

# COSE212: Programming Languages

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

Hakjoo Oh  
2025 Fall

# Review: Our Language So Far

Syntax:

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

Values:

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

# Review: Semantics Rules

(Some rules omitted)

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

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

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

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

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

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

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

# 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:

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

Values:

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

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

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

# Language Extension

Semantics:

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

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

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

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

# 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:

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

Values:

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

# Language Extension

Semantics:

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

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

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

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

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:

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

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

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

- 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:

- ❶ Manual memory management: 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
- ❷ 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<sup>1</sup> achieve both safety and control by using static type system.

---

<sup>1</sup><https://www.rust-lang.org>

# Manual Memory Management

Extend the language with the deallocation expression:

$$\begin{array}{c} E \rightarrow \vdots \\ | \text{ free}(E) \end{array}$$

Semantics rule:

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

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	<b>6,363</b>	5,092
php	105,613	1,129	148	197	<b>1,449</b>	649
git	49,475	350	19	95	<b>442</b>	258
openssl	21,009	220	36	12	<b>264</b>	61

# cf) Memory Errors in Industrial Practice

Programmers spend significant amount of time in fixing memory errors:

Subject	Status	Owner	Updated	Size	
Fi	Subject	Status	Owner	Updated	Size
Fi	Subject	Status	Owner	Updated	Size
Fi	Handled the memory leak	Merged	Mayank Haarit	May 17	
m	Fix double free	Merged	jusung son	May 16	
Fi	Fix memory leak	Merged	Vyacheslav Cherkashin	May 16	
Fi	Release version 1.0.4	Merged	Hyunho Kang	May 16	
[T	Fix heap-use-after-free error detected by Adress Sanitizer.	Merged	Lukasz Stelmach	May 16	
ec	Fix memory leak	Merged	MyungKi Lee	May 15	
Fi	[MM][TTS] Removed EWK_BRINGUP flag for TTS	Merged	joseph lolak	May 14	
Fi	Release version 1.0.2	Merged	Hyunho Kang	May 14	
R				May 11	
A				May 10	
Fi				May 10	
Fi				May 10	
fx				May 10	
[C				May 10	
M				May 9	
M	Fix covurity issues	Merged	seolheul kim	May 9	
Fi	evas_out : fix a memory leak.	Abandoned	junsu choi	May 9	
M	Fix Bundle memory leak	Merged	Hyunho Kang	May 9	
ta	genlist: prevent memory leak in item class update	Merged	SangHyeon Lee	May 8	
C	genlist: prevent memory leak in item class update	Merged	SangHyeon Lee	May 8	
R	[M63 Migration] Fix memory leak for evas and ecore event register	Merged	chen shurong	May 7	
Fi	Fix heap-use-after-free error detected by Adress Sanitizer.	Merged	Elmurod Talipov	May 4	
R	cbhm_helper: fixed the memory leak for covurity	Merged	Taehyub Kim	May 4	
	Fix memory leak	Merged	MyungKi Lee	May 4	
	Fix memory leak in app control	Merged	Chang Joo Lee	May 3	
	Fix memory leak issue	Merged	Jihoon Kim	May 2	

**1,700 error fixing  
commits in 3 years**

Can we automate the process?

## cf) Automatic Memory-Error Fixing

```
1  int append_data (Node *node, int *ndata) {
2      if (!(Node *n = malloc(sizeof(Node)))
3          return -1; // failed to be appended
4      n->data = ndata;
5      n->next = node->next; node->next = n;
6      return 0; // successfully appended
7  }
8
9  Node *lx = ... // a linked list
10 Node *ly = ... // a linked list
11 for (Node *node = lx; node != NULL; node = node->next) {
12     int *dptr = malloc(sizeof(int));
13     if (!dptr) return;
14     *dptr = *(node->data);
15     (-) append_data(ly, dptr); // potential memory-leak
16     (+) if ((append_data(ly, dptr)) == -1) free(dptr);
17 }
```

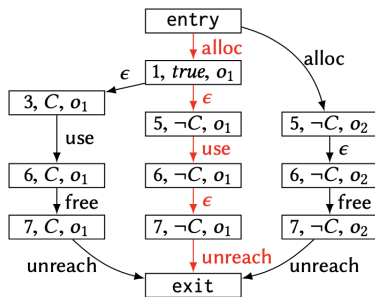


## cf) Automatic Memory-Error Fixing

```
1  struct node *cleanup; // list of objects to be deallocated
2  struct node *first = NULL;
3  for (...) {
4      struct node *new = xmalloc(sizeof(*new));
5      make_cleanup(new); // add new to the cleanup list
6      new->name = ...;
7      ...
8      if (...) {
9          first = new;
10         (+) tmp = first->name;
11         continue;
12     }
13     /* potential use-after-free: `first->name` */
14     (-) if (first == NULL || new->name != first->name)
15     (+) if (first == NULL || new->name != tmp)
16         continue;
17     do_cleanups(); // deallocate all objects in cleanup
18 }
```

## cf) Automatic Memory-Error Fixing

```
1  p = malloc(1); //o1
2  if (C)
3    q = p;
4  else
5    q = malloc(1); //o2
6  *p = 1;
7  free(q);
```



## cf) Automatic Memory-Error Fixing

Program	kLoC	INFER		SAVER								FootPatch [55]							
		#T	#F	Pre(s)	Fix(s)	G <sub>T</sub>	✓ <sub>T</sub>	Δ <sub>T</sub>	X <sub>T</sub>	G <sub>F</sub>	X <sub>F</sub>	Fix(s)	G <sub>T</sub>	✓ <sub>T</sub>	Δ <sub>T</sub>	X <sub>T</sub>	G <sub>F</sub>	X <sub>F</sub>	
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://pr1.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://pr1.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`:

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

After garbage collection:

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

# 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`:

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

After garbage collection:

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

## More Example

Environment and memory before GC:

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

Memory after GC:

$$\mathbf{GC}(\rho, \sigma) = \left[ \begin{array}{l} l_1 \mapsto 0 \\ l_2 \mapsto \{a \mapsto l_3, b \mapsto l_4\} \\ l_3 \mapsto l_4 \\ l_4 \mapsto (x, E, [z \mapsto l_5]) \\ l_5 \mapsto 0 \end{array} \right]$$

# Garbage Collection (GC): Formal Definition

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

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

$$\frac{l \in \text{reach}(\rho, \sigma) \quad \sigma(l) = \{x_1 \mapsto l_1, \dots, x_n \mapsto l_n\}}{\{l_1, \dots, l_n\} \subseteq \text{reach}(\rho, \sigma)}$$

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

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

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

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

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



## 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  $\sigma$  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