COSE212: Programming Languages

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

Hakjoo Oh
2024 Fall

Review: Our Language So Far

Syntax:

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

Values:

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

Review: Semantics Rules

(Some rules omitted)

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

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

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

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad [x \mapsto l]\rho, [l \mapsto v_1]\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_2}\ l \not\in \mathsf{Dom}(\sigma_1)$$

$$\frac{\begin{array}{c}\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \\ [x \mapsto l]\rho', [l \mapsto v]\sigma_2 \vdash E \Rightarrow v', \sigma_3\end{array}}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \not\in \mathsf{Dom}(\sigma_2)$$

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

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

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.

```
let arr[3] = { 1, 2, 3 }
in  arr[0] + arr[1] + arr[2]
```

Language Extension

Syntax:

$$
\begin{array}{rcl}
E & \to & \vdots \\
& | & \{\} \\
& | & \{\ x := E_1, y := E_2\ \} \\
& | & E.x \\
& | & E_1.x := E_2
\end{array}
$$

Values:

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

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:

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

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

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

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

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:

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

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:

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

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

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

- 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.

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:

$$\begin{array}{rcl} E & \to & \vdots \\ & \mid & \texttt{free}(E) \end{array}$$

Semantics rule:

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

where

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

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.

Repo.	#commits	ML	DF	UAF	Total	*-overflow
linux	721,119	3,740	821	1,986	**6,363**	5,092
php	105,613	1,129	148	197	**1,449**	649
git	49,475	350	19	95	**442**	258
openssl	21,009	220	36	12	**264**	61

cf) Memory Errors in Industrial Practice

Programmers spend significant amount of time in fixing memory errors:

Can we automate the process?

cf) Automatic Memory-Error Fixing

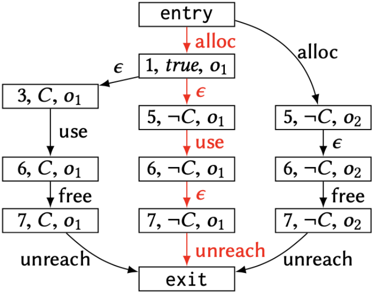
```
int append_data (Node *node, int *ndata) {
    if (!(Node *n = malloc(sizeof(Node)))
        return -1; // failed to be appended
    n->data = ndata;
    n->next = node->next; node->next = n;
    return 0; // successfully appended
}

Node *lx = ... // a linked list
Node *ly = ... // a linked list
for (Node *node = lx; node != NULL; node = node->next) {
    int *dptr = malloc(sizeof(int));
    if (!dptr) return;
    *dptr = *(node->data);
(-) append_data(ly, dptr); // potential memory-leak
(+) if ((append_data(ly, dptr)) == -1) free(dptr);
}
```

cf) Automatic Memory-Error Fixing

```
struct node *cleanup; // list of objects to be deallocated
struct node *first = NULL;
for (...) {
    struct node *new = xmalloc(sizeof(*new));
    make_cleanup(new); // add new to the cleanup list
    new->name = ...;
    ...
    if (...) {
        first = new;
(+)     tmp = first->name;
        continue;
    }
  /* potential use-after-free: `first->name` */
(-)  if (first == NULL || new->name != first->name)
(+)  if (first == NULL || new->name != tmp)
        continue;
  do_cleanups(); // deallocate all objects in cleanup
}
```

cf) Automatic Memory-Error Fixing

```
p = malloc(1); //o1
if (C)
  q = p;
else
  q = malloc(1); //o2
*p = 1;
free(q);
```



cf) Automatic Memory-Error Fixing

		Infer		SAVER								FootPatch [55]						
Program	**kLoC**	**#T**	**#F**	**Pre(s)**	**Fix(s)**	G_T	✓$_T$	△$_T$	✗$_T$	G_F	✗$_F$	**Fix(s)**	G_T	✓$_T$	△$_T$	✗$_T$	G_F	✗$_F$
rappel (ad8efd7)	2.1	1	0	0.5	0.3	1	1	0	0	0	0	5.3	1	1	0	0	0	0
flex (d3de49f)	22.3	3	4	5.8	1.7	0	0	0	0	0	0	26.2	0	0	0	0	1	1
WavPack (22977b2)	31.2	1	2	9.6	24.3	0	0	0	0	0	0	37.9	0	0	0	0	2	2
Swoole (a4256e4)	44.5	15	3	32.6	4.0	11	11	0	0	0	0	207.9	9	7	0	2	1	1
p11-kit (ead7a4a)	62.9	33	9	203.3	203.5	24	24	0	0	0	0	227.4	6	5	0	1	2	2
lxc (72cc48f)	63.0	3	5	56.0	4.3	3	3	0	0	0	0	134.6	0	0	0	0	1	1
x264 (d4099dd)	73.2	10	0	56.1	7.3	10	10	0	0	0	0	229.4	2	2	0	0	0	0
recutils-1.8	92.0	10	11	39.6	39.6	8	8	0	0	0	0	349.9	3	2	1	0	0	0
inetutils-1.9.4	116.9	4	5	24.2	2.7	4	4	0	0	0	0	107.9	0	0	0	0	0	0
snort-2.9.13	320.8	15	28	1527.8	112.6	11	10	1	0	0	0	1039.6	3	0	0	3	19	18
Total	828.9	95	67	1804.7	343.5	72	71	1	0	0	0	2366.1	24	15	1	8	26	25

- MemFix: Static Analysis-Based Repair of Memory Deallocation Errors for C.
  Junhee Lee, Seongjoon Hong, and Hakjoo Oh.
  FSE 2018: ACM Symposium on the Foundations of Software Engineering.
  `http://prl.korea.ac.kr/~pronto/home/papers/fse18.pdf`
- SAVER: Scalable, Precise, and Safe Memory-Error Repair.
  Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh.
  ICSE 2020: 42nd International Conference on Software Engineering.
  `http://prl.korea.ac.kr/~pronto/home/papers/icse20.pdf`

Automatic Memory Management (Garbage Collection)

1. When no more memory is available, pause the program execution.
2. Collect all the memory locations that will not be used anymore.
3. Remove those memory locations in the current memory.

E.g.,

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

The environment and memory right before evaluating a+1:

$$\rho = \{\texttt{f} \mapsto l_1, \texttt{a} \mapsto l_3\}, \sigma = \{l_1 \mapsto (\texttt{x}, \texttt{x+1}, \emptyset), l_2 \mapsto 0, l_3 \mapsto 1\}$$

After garbage collection:

$$\rho = \{\texttt{f} \mapsto l_1, \texttt{a} \mapsto l_3\}, \sigma = \{l_3 \mapsto 1\}$$

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.
  - We cannot write a program $H(p)$ that returns true iff program p terminates.
- 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.

Garbage Collection (GC) in Practice

1. When no more memory is available, pause the program execution.
2. Collect memory locations that are not reachable from the current environment.
3. Remove those memory locations in the current memory.

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

The environment and memory right before evaluating a+1:

$$\rho = \{\mathsf{f} \mapsto l_1, \mathsf{a} \mapsto l_3\}, \sigma = \{l_1 \mapsto (\mathtt{x}, \mathtt{x+1}, \emptyset), l_2 \mapsto 0, l_3 \mapsto 1\}$$

After garbage collection:

$$\rho = \{\mathsf{f} \mapsto l_1, \mathsf{a} \mapsto l_3\}, \sigma = \{l_1 \mapsto (\mathtt{x}, \mathtt{x+1}, \emptyset), l_3 \mapsto 1\}$$

More 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:

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

Garbage Collection (GC): Formal Definition

- Let $\textbf{reach}(\rho, \sigma)$ 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. $\square$

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