SOFTWARE FOUNDATIONS

VOLUME 1: LOGICAL FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

IMPCEVALFUN

AN EVALUATION FUNCTION FOR IMP

We saw in the Imp chapter how a naive approach to defining a function representing evaluation for Imp runs into difficulties. There, we adopted the solution of changing from a functional to a relational definition of evaluation. In this optional chapter, we consider strategies for getting the functional approach to work.

A Broken Evaluator

```
Require Import Coq.omega.Omega.
Require Import Coq.Arith.Arith.
Require Import Imp Maps.
```

Here was our first try at an evaluation function for commands, omitting `WHILE`.

```
Fixpoint ceval_step1 (st : state) (c : com) : state :=
  match c with
    | SKIP ⇒
        st
    | l ::= a1 ⇒
        st & { l --> (aeval st a1)}
    | c1 ;; c2 ⇒
        let st' := ceval_step1 st c1 in
        ceval_step1 st' c2
    | IFB b THEN c1 ELSE c2 FI ⇒
        if (beval st b)
          then ceval_step1 st c1
          else ceval_step1 st c2
    | WHILE b1 DO c1 END ⇒
        st (* bogus *)
  end.
```

As we remarked in chapter Imp, in a traditional functional programming language like ML or Haskell we could write the WHILE case as follows:

```
  | WHILE b1 DO c1 END => if (beval st b1) then ceval_step1 st (c1;;
      WHILE b1 DO c1 END) else st
```

Coq doesn't accept such a definition (`Error: Cannot guess decreasing argument of fix`) because the function we want to define is not guaranteed to terminate. Indeed, the changed `ceval_step1` function applied to the `loop` program from `Imp.v` would never terminate. Since Coq is not just a functional programming language, but also a consistent logic, any potentially non-terminating function needs to be rejected. Here is an invalid(!) Coq program showing what would go wrong if Coq allowed non-terminating recursive functions:

```
Fixpoint loop_false (n : nat) : False := loop_false n.
```

That is, propositions like `False` would become provable (e.g., `loop_false 0` would be a proof of `False`), which would be a disaster for Coq's logical consistency.

Thus, because it doesn't terminate on all inputs, the full version of `ceval_step1` cannot be written in Coq — at least not without one additional trick...

A Step-Indexed Evaluator

The trick we need is to pass an *additional* parameter to the evaluation function that tells it how long to run. Informally, we start the evaluator with a certain amount of "gas" in its tank, and we allow it to run until either it terminates in the usual way *or* it runs out of gas, at which point we simply stop evaluating and say that the final result is the empty memory. (We could also say that the result is the current state at the point where the evaluator runs out of gas — it doesn't really matter because the result is going to be wrong in either case!)

```
Fixpoint ceval_step2 (st : state) (c : com) (i : nat) : state :=
  match i with
  | O ⇒ { --> 0 }
  | S i' ⇒
    match c with
      | SKIP ⇒
          st
      | l ::= a1 ⇒
          st & { l --> (aeval st a1) }
      | c1 ;; c2 ⇒
          let st' := ceval_step2 st c1 i' in
          ceval_step2 st' c2 i'
      | IFB b THEN c1 ELSE c2 FI ⇒
          if (beval st b)
            then ceval_step2 st c1 i'
            else ceval_step2 st c2 i'
      | WHILE b1 DO c1 END ⇒
```

```
            if (beval st b1)
            then let st' := ceval_step2 st c1 i' in
                 ceval_step2 st' c i'
            else st
      end
    end.
```

Note: It is tempting to think that the index `i` here is counting the "number of steps of evaluation." But if you look closely you'll see that this is not the case: for example, in the rule for sequencing, the same `i` is passed to both recursive calls. Understanding the exact way that `i` is treated will be important in the proof of `ceval__ceval_step`, which is given as an exercise below.

One thing that is not so nice about this evaluator is that we can't tell, from its result, whether it stopped because the program terminated normally or because it ran out of gas. Our next version returns an `option state` instead of just a `state`, so that we can distinguish between normal and abnormal termination.

```
Fixpoint ceval_step3 (st : state) (c : com) (i : nat)
                     : option state :=
  match i with
  | O ⇒ None
  | S i' ⇒
    match c with
      | SKIP ⇒
          Some st
      | l ::= a1 ⇒
          Some (st & { l --> (aeval st a1) })
      | c1 ;; c2 ⇒
          match (ceval_step3 st c1 i') with
          | Some st' ⇒ ceval_step3 st' c2 i'
          | None ⇒ None
          end
      | IFB b THEN c1 ELSE c2 FI ⇒
          if (beval st b)
            then ceval_step3 st c1 i'
            else ceval_step3 st c2 i'
      | WHILE b1 DO c1 END ⇒
          if (beval st b1)
          then match (ceval_step3 st c1 i') with
               | Some st' ⇒ ceval_step3 st' c i'
               | None ⇒ None
               end
          else Some st
    end
  end.
```

We can improve the readability of this version by introducing a bit of auxiliary notation to hide the plumbing involved in repeatedly matching against optional states.

```
Notation "'LETOPT' x <== e1 'IN' e2"
   := (match e1 with
```

```
          | Some x ⇒ e2
          | None ⇒ None
        end)
   (right associativity, at level 60).

Fixpoint ceval_step (st : state) (c : com) (i : nat)
                    : option state :=
  match i with
  | O ⇒ None
  | S i' ⇒
    match c with
      | SKIP ⇒
          Some st
      | l ::= a1 ⇒
          Some (st & { l --> (aeval st a1)})
      | c1 ;; c2 ⇒
          LETOPT st' <== ceval_step st c1 i' IN
          ceval_step st' c2 i'
      | IFB b THEN c1 ELSE c2 FI ⇒
          if (beval st b)
            then ceval_step st c1 i'
            else ceval_step st c2 i'
      | WHILE b1 DO c1 END ⇒
          if (beval st b1)
          then LETOPT st' <== ceval_step st c1 i' IN
               ceval_step st' c i'
          else Some st
    end
  end.

Definition test_ceval (st:state) (c:com) :=
  match ceval_step st c 500 with
  | None ⇒ None
  | Some st ⇒ Some (st X, st Y, st Z)
  end.

(* Compute
     (test_ceval { --> 0 }
         (X ::= 2;;
          IFB (X <= 1)
            THEN Y ::= 3
            ELSE Z ::= 4
          FI)).
   ====>
      Some (2, 0, 4)   *)
```

Exercise: 2 stars, recommended (pup_to_n)

Write an Imp program that sums the numbers from `1` to `X` (inclusive: `1 + 2 + ... + X`) in the variable `Y`. Make sure your solution satisfies the test that follows.

```
Definition pup_to_n : com
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

```
(*

Example pup_to_n_1 :
  test_ceval {X --> 5} pup_to_n
  = Some (0, 15, 0).
Proof. reflexivity. Qed.
*)
```

□

Exercise: 2 stars, optional (peven)

Write a `While` program that sets `Z` to `0` if `X` is even and sets `Z` to `1` otherwise. Use `ceval_test` to test your program.

```
(* FILL IN HERE *)
```

□

Relational vs. Step-Indexed Evaluation

As for arithmetic and boolean expressions, we'd hope that the two alternative definitions of evaluation would actually amount to the same thing in the end. This section shows that this is the case.

```
Theorem ceval_step__ceval: ∀ c st st',
      (∃ i, ceval_step st c i = Some st') →
      c / st \\ st'.
```

+

Exercise: 4 stars (ceval_step__ceval_inf)

Write an informal proof of `ceval_step__ceval`, following the usual template. (The template for case analysis on an inductively defined value should look the same as for induction, except that there is no induction hypothesis.) Make your proof communicate the main ideas to a human reader; do not simply transcribe the steps of the formal proof.

```
(* FILL IN HERE *)
```

□

```
Theorem ceval_step_more: ∀ i1 i2 st st' c,
  i1 ≤ i2 →
  ceval_step st c i1 = Some st' →
  ceval_step st c i2 = Some st'.
Proof.
induction i1 as [|i1']; intros i2 st st' c Hle Hceval.
  - (* i1 = 0 *)
    simpl in Hceval. inversion Hceval.
  - (* i1 = S i1' *)
    destruct i2 as [|i2']. inversion Hle.
    assert (Hle': i1' ≤ i2') by omega.
    destruct c.
```

```
    + (* SKIP *)
      simpl in Hceval. inversion Hceval.
      reflexivity.
    + (* ::= *)
      simpl in Hceval. inversion Hceval.
      reflexivity.
    + (* ;; *)
      simpl in Hceval. simpl.
      destruct (ceval_step st c1 i1') eqn:Heqst1'o.
      * (* st1'o = Some *)
        apply (IHi1' i2') in Heqst1'o; try assumption.
        rewrite Heqst1'o. simpl. simpl in Hceval.
        apply (IHi1' i2') in Hceval; try assumption.
      * (* st1'o = None *)
        inversion Hceval.

    + (* IFB *)
      simpl in Hceval. simpl.
      destruct (beval st b); apply (IHi1' i2') in Hceval;
        assumption.

    + (* WHILE *)
      simpl in Hceval. simpl.
      destruct (beval st b); try assumption.
      destruct (ceval_step st c i1') eqn: Heqst1'o.
      * (* st1'o = Some *)
        apply (IHi1' i2') in Heqst1'o; try assumption.
        rewrite → Heqst1'o. simpl. simpl in Hceval.
        apply (IHi1' i2') in Hceval; try assumption.
      * (* i1'o = None *)
        simpl in Hceval. inversion Hceval. Qed.
```

Exercise: 3 stars, recommended (ceval__ceval_step)

Finish the following proof. You'll need `ceval_step_more` in a few places, as well as some basic facts about ≤ and `plus`.

```
Theorem ceval__ceval_step: ∀ c st st',
      c / st \\ st' →
      ∃ i, ceval_step st c i = Some st'.
Proof.
  intros c st st' Hce.
  induction Hce.
  (* FILL IN HERE *) Admitted.
```

□

```
Theorem ceval_and_ceval_step_coincide: ∀ c st st',
      c / st \\ st'
  ↔ ∃ i, ceval_step st c i = Some st'.
Proof.
  intros c st st'.
  split. apply ceval__ceval_step. apply ceval_step__ceval.
Qed.
```

Determinism of Evaluation Again

Using the fact that the relational and step-indexed definition of evaluation are the same, we can give a slicker proof that the evaluation *relation* is deterministic.

```
Theorem ceval_deterministic' : ∀ c st st1 st2,
     c / st \\ st1 →
     c / st \\ st2 →
     st1 = st2.
```

+