =[ c1 ]=> st' →
    st =[ if b then c1 else c2 end]=> st'
| E_IfFalse : forall st st' b c1 c2,
    beval st b = false →
    st =[ c2 ]=> st' →
    st =[ if b then c1 else c2 end]=> st'
| E_WhileFalse : forall b st c,
    beval st b = false →
    st =[ while b do c end ]=> st
| E_WhileTrue : forall st st' st'' b c,
    beval st b = true →
    st =[ c ]=> st' →
    st' =[ while b do c end ]=> st'' →
    st =[ while b do c end ]=> st''

where "st =[ c ]=> st'" := (ceval c st st').
```

We observe a couple drawbacks. The first is that this semantics is *partial*, as in there is no `mem` for a diverging program. It is not *structural*, as some rules are defined in terms of themselves. It is also not *executable*, as we can't extract the semantics as a program to be executed. And finally, it is a *deep* embedding so we can't reuse meta-level Coq functionality.

The question now becomes: can we do it differently? Unsurprisingly, yes.

Enter the Monad

A Monad is a datatype that can represent some kind of computational behavior. A monad supports a "well-behaved" notion of sequential composition. Some examples are mutable state, exceptions, nondeterminism, I/O, and divergence.

One of the biggest features of `Imp` was sequential composition. Intuitively what we do for a command `c1 ; c2` with input memory `st` is to let `st'` be the memory after `c1` then let `st''` be the memory after running `c2` on `st'`. Then `st''` is the final memory state. In fact we can write what we explained above as a program using `let` statements in Coq. However this doesn't work. We need to find an appropriate semantic domain. But before we do, more monads!

Consider the thought that an `Imp`-like factorial function would take in an initial memory state, change it up a bit, then return a memory state and our desired factorial. In essence, a function

$S \to S \times B$. This is exactly what we want in a monad! In particular, S is the monad.

There are some common operations associated with a state monad. One of them, `bind`, generalizes `let`. `bind` explicitly passes state through the monad. `k` can be seen as a continuation.

```
Definition state_bind {S A B} (m : state S A) (k:A → state S B) : state S B :=
    fun s => let (s', a) := m s in k a s'.
```

Another operation, `ret`, lifts a pure computation that doesn't change the state of the program.

```
Definition state_ret {S A} : A -> state S A :=
    fun (a:A) (s:S) => (s, a).
```

`get` and `put` operations work generically on any state. `get` returns the current value of the state, leaving the state unchanged. `put` updates the state, returning the trivial unit value.

```
Definition get {S} : state S S :=
    fun s => (s, s).

Definition put {S} (s:S) : state S unit :=
    fun _ => (s, tt).
```

Monads, generally

A Monad $M : Type \to Type$ supports:

- A function *ret* to embed pure computations
- A function *bind* to sequence computations

Some examples include identity, options supporting fail, and nondeterminism supporting choice.

Monads have to follow some laws. In particular, a well-behaved notion of sequential composition. Consider that these two programs should be equal:

```
(c1 ; c2) ; c3

c1 ; (c2 ; c3)
```

We can easily visualize this using nested `let` statements, and it's not too hard to generalize this mental image using `bind`s. We can even just substitute.

The laws a monad must comply with are:

1. Bind associativity.
2. Ret is left unit.
3. Ret is right unit.

```
Class MonadLaws {M} `{Monad M} := {
    bind_associativity :
    forall A B C (ma : M A) (kb : A -> M B) (kc : B -> M C),
      x <- (y <- ma ;; kb y) ;; kc x
      =
      y <- ma ;; x <- kb y ;; kc x

  ; bind_ret_l :
    forall A B (a : A) (kb : A -> M B),
      x <- ret a ;; kb x
      =
      kb a

  ; bind_ret_r :
    forall A (m : M A),
      x <- m ;; ret x
      =
      m
  }.
```

Monad Equivalences

Now that we've introduced monads, it's interesting to consider what it means for two monadic computations to be equivalent. We can define a monad equivalence as a relation between two monads. This relation should have the usual properties of equivalence relations: that is it should be reflexive, symmetric, and transitive. However, we should also have a property that *binds be proper*; that is, the bind operation respects Monad equivalence. This can be expressed as maintaining the logical relation. We can then define a monad with a particular notion of equivalence as:

```
Class MonadLaws M `{Monad M} `{EqM M} := {
    bind_associativity :
    forall A B C (ma : M A) (kb : A -> M B) (kc : B -> M C),
      eqM                                  (* <---- NEW! *)
      (x <- (y <- ma ;; kb y) ;; kc x)
      (y <- ma ;; x <- kb y ;; kc x)

  ; bind_ret_l :
    forall A B (a : A) (kb : A -> M B),
      eqM
      (x <- ret a ;; kb x)
      (kb a)

  ; bind_ret_r :
    forall A (m : M A),
      eqM
      (x <- m ;; ret x)
        m
```

```
  (* NEW! *)
  ; Proper_bind : forall {A B},
      @Proper (M A -> (A -> M B) -> M B)
              (eqM ==> pointwise_relation _ eqM ==> eqM) bind
  }.
```

Important! This generalization to use eqM for equality is the stepping stone to more powerful relational reasoning principles.

Extensible Semantics and the `Free` Monad

We'll eventually talk about handling loops using coinduction and interaction trees, but first we must learn about free monads.

Our motivating example is the expression problem where we want to define a datatype by cases while still being able to both add new cases and define new functions over the datatype without recompiling existing code.

A free monad is parameterized by a function $E : Type \to Type$ and a return type $R : Type$. Essentially we say that "every operation knows what kind of value it computes". E can be seen as the type of events. `op : E X` represents an operation that will return a value of type `X` which can be fed into the continuation `k`. This free monad allows us to support operations are generic using dependent types.

```
Inductive FFree (E : Type -> Type) (R : Type) : Type :=
| Ret (x : R)
| Do {X} (op : E X) (k : X -> FFree E R).
```

We can then define sequential composition for the free monad. This sequential composition function is a bind function while the return function is a monadic return function. As a consequence, our free monad is in fact a monad.

```
Fixpoint seq {E X Y} (e : FFree E X) (k : X -> FFree E Y) : FFree E Y :=
  match e with
  | Ret x   => k x
  | Do op h => Do op (fun n => seq (h n) k)
  end.

#[export] Instance FFreeM {E} : Monad (FFree E) :=
{|
    ret := @Ret E
  ; bind := @seq E
|}.
```