CS720

Logical Foundations of Computer Science

Lecture 12: Formalizing an imperative language

Tiago Cogumbreiro

Summary

- Generalizing code transformation
- Functions-as-relations versus functions
- Formalizing the semantics of an imperative language
- Generating code from Coq

`IndProp.v`

Due Thursday October 11, 11:59pm EST

`Logic.v`

Due Friday October 12, 11:59pm EST

`Imp.v`

Due Thursday October 18, 11:59pm EST

Recap functions as relations (1/2)

- What is the signature of the proposition that represents `plus`?

```
plus: nat → nat → nat
```

Recap functions as relations (1/2)

What is the signature of the proposition that represents `plus`?

```
plus: nat → nat → nat
```

```
Plus: nat → nat → nat → Prop
```

Recap functions as relations (2/2)

How do we represent `plus` as a proposition?

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | 0 ⇒ m
  | S p ⇒ S (plus p m)
  end.
```

Recap functions as relations (2/2)

How do we represent `plus` as a proposition?

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | 0 => m
  | S p => S (plus p m)
  end.
```

```
Induction Plus: nat → nat → nat → Prop :=
| plus_0: forall n, Plus 0 n n
| plus_n: forall n m o,
  Plus n m o →
  Plus (S n) m (S o).
```

$$\frac{}{0 + n = n} \qquad \frac{n + m = o}{\mathrm{S}(n) + m = \mathrm{S}(o)}$$

Recall optimize_0plus

```
Fixpoint optimize_0plus (a:aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | APlus (ANum 0) e2 ⇒ optimize_0plus e2
  | APlus e1 e2 ⇒ APlus (optimize_0plus e1) (optimize_0plus e2)
  | AMinus e1 e2 ⇒ AMinus (optimize_0plus e1) (optimize_0plus e2)
  | AMult e1 e2 ⇒ AMult (optimize_0plus e1) (optimize_0plus e2)
  end.
```

optimize_0plus as a relation

```
Inductive Opt_0plus: aexp → aexp → Prop :=
(* Optmize *)
| opt_0plus_do: forall a, Opt_0plus (APlus (ANum 0) a) a
(* No optimization *)
| opt_0plus_skip: forall a1 a2, a1 <> ANum 0 → Opt_0plus (a1 + a2) (a1 + a2)
(* Recurse *)
| opt_0plus_plus:
  forall a1 a2 a1' a2',
  Opt_0plus a1 a1' →
  Opt_0plus a2 a2' →
  Opt_0plus (APlus a1 a2) (APlus a1 a2')
| opt_0plus_minus: forall a1 a2 a1' a2',
  Opt_0plus a1 a1' → Opt_0plus a2 a2' → Opt_0plus (AMinus a1 a2) (AMinus a1' a2')
| opt_0plus_mult: forall a1 a2 a1' a2',
  Opt_0plus a1 a1' → Opt_0plus a2 a2' → Opt_0plus (AMult a1 a2) (AMult a1' a2').
```

How can we generalize the optimization step?

Generalizing optimizations

```
Inductive Opt (O : aexp → aexp → Prop) : aexp → aexp → Prop :=
(* No optimization *)
| opt_skip : forall a, (forall a', ~ O a a') → Opt O a a
(* Optimize code *)
| opt_do : forall a a', O a a' → Opt O a a'
(* Recurse *)
| opt_plus : forall a1 a2 a1' a2' : aexp,
             Opt O a1 a1' →
             Opt O a2 a2' → Opt O (a1 + a2) (a1' + a2')
| opt_minus : forall a1 a2 a1' a2' : aexp,
             Opt O a1 a1' →
             Opt O a2 a2' → Opt O (a1 - a2) (a1' - a2')
| opt_mult : forall a1 a2 a1' a2' : aexp,
             Opt O a1 a1' →
             Opt O a2 a2' → Opt O (a1 * a2) (a1' * a2').
```

Generalizing Soundness

```
Definition IsSound (O:aexp → aexp → Prop) :=
  forall a a',
  O a a' →
  forall st,
  aeval st a = aeval st a'.

Theorem opt_sound:
  forall O : aexp → aexp → Prop,
  IsSound O →
  IsSound (Opt O).
(* Show that [optimize_0plus] is sound *)
Inductive MyOpt: aexp → aexp → Prop :=
| my_opt_def: forall (a:aexp), MyOpt (0 + a) a.

Theorem my_opt_sound: IsSound (Opt MyOpt).
```

How to write a functional version of `Opt`?

A functional version of Opt

```
Fixpoint opt (f : aexp → option aexp) (a:aexp) : aexp :=
match f a with
| Some a ⇒ a (* Optimize step *)
| None ⇒
  match a with
  | APlus a1 a2 ⇒ opt f a1 + opt f a2     (* Recurse *)
  | AMinus a1 a2 ⇒ opt f a1 - opt f a2
  | AMult a1 a2 ⇒ opt f a1 * opt f a2
  | _ ⇒ a                                 (* Skip *)
  end
end.
```

Notice how `option` encodes the fact that the proposition may/may-not hold.

Proving opt_func soundness

```
Definition IsFuncSound f :=
  forall a a',
    f a = Some a' →
    forall st,
    aeval st a = aeval st a'.

Theorem opt_func_sound:
  forall f : aexp → option aexp,
  IsFuncSound f →
  forall (a : aexp) (st : state),
  aeval st a = aeval st (opt f a).
```

On functions as relations

- Notice how it was simpler to prove the same result using the inductive definition. Why?

On functions as relations

Notice how it was simpler to prove the same result using the inductive definition. Why?

- Functions-as-relations include an inductive principle *(Proof by induction on the derivation tree.)*
- Functions-as-relations are more expressive *(eg, representing non-terminating behaviors.)*
- Functions can use Coq's evaluation power *(recall proof by reflection, lecture 10)*
- Functions can be translated automatically into OCaml/Haskell *(next lecture)*

Abstract syntax

```
c ::= SKIP
    | x ::= a
    | c ;; c
    | IFB b THEN c ELSE c FI
    | WHILE b DO c END
```

```
Inductive com : Type :=
  | CSkip : com
  | CAss : string → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com.
```

The factorial of X:

```
Z ::= X;;
Y ::= 1;;
WHILE ! (Z = 0) DO
  Y ::= Y * Z;;
  Z ::= Z - 1
END
```

Inductive evaluation

```
Reserved Notation "c1 '/' st '\\' st'" (at level 40, st at level 39).

Inductive ceval : com → state → state → Prop :=
  | E_Skip : forall st, SKIP / st \\ st
  | E_Ass  : forall st a1 n x,
      aeval st a1 = n →
      (x ::= a1) / st \\ st & { x ⟶ n }
  | E_Seq : forall c1 c2 st st' st'',
      c1 / st  \\ st' →
      c2 / st' \\ st'' →
      (c1 ;; c2) / st \\ st''
  | E_IfTrue : forall st st' b c1 c2,
      beval st b = true →
      c1 / st \\ st' →
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_IfFalse : forall st st' b c1 c2,
      beval st b = false →
      c2 / st \\ st' →
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_WhileFalse : forall b st c,
      beval st b = false →
      (WHILE b DO c END) / st \\ st
  | E_WhileTrue : forall st st' st'' b c,
      beval st b = true →
      c / st \\ st' →
      (WHILE b DO c END) / st' \\ st'' →
      (WHILE b DO c END) / st \\ st''
  where "c1 '/' st '\\' st'" := (ceval c1 st st').
```

$$\frac{}{\texttt{SKIP} \mathbin{/} s \mathbin{\backslash\backslash} s}$$

$$\frac{\mathrm{aeval}(s, a_1) = n}{x \mathrel{::=} a_1 \mathbin{/} s \mathbin{\backslash\backslash} s \mathbin{\&} \{x \to n\}}$$

$$\frac{c_1 \mathbin{/} s_1 \mathbin{\backslash\backslash} s_2 \qquad c_2 \mathbin{/} s_2 \mathbin{\backslash\backslash} s_3}{c_1 ;; c_2 \mathbin{/} s_1 \mathbin{\backslash\backslash} s_3}$$

$$\frac{\mathrm{beval}(s, b) = \top \qquad c_1 \mathbin{/} s_1 \mathbin{\backslash\backslash} s_2}{\texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI} \mathbin{/} s_1 \mathbin{\backslash\backslash} s_2}$$

$$\frac{\mathrm{beval}(s, b) = \bot \qquad c_2 \mathbin{/} s_1 \mathbin{\backslash\backslash} s_2}{\texttt{IFB}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI} \mathbin{/} s_1 \mathbin{\backslash\backslash} s_2}$$

$$\frac{\mathrm{beval}(s, b) = \bot}{\texttt{WHILE}\ b\ \texttt{DO}\ c_1\ \texttt{END} \mathbin{/} s_1 \mathbin{\backslash\backslash} s_2}$$

$$\frac{\mathrm{beval}(s, b) = \top \qquad c \mathbin{/} s_1 \mathbin{\backslash\backslash} s_2 \qquad \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END} \mathbin{/} s_2 \mathbin{\backslash\backslash} s_3}{\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{END} \mathbin{/} s_1 \mathbin{\backslash\backslash} s_3}$$

Showing that ceval behaves like a function

```
Theorem ceval_deterministic: forall c st st1 st2,
     c / st \\ st1  →
     c / st \\ st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2.
```

`ImpCEvalFun.v`

(No homework.)

Non-terminating functions in Coq

Evaluating com programs (1^{st} try)

```
Fail Fixpoint ceval_f (st : state) (c : com) : state :=
    match c with
    | SKIP ⇒ st
    | x ::= a1 ⇒ st & { x ⟶ (aeval st a1) }
    | c1 ;; c2 ⇒ let st' := ceval_f st c1 in ceval_f st' c2
    | IFB b THEN c1 ELSE c2 FI ⇒
      if beval st b then ceval_f st c1 else ceval_f st c2
    | WHILE b DO c END ⇒ if beval st b
      then let st' := ceval_f st c in ceval_f st' (WHILE b DO c END)
      else st
  end.
```

> How to work around the termination checker?

Evaluating com programs (2nd try)

```
Fixpoint ceval_f (st : state) (c : com) (i : nat) : state :=
match i with
| O ⇒ st (* no more fuel *)
| S i' ⇒
  match c with
   | SKIP ⇒ st
   | x ::= a1 ⇒ st & { x ⟶ (aeval st a1) }
   | c1 ;; c2 ⇒ let st' := ceval_f st c1 i' in ceval_f st' c2 i'
   | IFB b THEN c1 ELSE c2 FI ⇒
     if beval st b then ceval_f st c1 i' else ceval_f st c2 i'
   | WHILE b DO c END ⇒ if beval st b
     then let st' := ceval_f st c i' in ceval_f st' (WHILE b DO c END) i'
     else st
    end
  end.
```

How do we distinguish between running out of fuel and terminating?

Evaluating com programs (3^{rd} try)

```
Fixpoint ceval_f (st : state) (c : com) (i : nat) : option state :=
match i with
| O ⇒ None (* no more fuel *)
| S i' ⇒
  match c with
  | SKIP ⇒ Some st
  | x ::= a1 ⇒ Some (st & { x ⟶ (aeval st a1) })
  | c1 ;; c2 ⇒
     match ceval_f st c1 i' with
     | Some st' ⇒ ceval_f st' c2 i' | None ⇒ None end
  | IFB b THEN c1 ELSE c2 FI ⇒
     if beval st b then ceval_f st c1 i' else ceval_f st c2 i'
  | WHILE b DO c END ⇒ if beval st b
    then match ceval_f st c i' with
      | Some st' ⇒ ceval_f st' (WHILE b DO c END) i'
      | None ⇒ None
    end else Some st
   end
  end.
```

Evaluating com programs (4th try)

```
Fixpoint ceval_f (st : state) (c : com) (i : nat) : option state :=
let seq o f := match o with | Some st ⇒ f st | None ⇒ None end in
match i with
| O ⇒ None
| S i' ⇒
  match c with
  | SKIP ⇒ Some st
  | x ::= a1 ⇒ Some (st & { x ⟶ (aeval st a1) })
  | c1 ;; c2 ⇒ seq (ceval_f st c1 i') (fun st' ⇒ ceval_f st' c2 i')
  | IFB b THEN c1 ELSE c2 FI ⇒
     if beval st b then ceval_f st c1 i' else ceval_f st c2 i'
  | WHILE b DO c END ⇒ if beval st b
    then seq (ceval_f st c i') (fun st' ⇒ ceval_f st' (WHILE b DO c END) i')
    else Some st
   end
  end.
```

`Extraction.v`

(No homework.)

Extracting types and functions

```
Require Import Imp.

Extraction Language Haskell.

Extraction aeval.
Extraction aexp.

Extraction Language OCaml.

Extraction aeval.
Extraction aexp. (* Prints the translated code. *)
(* Translates into file. *)
Extraction "imp1.ml" ceval_step.
```

Interacting with the generated code

```
(* impdriver.ml *)
let test s =
  print_endline s;
  let parse_res = parse (explode s) in
  (match parse_res with
  | NoneE _ -> print_endline ("Syntax error");
  | SomeE (c, _) ->
      let fuel = 1000 in
      match (ceval_step empty_state c fuel) with
      | None -> print_endline ("Still running after " ^ string_of_int fuel ^ " steps")
      | Some res ->
          print_endline (
              "Result: ["
            ^ string_of_int (res ['w']) ^ " "
            ^ string_of_int (res ['x']) ^ " "
            ^ string_of_int (res ['y']) ^ " "
            ^ string_of_int (res ['z']) ^ " ...]"));
  print_newline();
```

Running our interpreter

```
$ ocamlc -w -20 -w -26 -o impdriver imp.mli imp.ml impdriver.ml
$ ./impdriver
x:=1 ;; y:=2
Result: [0 1 2 0 ...]

true
Syntax error

SKIP
Result: [0 0 0 0 ...]

SKIP;;SKIP
Result: [0 0 0 0 ...]

WHILE true DO SKIP END
Still running after 1000 steps

x:=3
Result: [0 3 0 0 ...]

x:=3;; WHILE 0≤x DO SKIP END
Still running after 1000 steps

x:=3;; WHILE 1≤x DO y:=y+1;; x:=x-1 END
Result: [0 0 3 0 ...]
```

Summary

- Generalizing code transformation
- Functions-as-relations versus functions
- Formalizing the semantics of an imperative language
- Generating code from Coq