COSE212: Programming Languages

Lecture 8 — States

Hakjoo Oh
2015 Fall

Motivating Example

- How can we compute the number of times `f` has been called?

```
let f = proc (x) (x)
in (f (f 1))
```

- Does the following program work?

```
let counter = 0
in let f = proc (x) (let counter = counter + 1
                     in x)
   in let a = (f (f 1))
      in counter
```

- The language should support *effects*.
- Effects are implemented by introducing *memory* (*store*) and *locations* (*reference*).

Computational Effects

Programming languages support effects explicitly or implicitly.

- Explicit languages provide a clear account of allocation, dereference, and mutation of memory cells, e.g., ML.
- In implict languages, they are built-in, e.g., C and Java.

A Language with Explict References

$$
\begin{array}{rcl}
P & \to & E \\
E & \to & n \mid x \\
& \mid & E + E \mid E - E \\
& \mid & \texttt{zero?}\ E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
& \mid & \texttt{let}\ x = E\ \texttt{in}\ E \\
& \mid & \texttt{proc}\ x\ E \mid E\ E \\
& \mid & \texttt{ref}\ E \\
& \mid & !\ E \\
& \mid & E := E \\
& \mid & E; E
\end{array}
$$

- $\texttt{ref}\ E$ allocates a new location and store the value of E in it.
- $!\ E$ returns the contents of the location that E refers to.
- $E_1 := E_2$ changes the contents of the location (E_1) by the value of E_2.

Example 1

- ```
  let counter = ref 0
  in let f = proc (x) (counter := !counter + 1; !counter)
     in let a = (f 0)
        in let b = (f 0)
           in (a - b)
  ```
- ```
  let f = let counter = ref 0
          in proc (x) (counter := !counter + 1; !counter)
  in let a = (f 0)
     in let b = (f 0)
        in (a - b)
  ```
- ```
  let f = proc (x) (let counter = ref 0
                    in (counter := !counter + 1; !counter))
  in let a = (f 0)
     in let b = (f 0)
        in (a - b)
  ```
  ```

Example 2

We can make chains of references:

```
let x = ref (ref 0)
in (!x := 11; !(!x))
```

Semantics

Memory is modeled as a finite map from locations to values:

$$
\begin{array}{rcl}
Val & = & \mathbb{Z} + Bool + Procedure + Loc \\
Procedure & = & Var \times E \times Env \\
\rho \in Env & = & Var \to Val \\
\sigma \in Mem & = & Loc \to Val
\end{array}
$$

Semantics rules describe memory effects:

$$\rho, \sigma \vdash E \Rightarrow v, \sigma'$$

Semantics

Existing rules are enriched with stores:

$$\frac{}{\rho, \sigma \vdash n \Rightarrow n, \sigma} \qquad \frac{}{\rho, \sigma \vdash x \Rightarrow \rho(x), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 + E_2 \Rightarrow n_1 + n_2, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow 0, \sigma_1}{\rho, \sigma_0 \vdash \texttt{zero?}\ E \Rightarrow \mathit{true}, \sigma_1} \qquad \frac{\rho, \sigma_0 \vdash E \Rightarrow n, \sigma_1}{\rho, \sigma_0 \vdash \texttt{zero?}\ E \Rightarrow \mathit{false}, \sigma_1}\ n \neq 0$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \mathit{true}, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad [x \mapsto v_1]\rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_2}$$

$$\frac{}{\rho, \sigma \vdash \texttt{proc}\ x\ E \Rightarrow (x, E, \rho), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \vdash (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \qquad [x \mapsto v]\rho', \sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}$$

Semantics

Rules for new constructs:

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash \texttt{ref}\ E \Rightarrow l, [l \mapsto v]\sigma_1}\ l \notin \mathsf{Dom}(\sigma_1)$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow l, \sigma_1}{\rho, \sigma_0 \vdash !\ E \Rightarrow \sigma_1(l), \sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow l, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash E_1 := E_2 \Rightarrow v, [l \mapsto v]\sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

Example

$$\frac{}{\boldsymbol{\rho}, \boldsymbol{\sigma_0} \vdash \texttt{let x = ref (ref 0) in (!x := 11; !(!x))} \Rightarrow}$$

A Language with Implict References

$$\begin{array}{rcl} P & \rightarrow & E \\ E & \rightarrow & n \mid x \\ & \mid & E + E \mid E - E \\ & \mid & \texttt{zero?}\ E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\ & \mid & \texttt{let}\ x = E\ \texttt{in}\ E \\ & \mid & \texttt{proc}\ x\ E \mid E\ E \\ & \mid & \texttt{set}\ x = E \\ & \mid & E; E \end{array}$$

- Every variable is mutable (i.e., changeable).
- $\texttt{set}\ x = E$ change the contents of x by the value of E.
- Locations are created with each binding operation: call and let.

Examples

- ```
  let f = let count = 0
          in proc (x) (set count = count + 1; count)
  in let a = (f 0)
     in let b = (f 0)
        in a - b
  ```
- ```
  let f = proc (x)
           proc (y)
            (set x = x + 1; x - y)
  in ((f 44) 33)
  ```

Semantics

Every variable denotes a reference:

$$\begin{array}{rcl} \boldsymbol{Val} & = & \mathbb{Z} + \boldsymbol{Bool} + \boldsymbol{Procedure} \\ \boldsymbol{Procedure} & = & \boldsymbol{Var} \times \boldsymbol{E} \times \boldsymbol{Env} \\ \boldsymbol{\rho} \in \boldsymbol{Env} & = & \boldsymbol{Var} \to \boldsymbol{Loc} \\ \boldsymbol{\sigma} \in \boldsymbol{Mem} & = & \boldsymbol{Loc} \to \boldsymbol{Val} \end{array}$$

Semantics

$$\frac{}{\rho, \sigma \vdash n \Rightarrow n, \sigma} \qquad \frac{}{\rho, \sigma \vdash x \Rightarrow \sigma(\rho(x)), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 + E_2 \Rightarrow n_1 + n_2, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow 0, \sigma_1}{\rho, \sigma_0 \vdash \texttt{zero? } E \Rightarrow \mathit{true}, \sigma_1} \qquad \frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \mathit{true}, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v, \sigma_2}$$

$$\frac{}{\rho, \sigma \vdash \texttt{proc } x\ E \Rightarrow (x, E, \rho), \sigma} \qquad \frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash \texttt{set } x = E \Rightarrow v, [\rho(x) \mapsto v]\sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad [x \mapsto l]\rho, [l \mapsto v_1]\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let } x = E_1 \texttt{ in } E_2 \Rightarrow v, \sigma_2}\ l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{\begin{array}{c}\rho, \sigma_0 \vdash E_1 \vdash (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \\ [x \mapsto l]\rho', [l \mapsto v]\sigma_2 \vdash E \Rightarrow v', \sigma_3\end{array}}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \notin \mathrm{Dom}(\sigma_2)$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

Example

```
let f = let count = 0
        in proc (x) (set count = count + 1; count)
in let a = (f 0)
   in let b = (f 0)
      in a - b
```

Call-By-Value Parameter-Passing

What is the value of the following program?

```
let p = proc (x) (set x = 4)
in let a = 3
   in ((p a); a)
```

The call semantics:

$$\frac{\rho, \sigma_0 \vdash E_1 \vdash (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \qquad [x \mapsto l]\rho', [l \mapsto v]\sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \notin \mathrm{Dom}(\sigma_2)$$

Call-by-value parameter-passing:

- The formal parameter refers to a new location containing the value of the actual parameter.
- The most commonly used form of parameter-passing.

Call-By-Reference Parameter-Passing

The location of the caller's variable is passed, rather than the contents of the variable.

- Extend the syntax:

$$
\begin{array}{rcl}
E & \rightarrow & \vdots \\
 & | & E\ E \\
 & | & E\ \langle y \rangle
\end{array}
$$

- Extend the semantics:

$$
\frac{\rho, \sigma_0 \vdash E_1 \vdash (x, E, \rho'), \sigma_1 \qquad [x \mapsto \rho(y)]\rho', \sigma_1 \vdash E \Rightarrow v', \sigma_2}{\rho, \sigma_0 \vdash E_1\ \langle y \rangle \Rightarrow v', \sigma_2}
$$

Examples

- ```
  let p = proc (x) (set x = 4)
  in let a = 3
     in ((p <a>); a)
  ```
- ```
  let f = proc (x) (set x = 44)
  in let g = proc (y) (f <y>)
     in let z = 55
        in ((g <z>); z)
  ```
- ```
  let swap = proc (x) proc (y)
                let temp = x
                in (set x = y; set y = temp)
  in let a = 33
     in let b = 44
        in (((swap <a>) <b>); (a-b))
  ```
  ```

Variable Aliasing

More than one call-by-reference parameter may refer to the same location:

```
let b = 3
in let p = proc (x) proc (y)
            (set x = 4; y)
   in ((p <b>) <b>)
```

- A *variable aliasing* is created: x and y refer to the same location
- With aliasing, reasoning about program behavior is very difficult, because an assignment to one variable may change the value of another.

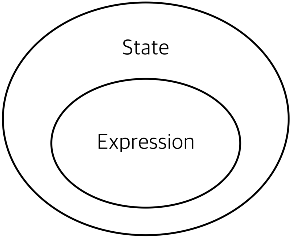
cf) Eager vs. Lazy Evaluation

```
letrec infinite-loop (x) = infinite-loop (x)
in let f = proc (x) (1)
   in  (f (infinite-loop 0))
```

- In eager evaluation, procedure arguments are completely evaluated before passing them to the procedure.
- In lazy evaluation, evaluation of arguments is delayed until it is needed by the procedure body.
- Shortcoming of lazy evaluation?

Summary

Our current language:



- Explicit and implicit supports for effects.
- Parameter-passing variations: call-by-value, call-by-reference