ENE4014: Programming Languages

Lecture 9 — Design and Implementation of PLs (5) Records, Pointers, and Garbage Collection

Woosuk Lee
2024 Spring

Review: Our Language So Far

Syntax:

$$
\begin{array}{rcl}
P & \to & E \\
E & \to & n \\
& | & x \\
& | & E + E \\
& | & \texttt{iszero}\ E \\
& | & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
& | & \texttt{let}\ x = E\ \texttt{in}\ E \\
& | & \texttt{proc}\ x\ E \\
& | & E\ E \\
& | & E\ \langle y \rangle \\
& | & x := E \\
& | & E; E
\end{array}
$$

Values:

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

Review: Semantics Rules

(Some rules omitted)

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

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

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

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

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

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

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

Plan

Extend the language with

- records (structured data),
- pointers, and
- memory management.

Records (Structured Data)

A record (i.e., `struct` in C) is a collection of named memory locations.

```
let student = { id := 201812, age := 20 }
in  student.id + student.age

let tree = { left := {}, v := 0, right := {} }
in  tree.right := { left := {}, v := 2, right := 3 }
```

cf) Arrays are also collections of memory locations, where the names of the locations are natural numbers.

Language Extension

Syntax:

$$\begin{array}{rcl} E & \to & \vdots \\ & \mid & \{\} \\ & \mid & \{\ x := E, y := E\ \} \\ & \mid & E.x \\ & \mid & E.x := E \end{array}$$

Values:

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

A record value r is a finite function (i.e., table):

$$\{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\}$$

Language Extension

Semantics:

$$\frac{}{\rho, \sigma \vdash \{\} \Rightarrow \cdot, \sigma}$$

$$\frac{\rho, \sigma \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2 \qquad l_1, l_2 \not\in Dom(\sigma_2)}{\rho, \sigma \vdash \{\ x := E_1, y := E_2\ \} \Rightarrow \{x \mapsto l_1, y \mapsto l_2\}, [l_1 \mapsto v_1, l_2 \mapsto v_2]\sigma_2}$$

$$\frac{\rho, \sigma \vdash E \Rightarrow r, \sigma_1}{\rho, \sigma \vdash E.x \Rightarrow \sigma_1(r(x)), \sigma_1}$$

$$\frac{\rho, \sigma \vdash E_1 \Rightarrow r, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma \vdash E_1.x := E_2 \Rightarrow v, [r(x) \mapsto v]\sigma_2}$$

Pointers

Let memory locations to be first-class values.

```
let x = 1 in
  let y = &x in
    *y := *y + 2

let x = { left := {}, v := 1, right := {} } in
  let y = &x.v
    *y := *y + 2

let f = proc (x) (*x := *x + 1) in
  let a = 1 in
    (f &a); a

let f = proc (x) (&x) in
  let p = (f 1) in
    *p := 2
```

Language Extension

Syntax:

$$\begin{array}{rcl} E & \to & \vdots \\ & | & \&x \\ & | & \&E.x \\ & | & *E \\ & | & *E := E \end{array}$$

Values:

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

Language Extension

Semantics:

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

$$\frac{\rho, \sigma \vdash E \Rightarrow r, \sigma_1}{\rho, \sigma \vdash \&E.x \Rightarrow r(x), \sigma_1}$$

$$\frac{\rho, \sigma \vdash E \Rightarrow l, \sigma_1}{\rho, \sigma \vdash *E \Rightarrow \sigma_1(l), \sigma_1}$$

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

Note that the meaning of $*E$ varies depending on its location.

- When it is used as l-value, $*E$ denotes the location that E refers to.
- When it is used as r-value, $*E$ denotes the value stored in the location.

Need for Memory Management

- New memory is allocated in let, call, and record expressions:

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

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

$$\frac{\rho, \sigma \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2 \qquad l_1, l_2 \notin Dom(\sigma_2)}{\rho, \sigma \vdash \{\ x := E_1, y := E_2\ \} \Rightarrow \{x \mapsto l_1, y \mapsto l_2\}, [l_1 \mapsto v_1, l_2 \mapsto v_2]\sigma_2}$$

- Allocated memory is never deallocated during program execution, eventually leading to memory exhaustion: e.g.,

  `let forever (x) = (forever x) in (forever 0)`

- We need to recycle memory that will no longer be used in the future.
- How can we know that memory will not be used in the future? Can we automate memory recycling?

Automatic Memory Management is Undecidable

- A bad news: exactly identifying memory locations that will be used in the future is impossible.
- Otherwise, we can solve the Halting problem, which is unsolvable.
  - We cannot write a program $H(p)$ that returns true iff program p terminates.
  - function f () = if H(f) then (while true skip) else skip
  - Does f() terminate?
    - ⋆ If f() terminates, it should not terminate.
    - ⋆ If f() is non-terminating, it should terminate (Contradiction!).

Automatic Memory Management is Undecidable

- Suppose we have an algorithm G that can exactly find the memory locations that will be used in the rest program execution.
- Then, we can construct $H(p)$ as follows:
  1. H takes p and execute the following program:

     ```
     let x = malloc() in p; x
     ```

     where `x` is a variable not used in p.
  2. Invoke the procedure G right before evaluating p, and find the location set S that will be used in the future.
     - ★ When S contains the location stored in `x`, p terminates.
     - ★ Otherwise, p does not terminate.

Approaches to Memory Management

Two approaches that trade-off control and safety:

1. Manual memory mangement: manually deallocate every unused memory locations.
    - E.g., C, C++
    - Pros: Fine control over the use of memory
    - Cons: Burden of writing correct code is imposed on programmers
2. Runtime garbage collection: *approximately* find memory locations that will not be used in the future and recycle them.
    - E.g., Java, OCaml
    - Pros: Memory safety
    - Cons: Fine control is impossible / Runtime overhead

cf) Some recent programming languages like Rust[1] achieve both safety and control by using static type system.

---

[1] `https://www.rust-lang.org`

Manual Memory Management

Extend the language with the deallocation expression:

$$\begin{array}{rcl} E & \rightarrow & \vdots \\ & | & \texttt{free}(E) \end{array}$$

Semantics rule:

$$\frac{\rho, \sigma \vdash E \Rightarrow l, \sigma_1}{\rho, \sigma \vdash \texttt{free}(E) \Rightarrow \cdot, \sigma_1|_{Dom(\sigma_1) \setminus \{l\}}} \; l \in Dom(\sigma_1)$$

where

$$\sigma|_X(l) = \begin{cases} \sigma(l) & \text{if } l \in X \\ \textbf{undef} & \text{if } l \notin X \end{cases}$$

Manual Memory Management

- Unfortunately, memory management is too difficult to do correctly, leading to the three types of errors in C:
  - Memory-leak: deallocate memory too late
  - Double-free: deallocate memory twice
  - Use-after-free: deallocate memory too early (dangling pointer)
- These errors are common in practice, becoming significant sources of security vulnerabilities.

Garbage Collection (GC)

Automatic garbage collection works in three steps:

1. Pause the program execution.
2. Collect memory locations reachable from the current environment.
3. Recycle unreachable memory locations.

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

Example

Environment and memory before GC:

$$\rho = \begin{bmatrix} x \mapsto l_1 \\ y \mapsto l_2 \end{bmatrix} \qquad \sigma = \begin{bmatrix} l_1 \mapsto 0 \\ l_2 \mapsto \{a \mapsto l_3, b \mapsto l_1\} \\ l_3 \mapsto l_4 \\ l_4 \mapsto (x, E, [z \mapsto l_5]) \\ l_5 \mapsto 0 \\ l_6 \mapsto l_7 \\ l_7 \mapsto l_6 \end{bmatrix}$$

Memory after GC:

$$\textsf{GC}(\rho, \sigma) = \begin{bmatrix} l_1 \mapsto 0 \\ l_2 \mapsto \{a \mapsto l_3, b \mapsto l_4\} \\ l_3 \mapsto l_4 \\ l_4 \mapsto (x, E, [z \mapsto l_5]) \\ l_5 \mapsto 0 \end{bmatrix}$$

Garbage Collection (GC): Formal Definition

- Let **reach**(ρ, σ) be the set of locations in σ that are reachable from the entries in ρ. It is the smallest set that satisfies the rules:

$$\frac{}{\rho(x) \in \textbf{reach}(\rho, \sigma)}\ x \in Dom(\rho) \qquad \frac{l \in \textbf{reach}(\rho, \sigma) \qquad \sigma(l) = l'}{l' \in \textbf{reach}(\rho, \sigma)}$$

$$\frac{l \in \textbf{reach}(\rho, \sigma) \qquad \sigma(l) = \{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\}}{\{l_1, \ldots, l_n\} \subseteq \textbf{reach}(\rho, \sigma)}$$

$$\frac{l \in \textbf{reach}(\rho, \sigma) \qquad \sigma(l) = (x, E, \rho')}{\textbf{reach}(\rho', \sigma) \subseteq \textbf{reach}(\rho, \sigma)}$$

- Let **GC** be the garbage-collecting procedure:

$$\textbf{GC}(\rho, \sigma) = \sigma|_{\textbf{reach}(\rho, \sigma)}$$

- Before evaluating an expression, perform **GC**:

$$\rho, \textbf{GC}(\rho, \sigma) \vdash E \Rightarrow v, \sigma'$$

Safe but Incomplete

GC performs memory management in an approximate but safe way.

Theorem (Safety of GC)

In the inference of $(\rho, \sigma \vdash E \Rightarrow v, \sigma')$*, the set of used (read or written) locations in* σ *is included in* $\mathbf{reach}(\rho, \sigma)$.

Proof.

By induction on E. ☐

However, some locations that will not be used may be reachable.

Summary

The final programming language:

- expressions, procedures, recursion,
- states with explicit/implicit references
- parameter-passing variations
- records, pointers, and automatic garbage collection