Lecture 13: PCF & Natural Semantics

CSC 131

Kim Bruce

PCF Semantics w/Environments

- Substitution slow & space consuming
- Can't handle terms w/free variables
- Environment allows to evaluate once.
- Meaning now separate set of values -- not just rewriting
- Meaning of function is closure, which carries around its environment of definition.

The Problem

- Program:
  - y = 4
  - f x = x + y
  - g (h) = let y = 5 in (h 2) + y
  - g(f)
- When evaluate (h 2), the needed y is out of scope!

Values of Answers

- Key difference w/ new interpreter
  - Update environment, not rewrite term!
  - Not destructive!
- Mutually recursive type definitions:

```
data Value = NUM Int | BOOL Bool | SUCC | PRED |
        ISZERO | CLOSURE (String, Term, Env) |
        THUNK (Term, Env) | ERROR (String, Value)
type Env = [(String, Value)]
```

Solving the Problem

- Program:
  - y = 4
  - f x = x + y
  - g (h) = let y = 5 in (h 2) + y
  - g(f)
- f *evaluates to* <fn x => x+y, [y->4]>
- g(f) *partially evaluates to* (h 2) + y *in environment where* env = [y->5, h-> <fn x => x+y, [y->4]>]

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

$$\frac{(e1, ev, s) \Rightarrow (m, s') \quad (e2, ev, s') \Rightarrow (n, s'')}{(e1 + e2, ev, s) \Rightarrow (m+n, s'')}$$

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

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

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

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 $\Rightarrow$ 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 $\Rightarrow$ Q, R $\Rightarrow$ 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: exponent_0 >= 0}
base <- base_0
exponent <- exponent_0
ans <- 1
while exponent > 0 do
{assert:  ans * (base ** exponent) = base_0 ** exponent_0}
{            & 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 = base_0 ** exponent_0}
```

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 ⊢ n: Int, if n is an integer

E ⊢ true: Bool,   E ⊢ false: Bool

E ⊢ succ: Int → Int,   E ⊢ pred: Int → Int

E ⊢ iszero: Int → Bool

E ⊢ x: E(x)