CS720

Logical Foundations of Computer Science

Lecture 11: Formalizing an expression language

Tiago Cogumbreiro

Today's objectives

Programming language theory

- Introduce imperative languages
- Show an implementation of an interpreter
- Show an implementation of a compiler

Coq / HW5 skills

- Represent functions as propositions
- Proof automation

Expected background

- You have seen programming language implementation (via CS450/CS451)

IMP

```
Z := X;
Y := 1;
while Z ≠ 0 do
  Y := Y * Z;
  Z := Z - 1
end
```

Formalizing a basic imperative language

IMP from the ground up

- Syntax
- Semantics (operational)
- Formalization

Syntax

What syntactic categories do we find in this program?

```
Z := X;
Y := 1;
while Z ≠ 0 do
  Y := Y * Z;
  Z := Z - 1
end
```

Syntax

What syntactic categories do we find in this program?

```
Z := X;
Y := 1;
while Z ≠ 0 do
  Y := Y * Z;
  Z := Z - 1
end
```

1. Arithmetic expressions
2. Boolean expressions
3. Commands (eg, assignments, loops)

Syntax of arithmetic

```
Inductive aexp : Type :=
  | ANum: nat → aexp
  | AId: string → aexp
  | APlus: aexp → aexp → aexp
  | AMinus: aexp → aexp → aexp
  | AMult: aexp → aexp → aexp.
```

$$a ::= n \mid x \mid a + a \mid a - a \mid a \times a$$

- A literal n, represented as `ANum`, example `ANum 3`
- A program variable x, represented as `AId`, example `AId "x"`
- Addition represented as `APlus`, example `APlus (ANum 1) (AId "x")` to denote $1 + x$
- Subtraction represented as `AMinus`
- Multiplication represented as `AMult`

Syntax of booleans

```
Inductive bexp : Type :=
  | BTrue                  (* BTrue : bexp                *)
  | BFalse                 (* BFalse: bexp                *)
  | BEq (a1 a2 : aexp)     (* BEq: aexp → aexp → bexp     *)
  | BNeq (a1 a2 : aexp)    (* BNeq: aexp → aexp → bexp    *)
  | BLe (a1 a2 : aexp)     (* BLe: aexp → aexp → bexp     *)
  | BGt (a1 a2 : aexp)     (* BGt: aexp → aexp → bexp     *)
  | BNot (b : bexp)        (* BNot: bexp → bexp           *)
  | BAnd (b1 b2 : bexp).   (* BAnd: bexp → bexp → bexp    *)
```

$$b ::= \texttt{true} \mid \texttt{false} \mid a = a \mid a \neq a \mid a \leq a \mid !b \mid b \& b$$

Syntax of commands

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

$$c ::= \mathbf{skip} \mid x := a \mid c; c \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{while}\ b\ \mathbf{do}\ c$$

How do we give meaning to a language?

We show how to run it.

(Operational Semantics)

CS450 in a hurry

Evaluating expressions with an ***interpreter***

> Interpreter: a program that executes an abstract syntax.

```
  Fixpoint aeval (st: state) (a: aexp) : nat :=
    match a with
    | ANum n ⇒ n
    | AId x ⇒ st x
    | APlus  a1 a2 ⇒ (aeval st a1) + (aeval st a2)
    | AMinus a1 a2 ⇒ (aeval st a1) - (aeval st a2)
    | AMult  a1 a2 ⇒ (aeval st a1) * (aeval st a2)
    end.

(* x + (2 * 3) *)
Goal aeval empty_st (APlus (AId "x") (AMult (ANum 2) (ANum 3))) = 6.
Proof.  reflexivity. Qed.
```

Function versus proposition

```
Fixpoint aeval (st:state) (a:aexp):=
match a with
| ANum n ⇒ n     (* E_ANum   *)
| AId x ⇒ st x   (* E_AId    *)
| APlus e1 e2 ⇒ (* E_APlus  *)
  let n1 = aeval st e1 in
  let n2 = aeval st e2 in
  n1 + n2
| AMinus e1 e2 ⇒(* E_AMinus *)
  let n1 = aeval st e1 in
  let n2 = aeval st e2 in
  n1 - n2
| AMult e1 e2 ⇒ (* E_AMult  *)
  let n1 = aeval st e1 in
  let n2 = aeval st e2 in
  n1 * n2
end.
```

```
Inductive aevalR (st:state): aexp → nat → Prop :=

| E_ANum (n : nat) : aevalR st (ANum n) n
| E_AId (x : string) : aevalR st (AId x) (st x)
| E_APlus (e1 e2 : aexp) (n1 n2 : nat) :
    aevalR st e1 n1 →
    aevalR st e2 n2 →
    aevalR st (APlus e1 e2) (n1 + n2)
| E_AMinus (e1 e2 : aexp) (n1 n2 : nat) :
    aevalR st e1 n1 →
    aevalR st e2 n2 →
    aevalR st (AMinus e1 e2) (n1 - n2)
| E_AMult (e1 e2 : aexp) (n1 n2 : nat) :
    aevalR st e1 n1 →
    aevalR st e2 n2 →
    aevalR st (AMult e1 e2) (n1 * n2).
```

Typesetting proposition

```
Inductive aevalR (st:state): aexp → nat → Prop :=
| E_ANum (n : nat) : aevalR st (ANum n) n
| E_AId (x : string) : aevalR st (AId x) (st x)
| E_APlus (e1 e2 : aexp) (n1 n2 : nat) :
    aevalR st e1 n1 →
    aevalR st e2 n2 →
    aevalR st (APlus e1 e2) (n1 + n2)
| E_AMinus (e1 e2 : aexp) (n1 n2 : nat) :
    aevalR st e1 n1 →
    aevalR st e2 n2 →
    aevalR st (AMinus e1 e2) (n1 - n2)
| E_AMult (e1 e2 : aexp) (n1 n2 : nat) :
    aevalR st e1 n1 →
    aevalR st e2 n2 →
    aevalR st (AMult e1 e2) (n1 * n2).
```

$$\frac{}{\sigma, n \Rightarrow n}$$

$$\frac{}{\sigma, x \Rightarrow \sigma(x)}$$

$$\frac{\sigma, e_1 \Rightarrow n_1 \qquad \sigma, e_2 \Rightarrow n_2}{\sigma, e_1 + e_2 \Rightarrow n_1 + n_2}$$

$$\frac{\sigma, e_1 \Rightarrow n_1 \qquad \sigma, e_2 \Rightarrow n_2}{\sigma, e_1 - e_2 \Rightarrow n_1 - n_2}$$

$$\frac{\sigma, e_1 \Rightarrow n_1 \qquad \sigma, e_2 \Rightarrow n_2}{\sigma, e_1 * e_2 \Rightarrow n_1 * n_2}$$

Proving correctness

```
Lemma aeval_iff_aevalR : forall st a n,
  aevalR st a n <-> aeval st a = n.
Proof.
```

From prop to function

```
Inductive ceval : state → com → state → Prop :=
| E_Skip : forall st,
    ceval st CSkip st
| E_Asgn  : forall st a n x,
    aevalR st a n →
    ceval st (CAsgn x a) (x !→ n ; st)
| E_Seq : forall c1 c2 st st' st'',
    ceval st  c1 st'  →
    ceval st' c2 st'' →
    ceval st (CSeq c1 c2) st''
| E_IfTrue : forall st st' b c1 c2,
    bevalR st b true →
    ceval st c1 st' →
    ceval st (CIf b c1 c2) st'
| E_IfFalse : forall st st' b c1 c2,
    bevalR st b false →
    ceval st c2 st' →
    ceval st (CIf b c1 c2) st'
```

From prop to function

```
| E_WhileFalse : forall b st c,
    bevalR st b false →
    ceval st (CWhile b c) st
| E_WhileTrue : forall st st' st'' b c,
    bevalR st b true →
    ceval st c st' →
    ceval st'(CWhile b c) st'' →
    ceval st (CWhile b c) st''
```

From prop to function

```
| E_WhileFalse : forall b st c,
    bevalR st b false →
    ceval st (CWhile b c) st
| E_WhileTrue : forall st st' st'' b c,
    bevalR st b true →
    ceval st c st' →
    ceval st'(CWhile b c) st'' →
    ceval st (CWhile b c) st''
```

This cannot be implemented directly as a Coq function!