CS450

Structure of Higher Level Languages

Lecture 16: Garbage collection

Tiago Cogumbreiro

Today we will...

- Introduce memory management
- Reference counting garbage collection
- Mark-and-sweep garbage collection

> Inspired by Professor Michelle Mills' lecture on garbage collection, Colorado State University.

Motivation

```
(eval-term*?
  '[(E0)] 'E0
  ;; Program
  '[(define (f x) (lambda (y) x))
    (f 2)
    (f 10)
    (f 5)]
  ;; Output Value
  '(closure E3 (lambda (y) x))
  ;; Output Memory
  ???)
```

Motivation

```
(eval-term*?
  '[(E0)] 'E0
  ;; Program
  '[(define (f x) (lambda (y) x))
    (f 2)
    (f 10)
    (f 5)]
  ;; Output Value
  '(closure E3 (lambda (y) x))
  ;; Output Memory
  '[(E0 . [(f . (closure E0 (lambda (x) (lambda (y) x))))])
    (E1 . [E0 (x . 2)])
    (E2 . [E0 (x . 10)])
    (E3 . [E0 (x . 5)])])
```

Motivation

```
(eval-term*?
  ;; Input memory
  '[(E0)]
  ;; Env
  'E0
  ;; Program
  '[(define (f x) (lambda (y) x))
    (f 2)
    (f 10)
    (f 5)
    (f 99)
    (f 98)]
  ;; Output Value
  '(closure ?? (lambda (y) x))
  ;; Output Memory
  '[(E0 . [(f . (closure E0 (lambda (x) (lambda (y) x))))])
    (E1 . [E0 (x . 2)])
    (E2 . [E0 (x . 10)])
    (E3 . [E0 (x . 5)])
    ???
```

Motivation

```
(eval-term*?
  ;; Input memory
  '[(E0)]
  ;; Env
  'E0
  ;; Program
  '[(define (f x) (lambda (y) x))
    (f 2)
    (f 10)
    (f 5)
    (f 99)
    (f 98)]
  ;; Output Value
  '(closure E5 (lambda (y) x))
  ;; Output Memory
  '[(E0 . [(f . (closure E0 (lambda (x) (lambda (y) x))))])
    (E1 . [E0 (x . 2)])
    (E2 . [E0 (x . 10)])
    (E3 . [E0 (x . 5)])
    (E4 . [E0 (x . 99)])
    (E5 . [E0 (x . 98)])])
```

Motivation

- Whenever we call function `f` it creates a new function, which allocates a new frame.

Do we need these frames?

Motivation

- Whenever we call function `f` it creates a new function, which allocates a new frame.

Do we need these frames?

Can we reclaim them?

Motivation

> Whenever we call function `f` it creates a new function, which allocates a new frame.

Do we need these frames?

Can we reclaim them?

How do we know if we need these frames?

Memory management

Objective: Discard any memory that will not be used in the future

- **Manual:** The programmer explicitly controls when data is reclaimed.
- **Automatic (Garbage Collection):** An algorithm controls when data is reclaimed

Memory management

- **Soundness:** we must only reclaim unneeded data.
  Problem: Reclaiming memory too soon leads to dangling references, which gives rise to crashes or security breaches.
- **Completeness:** we must *eventually* reclaim unneeded data.
  Problem: Forgetting to reclaim memory leads to resource depletion, which gives rise to memory swapping, slowness, or denial of service.

Quiz

If our garbage collector works by never reclaiming memory. Is it sound? Is it complete?

Memory management challenges

> The choice between *automatic* and *manual* memory management is a balance between many design constrains:

1. what is the impact of a soundness failure? low impact, then manual
2. what is the impact of a completeness failure? low impact, then automatic
3. how easy to program? easy → automatic
4. how easy to profile? easy → manual
5. how easy to implement? easy → manual
6. human intervention? ok → manual

Manual memory management

- **Pro:** Can be very efficient
- **Pro:** Lets the programmer control when memory should be reclaimed (eg, real time problems, games)
- **Pro:** Implementing manual memory management is generally easier than automatic memory management
- **Con:** More code to maintain
- **Con:** Ensuring correctness can be difficult and hazardous

Did you know?

> Rust is an example of a new language that introduces manual memory management that is *assisted* by the compiler, which helps in reducing memory-management code and also enforces correctness. The implementation of this technique is considerably more involved tha a traditional unsafe memory management.

Automatic memory management

- **Pro:** Less code to maintain
- **Pro:** Memory-management correctness is **guaranteed** (aka memory *safety*)
- **Con:** More difficult to control when memory should be reclaimed
- **Con:** Implementing automatic memory management is generally more complicate than manual memory management

Did you know?

1. Researchers are experimenting with extending C# with API's that allow for safe **manual** memory management (Parkinson *et al*, 2017), allowing the programmer to get the best of both worlds.
2. One of the biggest difficulties of handling automatic memory management is **parallelism**; *concurrent* garbage collection is one of the most intricate pieces of technology in programming language development.

Garbage collection

aka Automatic memory management

Garbage collection must be conservative

> How do we know if a piece of memory will be used in the future? The garbage collector **cannot** predict the future.

Therefore, garbage collection can only discard memory it can *prove* that cannot be used. Proving that some memory is not needed is done by finding its uses (references).

Did you know?

> The garbage collector must be able to iterate over all references in memory. In C any number can be considered a pointer, which makes C garbage collection unsound by definition. The garbage collector must introduce assumptions. For instance, assume that numbers will never used as references.

Garbage collection

Overview

- **Reference counting:** incrementally maintain the count of memory usage; when count is zero reclaim memory (eg, Python, C++, Objective-C, Rust)
- **Reachability:** do a full-memory sweep by following references; unreachable memory is discarded (eg, Racket, Java, JavaScript)

Garbage collection:

Reference counting

Reference counting

1. Map the reference count for each handle in the heap
2. When allocating a handle, that reference is set to 0; increment each reference in the initial value
3. When updating a handle, decrement each reference in the old value and increment each reference of the new value
4. If a reference count reaches zero, then that reference is garbage; collect it!

Example

```
Node x = null; Node y = null;
x = new Node(3, null); // o1: 1
y = x;                 // o1: 2
x = null;              // o1: 1
y = x;                 // o1: 0
```

```
(define h1-eff (heap-alloc h0 (list 3 null))); o1:0
(define h1 (eff-state h1-eff))
(define o1 (eff-result h1-eff))
(define h2 (heap-put h1 x o1))                 ; o1:1
(define h3 (heap-put h2 y (heap-get x)))       ; o1:2
(define h4 (heap-put h3 x null))               ; o1:1
(define h5 (heap-put h4 y (heap-get x)))       ; o1:0
```

Reference counting example (1)

> By inspecting the frames allocated in the heap we can compute the reference count of each handle. **We can garbage collect any frame whose reference count is zero.** *A handle is referenced by a frame if, and only if, the handle is contained in a frame.* If `E` is a parent handle in a frame, then it is contained in that frame. If `E` is the environment of a closure stored in a local binding of a frame, we say that `E` is *contained* in that frame.

Which handles are garbage?

Heap

```
'[(E0 . [(f . (closure E0 (lambda (x) (lambda (y) x))))])
  (E1 . [E0 (x . 2)])
  (E2 . [E0 (x . 10)])
  (E3 . [E0 (x . 5)])]
```

Reference counting example (1)

> By inspecting the frames allocated in the heap we can compute the reference count of each handle. **We can garbage collect any frame whose reference count is zero.** *A handle is referenced by a frame if, and only if, the handle is contained in a frame.* If `E` is a parent handle in a frame, then it is contained in that frame. If `E` is the environment of a closure stored in a local binding of a frame, we say that `E` is *contained* in that frame.

Which handles are garbage?

Heap

```
'[(E0 . [(f . (closure E0 (lambda (x) (lambda (y) x))))])
  (E1 . [E0 (x . 2)])
  (E2 . [E0 (x . 10)])
  (E3 . [E0 (x . 5)])]
```

Reference count

```
E0: 4
E1: 0
E2: 0
E3: 0
```

> We can safely collect `E1`, `E2`, and `E3`.

Reference count example (2)

```
(eval-term*?
  '[(E0)]
  'E0
  '[
    (define (f x)
      (define (z a) x)
      (lambda (y) (z y)))
    (f 0)
    (f 10)]
  '(closure E2 (lambda (y) (z y)))
  '[(E0 . [(f . (closure E0 (lambda (x) (define (z a) x) (lambda (y) (z y)))))])
    (E1 . [E0 (x .  0) (z . (closure E1 (lambda (a) x)))])
    (E2 . [E0 (x . 10) (z . (closure E2 (lambda (a) x)))])])
```

Reference count example (2)

```
(eval-term*?
  '[(E0)]
  'E0
  '[
    (define (f x)
      (define (z a) x)
      (lambda (y) (z y)))
    (f 0)
    (f 10)]
  '(closure E2 (lambda (y) (z y)))
  '[(E0 . [(f . (closure E0 (lambda (x) (define (z a) x) (lambda (y) (z y)))))])
    (E1 . [E0 (x .  0) (z . (closure E1 (lambda (a) x)))])
    (E2 . [E0 (x . 10) (z . (closure E2 (lambda (a) x)))])])
```

Reference counting

```
E0: 3   E1: 1   E2: 1
```

Reference counting

- **Allocation and update time overhead:** must traverse the values being stored for reference counting update
- **Reclamation is local and incremental** heap becomes fragmented
- **Cannot handle cyclic data structures**
- **Space overhead:** must maintain the reference counting for each handle
- **Reclamation is predictable:** we can infer exactly when memory reclamation is happening, which is useful for time-sensitive algorithms (such as real time algorithms and games)

Reference counting is of limited use to us

because frames can have cycles!

Garbage collection:

Reachability

Reachability (aka tracing)

Mark-and-sweep algorithm

1. **Mark.** Your starting point are your "globals". For each reference, traverse its usages, while collecting all visited references; avoid cycles.
2. **Sweep.** Copy all visited references to a new heap and discard old heap.

Mark-and-sweep example (1)

> The heap can represent a graph where we draw a *reference-use* edge from each key to each handle in the values. If we start Mark-and-sweep from `E0`, then we can collect `E1`, `E2`, and `E3`.

Heap

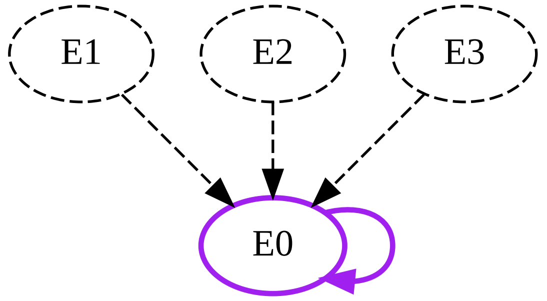
```
'[(E0 . (f closure E0 (lambda (x) (lambda (y) x))))
  (E1 . [E0 (x . 2)])
  (E2 . [E0 (x . 10)])
  (E3 . [E0 (x . 5)])]
```

Mark-and-sweep example (1)

> The heap can represent a graph where we draw a *reference-use* edge from each key to each handle in the values. If we start Mark-and-sweep from `E0`, then we can collect `E1`, `E2`, and `E3`.

Heap

```
'[(E0 . (f closure E0 (lambda (x) (lambda (y) x))))
  (E1 . [E0 (x . 2)])
  (E2 . [E0 (x . 10)])
  (E3 . [E0 (x . 5)])]
```



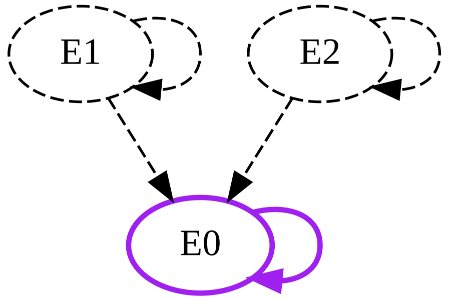
Mark-and-sweep example (2)

> If we start Mark-and-sweep from `E0`, then we can safely garbage collect `E1` and `E2`, as it cannot be reached from any global, that is we cannot reach `E1` nor `E2` from `E0`.

Heap

```
'[(E0 . [(f . (closure E0 (lambda (x) (define (z a) x) (lambda (y) (z y)))))])
  (E1 . [E0 (x .  0) (z . (closure E1 (lambda (a) x))) ])
  (E2 . [E0 (x . 10) (z . (closure E2 (lambda (a) x))) ])
]
```

Reference use



Mark-and-sweep summary

- **No allocation and no update time overhead**
- **Reclamation is global must stop the world** must copy all references from one heap to another; the whole heap must be traversed; no fragmentation
- **No space overhead per-reference**
- **Space overhead to create new heap**
- **Reclamation is not predictable:** garbage collection is a global operation so no amortization possible

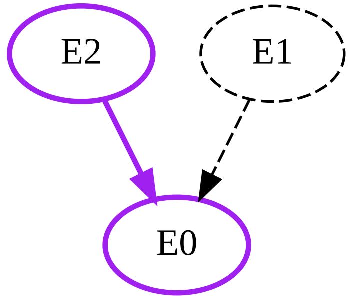
Mark-and-sweep example (3)

If start Mark-and-sweep from `E2`, then we can safely garbage collect `E1`.

Heap

```
'[(E0 . [(x . 10)])
  (E1 . [E0 (x . 20) ])
  (E2 . [E0 (x . 30) ])
]
```

Reference use



Handle creation problem

Before garbage collection

```
'[(E0 . [(x . 10)])
  (E1 . [E0 (x . 20) ])
  (E2 . [E0 (x . 30) ])
]
```

After garbage collection

```
'[(E0 . [(x . 10)])
  (E2 . [E0 (x . 30) ])
```

> What happens if we allocate some data in the heap above?

```
(define (heap-alloc h v)
  (define new-id (handle (hash-count (heap-data h))))
  (define new-heap (heap (hash-set (heap-data h) new-id v)))
  (eff new-heap new-id))
```

Handle creation problem

- What happens if we allocate frame `[E0 (x . 9)]` (some frame without bidings)?

Before adding a frame

```
'[(E0 . [(x . 10)])
  (E2 . [E0 (x . 30) ])
```

Handle creation problem

What happens if we allocate frame `[E0 (x . 9)]` (some frame without bidings)?

Before adding a frame

```
'[(E0 . [(x . 10)])
  (E2 . [E0 (x . 30) ])
```

After adding a frame

```
'[(E0 . [(x . 10)])
  (E2 . [E0 (x . 9) ])
```

Using `hash-count` is not enough!

We must ensure that handle creation plays well with GC

Moving versus non-moving garbage collection

- **Non-moving.** If garbage collection simply claims unreachable data, then garbage collection faces the problem of fragmentation (which we noticed in the previous example)
- **Moving.** Alternatively, garbage collection may choose to "move" the references around by placing data in different locations, which handles the problem of fragmentation, but now it must be able to translate the references in the data

Specifying Mark-and-sweep

Specifying Mark

Given an initial handle, collect the set of reachable handles.
We say that a handle x directly connects to a handle y if handle y is *contained* in the frame addressed by x. We say that a handle is contained in frame in either situation:

1. If the frame has a parent, then that handle is contained in the frame.
2. If a closure is a local value of the frame, and that closure captures handle x, then x is contained in the frame.

Specifying Mark

Homework 6

Function `frame-refs` must return the set of contained handles.

Example 1

```
(check-equal?
 (frame-refs
   (parse-frame
    '(E2
      (x . 0)
      (y . (closure E0 (lambda (x) x)))
      (z . (closure E1 (lambda (x) x))))))
 (set (handle 0) (handle 1) (handle 2)))
```

Example 2

```
(check-equal?
 (frame-refs
  (parse-frame
    ; no parent!
    '((x . 0)
      (y . (closure E0 (lambda (x) x)))
      (z . (closure E1 (lambda (x) x))))))
 (set (handle 0) (handle 1)))
```

Sets in Racket

```
(require racket/set) ; ← do not forget to load the sets library
```

Constructors

- `(set v1 v2 v3 ...)` creates a (possibly empty) set of values, corresponds to $\{v_1, v_2, v_3, \ldots\}$
- `(set-union s1 s2)` returns a new set that holds the union of sets `s1` and `s2`, corresponds to $s_1 \cup s_2$
- `(set-add s x)` returns a new set that holds the elements of `s` and also element `x`, corresponds to $s \cup \{x\}$
- `(set-subtract s1 s2)` returns a new set that consists of all elements that are in `s1` but are not in `s2`, corresponds to $\{x \mid x \in s_1 \land x \notin s_2\}$

Sets in Racket

Selectors

- `(set-member? s x)` returns if `x` is a member of set `s`, corresponds to $x \in s$
- `(set->list s)` converts set `s` into a list

Homework 6

> **How do you iterate over the values of a frame?** You might want to look at function `frame-fold` or function `frame-values`.

Specifying Mark-and-sweep

Specifying Sweep

1. **What is the input?**

Specifying Mark-and-sweep

Specifying Sweep

1. **What is the input?** `heap`**?** and set of handles
2. **Which functional pattern?**

Specifying Mark-and-sweep

Specifying Sweep

1. **What is the input?** `heap?` and set of handles
2. **Which functional pattern?** A `filter`. See `heap-filter`.
3. **What are we keeping?**

Specifying Mark-and-sweep

Specifying Sweep

1. **What is the input?** `heap?` and set of handles
2. **Which functional pattern?** A `filter`. See `heap-filter`.
3. **What are we keeping?** All handles in the input set