Using Imp

Type Theory and Coq

Tom Salet

Radboud University Nijmegen

May 13, 2016

Imp – Previous lecture

Formal Coq definition – Arithmetic expressions

Arithmetic expressions

```
Inductive aexp : Type :=
  | ANum : nat -> aexp
  | APlus : aexp -> aexp -> aexp
  | AMinus : aexp -> aexp -> aexp
  | AMult : aexp -> aexp -> aexp.
```

Normal form

```
a ::= nat
    | a + a
    | a - a
    | a * a
```

Imp – Previous lecture

Formal Coq definition – Arithmetic expressions

Arithmetic expressions

```
Inductive aexp : Type :=
  | ANum : nat -> aexp
  | APlus : aexp -> aexp -> aexp
  | AMinus : aexp -> aexp -> aexp
  | AMult : aexp -> aexp -> aexp.
```

Evaluation

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

Normal form

```
a ::= nat
    | a + a
    | a - a
    | a * a
```

Imp – Previous lecture

Formal Coq definition – Boolean expressions

Boolean expressions

```
Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp -> aexp -> bexp
  | BLe : aexp -> aexp -> bexp
  | BNot : bexp -> bexp
  | BAnd : bexp -> bexp -> bexp.
```

Normal form

```
b ::= true
    | false
    | a = a
    | a <= a
    | not b
    | b and b
```

Imp – Previous lecture

Formal Coq definition – Boolean expressions

Boolean expressions

```
Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp -> aexp -> bexp
  | BLe : aexp -> aexp -> bexp
  | BNot : bexp -> bexp
  | BAnd : bexp -> bexp -> bexp.
```

Normal form

```
b ::= true
    | false
    | a = a
    | a <= a
    | not b
    | b and b
```

Evaluation

```
Fixpoint beval (b : bexp) : bool :=
  match b with
  | BTrue       => true
  | BFalse      => false
  | BEq a1 a2   => beq_nat (aeval a1) (aeval a2)
  | BLe a1 a2   => ble_nat (aeval a1) (aeval a2)
  | BNot b1     => negb (beval b1)
  | BAnd b1 b2  => andb (beval b1) (beval b2)
  end.
```

Imp – Previous lecture

Formal Coq definition – Commands

Commands

```
Inductive com : Type :=
  | CSkip : com
  | CAss : id -> aexp -> com
  | CSeq : com -> com -> com
  | CIf : bexp -> com -> com -> com
  | CWhile : bexp -> com -> com.
```

Normal form

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

Imp – Previous lecture

Formal Coq definition – Commands

Commands

```
Inductive com : Type :=
  | CSkip : com
  | CAss : id -> aexp -> com
  | CSeq : com -> com -> com
  | CIf : bexp -> com -> com -> com
  | CWhile : bexp -> com -> com.
```

Normal form

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

Evaluation

Relation, no function, `c1 / st ⇓ st'`:

```
Inductive ceval : com -> state -> state -> Prop
```

Contents

ImpParser

A parser for the `Imp` language

Contents

ImpParser

A parser for the `Imp` language

ImpCEvalFun

Defining `ceval` as *function*

Contents

ImpParser

A parser for the `Imp` language

ImpCEvalFun

Defining `ceval` as *function*

Extraction

Extracting into other languages, such as OCaml

ImpParser

Typical Imp program:

```
z := x;;
y := 1;;
WHILE not (z == 0) DO
  y := y * z;;
  z := z-1
END
```

ImpParser

Typical Imp program:

```
z := x;;
y := 1;;
WHILE not (z == 0) DO
  y := y * z;;
  z := z-1
END
```

Formalised in Coq:

```
Id 0 ::= AId (Id 1);;
Id 2 ::= ANum 1;;
WHILE BNot (BEq (AId (Id 0)) (ANum 0)) DO
  Id 2 ::= AMult (AId (Id 2)) (AId (Id 0));;
  Id 0 ::= AMinus (AId (Id 0)) (ANum 1)
END
```

ImpCEvalFun

First attempt

First try from last lecture

```
Fixpoint ceval_step1 (st : state)
      (c : com) : state :=
  match c with
    | SKIP =>
        st
    | l ::= a1 =>
        update 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.
```

ImpCEvalFun

Adding a step counter

```
Fixpoint ceval_step1 (st : state)
      (c : com) : state :=
  match c with
    | SKIP =>
        st
    | l ::= a1 =>
        update 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.
```

```
Fixpoint ceval_step2 (st : state) (c : com)
      (i : nat) : state :=
  match i with
  | O => empty_state
  | S i' =>
    match c with
      | SKIP =>
          st
      | l ::= a1 =>
          update 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.
```

ImpCEvalFun

Adding some error handling

```
Fixpoint ceval_step2 (st : state) (c : com) (i : nat) :        state :=
  match i with
  | O => empty_state
  | S i' =>
    match c with
      | SKIP =>      st
      | l ::= a1 =>      (update 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.
```

ImpCEvalFun

Adding some error handling

```
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 (update 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.
```

ImpCEvalFun

Improving readability

```
Notation "'LETOPT' x <== e1 'IN' e2"
   := (match e1 with
         | Some x => e2
         | None => None
       end)
   (right associativity, at level 60).
```

ImpCEvalFun

Defining the final ceval_step

```
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 (update 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.
```

ImpCEvalFun

Defining the final ceval_step

```
Fixpoint ceval_step2 (st : state) (c : com) (i : nat) :        state :=
  match i with
  | O => empty_state
  | S i' =>
    match c with
      | SKIP =>
                st
      | l ::= a1 =>
                (update 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.
```

ImpCEvalFun

Defining the final ceval_step

```
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 (update 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.
```

Summary

ImpParser

A parser for the `Imp` language

ImpCEvalFun

Command evaluation function with step counter

Extraction

Extracting into other languages, such as OCaml