Lecture 12: PCF Semantics

CSC 131
Kim Bruce

PCF Syntax & Semantics

```
e ::= x | n | true | false | succ | pred | iszero |
      if e then e else e | (fn x => e) | (e e) |
      rec x => e | let x = e1 in e2 end
```

- Substitution semantics proceeded by simplifying parse trees.
  - Function from trees to trees.
- Straightforward, but inefficient, so not commonly used.

PCF Syntax & Semantics with Environments

```
env:: string -> value

(0) (id, env) ⇒ env(id)

(1) (n, env) ⇒ n   for n an integer.

(2) (true, env) ⇒ true, (false, e) ⇒ false

(3) (error, env) ⇒ error

(4) (succ, env) ⇒ succ, similarly for other initial functions

    (b, env) ⇒ true     (e1, env) ⇒ v
(5) ------------------------------------
      (if b then e1 else e2, env) ⇒ v
```

More PCF Semantics

```
    (b, env) ⇒ false   (e2, env) ⇒ v
(6) -------------------------------
    (if b then e1 else e2, env) ⇒ v

    (e1, env) ⇒ succ    (e2, env) ⇒ n
(7) --------------------------------
         ((e1 e2), env) ⇒ (n+1)

(8) ...

(9) ...
```

Revised PCF Semantics

Closure(x,e,env)

```
(10)        ((fn x => e), env) ⇒ <fn x => e, env>

        (e1,env) ⇒ <fn x => e3, env'>   (e2, env) ⇒ v1
                  (e3, env'[v1/x]) ⇒ v
(11)    ----------------------------------------
                    ((e1 e2), env) ⇒ v
```

Thunk(rec x => e, env)

```
            (e, env[(rec x => e)/x]) ⇒ v
(12)        ----------------------------
              ((rec x => e), env) ⇒ v
```

Imperative Languages

Adding State For Assignment

$$\frac{(e_1, ev, s) \Rightarrow (m, s') \quad (e_2, ev, s') \Rightarrow (n, s'')}{(e_1 + e_2, ev, s) \Rightarrow (m+n, s'')}$$

$$\frac{(M, ev, s) \Rightarrow (v, s')}{(X := M, ev, s) \Rightarrow (v, s'[v / ev(X)])}$$

$$(\text{fn } x \Rightarrow M, ev, s) \Rightarrow (< \text{fn } x \Rightarrow M, ev >, s)$$

$$\frac{(f, ev, s) \Rightarrow (< \text{fn } x \Rightarrow M, ev' >, s'), \quad (N, ev, s') \Rightarrow (v, s''), \quad (M, ev'[v/X], s'') \Rightarrow (v', s''')}{(f(N), ev, s) \Rightarrow (v', s''')}$$

Summary of Operational Semantics

- Meaning of program is sequence of states go through during execution
- Useful for compiler writers, complexity analysis
- Ideal is abstract machine that is simple enough that it is impossible to misunderstand operation.
- Should be easy to map to any computer.

Axiomatic Semantics

- No model of computation.
- Specification of meaning via pre- and post-conditions:
  - {P} stats {Q}
  - If P is true before executing stats and computation halts, then Q will be true at end.

Axiomatic Rules

- Assignment axiom:
  - {P [expression / id]} `id := expression` {P}
  - Ex: {a+47 > 0} `x := a+47` {x > 0}
  - {x > 1} `x := x - 1` {x > 0}
- While rule:
  - If {P & B} `stats` {P}, then
    {P} `while B do stats` {P & not B}
  - P is *invariant* of loop.

Axiomatic Rules

- Composition rule:
  - If {P} `S1` {Q}, {R} `S2` {T}, and Q ⇒ R, then {P} `S1; S2` {T}
- Conditional rule:
  - If {P & B} `S1` {Q}, {P & not B} `S2` {Q},
    then {P} `if B then S1 else S2` {Q}
- Consequence rule:
  - If P ⇒ Q, R ⇒ T, and {Q} `S` {R}, then {P} `S` {T}

Correctness using Axioms & Rules

- Due to Bob Floyd & Tony Hoare
- Prove {precondition} `Prog` {postcondition}
- Usually work backwards from postcondition.

```
{Pre: exponent₀ >= 0}
base <- base₀
exponent <- exponent₀
ans <- 1
while exponent > 0 do
{assert:  ans * (base ** exponent) = base₀ ** exponent₀}
{           & exponent >= 0}
    if odd(exponent) then
            ans<- ans*base
            exponent <- exponent - 1
    else
            base <- base * base
            exponent <- exponent div 2
    end if
end while
{Post: exponent = 0 & ans = base₀ ** exponent₀}
```

Steps in Proof

- Show
  `ans * (base ** exponent) = base₀ ** exponent₀`
  `& exponent >= 0`
  is loop invariant
- Show postcondition follows from
  `(ans * (base ** exponent) = base₀ ** exponent₀`
  `& exponent >= 0) & not(exponent > 0)`
- Push invariant back to beginning of program.

Type Safety

- Is there any connection between type checking rules and semantics?
- If E |- e: T, what does that say about computation (e, *env*) ⇒ *v*?
- If E and *env* "correspond", then expect *v*: T

Typed PCF

- T :: = Int | Bool | T -> T
- Provide identifiers w/type when introduced.
- e ::= x | n | true | false | succ | pred | iszero |
  if e then e else e | (fn (x:T) => e) | (e e) |
  rec (x:T) => e *Ignore recursion for now!*

Type-checking Rules

E is *type environment*: identifiers → types

$E \vdash n$: Int, if n is an integer

$E \vdash$ true: Bool, $E \vdash$ false: Bool

$E \vdash$ succ: Int → Int, $E \vdash$ pred: Int → Int

$E \vdash$ iszero: Int → Bool

$E \vdash x: E(x)$

More Type-Checking Rules

$$\frac{E \vdash e: \text{Bool},\ \ E \vdash e_1: T,\ \ E \vdash e_2: T}{E \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : T}$$

$$\frac{E \vdash f: T \to U,\ \ E \vdash x: T}{E \vdash (f\ x): U}$$

$$\frac{E[x{:}T] \vdash \text{body}: U}{E \vdash (\text{fn } (x{:}T) \Rightarrow \text{body}): T \to U}$$

Computation Rules

(id, *env*) ⇒ *env*(id) (n, *env*) ⇒ *n*, for n an int

(true, *env*) ⇒ *true* (false, *env*) ⇒ *false*

(succ, *env*) ⇒ *succ* (pred, *env*) ⇒ *pred*

(iszero, *env*) ⇒ *iszero*

$$\frac{(b, env) \Rightarrow true,\ \ (e_1, env) \Rightarrow v}{(\text{if } b \text{ then } e_1 \text{ else } e_2, env) \Rightarrow v}$$

....

Computation Rules

((fn (x:T) => e), *env*) ⇒< fn (x:T) => e, *env* >

$$\frac{(e_1, env) \Rightarrow \langle \text{fn } (x{:}T) \Rightarrow e_3, env' \rangle \quad (e_2, env) \Rightarrow v_2,\ \ (e_3, env'[v_2/x]) \Rightarrow v}{((e_1\ e_2), env) \Rightarrow v}$$

Typing Values

$\vdash_v$ *n* : Int, for n an integer

$\vdash_v$ *true* : Bool

$\vdash_v$ *false* : Bool

$\vdash_v$ *succ* : Int → Int

$\vdash_v$ *pred* : Int → Int

$\vdash_v$ *iszero* : Int → Bool

Environment Compatibility

$$\frac{E' \vdash \text{fn } (x{:}T) \Rightarrow e{:}\ T \to U}{\vdash_v \langle \text{fn } (x{:}T) \Rightarrow e,\ env \rangle{:}\ T \to U}$$

for E' s.t E' and *env* are *compatible*

E and *env* are *compatible* iff

domain(E) = domain(*env*), and

for all x in domain(*env*), $\vdash_v$ *env*(x): E(x)

Safety

Theorem: (Subject Reduction) Let E and *env* be compatible environments.

Let e be a term of typed PCF. If E ⊢ e: T and (e,*env*) ⇒ *v*, then $\vdash_v$ *v*: T.

Proof: By induction on proof of E ⊢ e: T

Proof

<u>Conditional</u>: S'pose E ⊢ if b then e1 else e2: T because
E ⊢ b: boolean, E⊢e1: T, and E⊢e2: T.
Two cases depending on the evaluation of b.

<u>Case 1:</u> (b, *env*) ⇒ *true*.

Then if (e1,*env*) ⇒ *v*, it follows that
(if b then e1 else e2, *env*) ⇒ *v*.

By induction and E⊢e1: T, it follows that $\vdash_v$ *v*: T, which is all we need.

<u>Case 2</u>: (b, *env*) ⇒ *false* is similar.

Application case/*succ*:

Let e be (e1 e2) and E ⊢ (e1 e2): U because
E ⊢ e1: T → U and E ⊢ e2: T.
Suppose also that (e1, *env*) ⇒ *f* and (e2, *env*) ⇒ *v2*.

By induction, $\vdash_v f$: T → U and $\vdash_v$ *v2*: T.
Need to show final result has type U.

<u>Subcase</u>: *f* is *succ*. Then T = U = Int.
Because $\vdash_v$ *v2*: Int, *v2* must be an integer.
By the semantics rule, the result is *v2+1*, which has type Int, as desired, as E ⊢ (e1 e2): Int.

Closure case

<u>Subcase</u>: *f* is < fn (x:T) => e3, *env*' >
By induction, $\vdash_v$ < fn (x:T) => e3, *env*' >: T → U &
∃ E' s.t. E', *env*' compatible & E'⊢ fn (x:T) => e3: T → U

Spose (e3, *env*' [*v2*/x]) ⇒ *v*. Most show $\vdash_v$ *v*: U.
By above, must have had E'[x:T] ⊢ e3: U.

E' & *env*' compatible ⇒ E'[x:T] & *env*' [*v2*/x] compatible
So from (e3, *env*' [*v2*/x]) ⇒ *v*,
it follows by induction that $\vdash_v$ *v*: U.

Type Safety

- Errors have been made in type systems.
  - See examples in OO languages
- Need to verify that type system is consistent with semantics.
- Progress Lemma (*computations don't get stuck*) not shown here, but also important

Attributes of Variable

- Scope Done!
- Lifetime
- Location
- Value

Lifetime

- FORTRAN - all allocated statically - ∞
- Stack-based (C/C++/Java/Pascal/...)
  - local vbles/ parameters: method/procedure/block entry to exit
  - allocate space in activation record on run-time stack
- Heap allocated variable
  - lifetime independent of scope
- Static - global vbles or static vbles

Value & Location

- Sometimes referred to as l-value & r-value
  - x = x + 1 *What does each occurrence of x stand for?*
  - location normally bound when declaration processed
- Normally values change dynamically
  - if frozen at compilation then called constants
  - Java final variables frozen when declaration processed.
  - Java static final bound at compile time.

Aliases

- x and y are aliases if both refer to same location.
- If x, y are aliases then changes to x affect value of y.
- Java has uniform model where assignment is by "sharing", so create aliases.
- Language that mix are more confusing.
  - Common mistakes occur when not realize aliases. E.g, add elt to priority queue and then change it ...

Pointers

- "Pointers have been lumped with the goto statement as a marvelous way to create impossible to understand programs"
  - K & R, C Programming Language
- Problems
  - Dangling pointers -- leave pointer to recycled space
    - stack frame popped or recycled heap item
  - Dereference nil pointers or other illegal address
  - Unreachable garbage
  - in C: p+1 different from (int)p + 1