CS450

Structure of Higher Level Languages

Lecture 27: Mark and sweep; sets; refactoring evaluation

Tiago Cogumbreiro
Garbage Collection
and our implementation of Heap
Handle creation problem

Before garbage collection

\[
\begin{align*}
\text{E0} & . [(x . 10)] \\
\text{E1} & . [\text{E0} (x . 20)] \\
\text{E2} & . [\text{E0} (x . 30)]
\end{align*}
\]

After garbage collection

\[
\begin{align*}
\text{E0} & . [(x . 10)] \\
\text{E2} & . [\text{E0} (x . 30)]
\end{align*}
\]

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

\[
\begin{align*}
\text{(define } \text{heap-alloc } h \text{ v)} \\
\text{(define new-id (handle (hash-count (heap-data h)))} \\
\text{(define new-heap (heap (hash-set (heap-data h) new-id v)))} \\
\text{(eff new-heap new-id))}
\end{align*}
\]
Handle creation problem

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

Before adding a frame

\[
\begin{align*}
\left( (x \ . \ 10) \right) \\
\left( E2 \ . \ [E0 \ (x \ . \ 30)] \right)
\end{align*}
\]
Handle creation problem

What happens if we allocate frame \([E_0 (x \cdot 9)]\) (some frame without bindings)?

Before adding a frame

\[
\begin{align*}
&(E_0 \cdot [(x \cdot 10)]) \\
&(E_2 \cdot [E_0 (x \cdot 30)])
\end{align*}
\]

After adding a frame

\[
\begin{align*}
&(E_0 \cdot [(x \cdot 10)]) \\
&(E_2 \cdot [E_0 (x \cdot 9)])
\end{align*}
\]

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.
Homework 6
Homework 6

1. frame-refs given a frame return a set of handles contained in that frame
2. mem-mark given a function that returns the contained handles of an element, and an initial handle, returns the set of reachable handles (including the initial handle).
3. mem-sweep given a heap and a set of handles returns a new heap which only contains the handles in the given set.
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 (E2 (x . 0) (y . (closure E0 (lambda (x) x))) (z . (closure E1 (lambda (x) x))))))) (set (handle 0) (handle 1)))
Sets in Racket

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

Constructors

- \((\text{set } v_1 \; v_2 \; v_3 \; \ldots)\) creates a (possibly empty) set of values, corresponds to 
  \(\{v_1, v_2, v_3, \ldots\}\)
- \((\text{set-union } s_1 \; s_2)\) returns a new set that holds the union of sets \(s_1\) and \(s_2\), corresponds to \(s_1 \cup s_2\)
- \((\text{set-add } s \; x)\) returns a new set that holds the elements of \(s\) and also element \(x\), corresponds to \(s \cup \{x\}\)
- \((\text{set-subtract } s_1 \; s_2)\) returns a new set that consists of all elements that are in \(s_1\) but are not in \(s_2\), corresponds to \(\{x \mid x \in s_1 \land x \notin s_2\}\)
Sets in Racket

Selectors

- \((\text{set-member? } s \ x)\) returns if \(x\) is a member of set \(s\), corresponds to \(x \in s\)
- \((\text{set->list } s)\) 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
3. **What are we keeping?**
Specifying Mark-and-sweep

Specifying Sweep

1. **What is the input?** heap? and set of handles
3. **What are we keeping?** All handles in the input set
Today we will...

- Introduce a functional pattern monads
- Introduce state monads
Revisiting our reduction rules

\[ \Rightarrow_H v \Downarrow_E v \Rightarrow_H \]

\[ \Rightarrow_{H_1} e_f \Downarrow_E (E_f, \lambda x . t_b) \Rightarrow_{H_2} e_a \Downarrow_E v_a \Rightarrow_{H_3} E_b \leftarrow E_f + [x := v_a] \Rightarrow_{H_4} t_b \Downarrow_E v_b \Rightarrow_{H_5} \]

\[ \Rightarrow_{H_1} (e_f e_a) \Downarrow_E v_b \Rightarrow_{H_5} \]

Effectful computation can be divided into three categories:

- Side-effect free computation in **blue**
- Computation that directly produces side effect in **red**
- Computation that indirectly produces some side-effect in **black**

We are \( \Rightarrow \) chaining \( \Rightarrow \) effectful \( \Rightarrow \) computations \( \Rightarrow \), that is the variables declared on the left-hand side of \( \Rightarrow \) should be accessible in the right-hand side.
The memory needs to be passed along from one function call to the next. How can we refactor this code so that some function does that for us?
Refactoring evaluation of application

At each step we separate the result from the state. Our goal is to \textbf{abstract} the memory threading, that is to refactor away this mechanic unpacking of the side effect structure.
In today's class, we introduce an abstraction that allows us to achieve something similar to the pseudo-code below. We highlight in yellow effectful definitions and operations.

```plaintext
;; e1 ⇓E v1
(define* v1 (d:eval-exp* env e1))

;; E' ← E + [x := v1]
(define* env2 (environ-push* env y v1))

;; e2 ⇓E' v2
(define* v2 (d:eval-exp* env2 e2))
```
Roadmap: abstracting effectful computation

Combining:

- **Effectful operations**: `s:eval-exp` and `environ-push`, with
- **Effectful variable declaration**: `v1`, `env2`, and `v2`

```scheme
;; e1 ⇓E v1
(define* v1 (d:eval-exp env e1))
;; E' ← E + [x := v1]
(define* env2 (environ-push* env y v1))
;; e2 ⇓E' v2
(define* v2 (d:eval-exp* env2 e2))
```

\[ e_1 \Downarrow_E v_1 \quad \Rightarrow \quad E' \leftarrow E + [y := v] \quad \Rightarrow \quad e_2 \Downarrow_{E'} v_2 \]
A proxy example

Arithmetic on the heap
Example

Consider a heap of integers. We allocate two integers and then a third integer that holds the sum of the first two.

(define (prog1 h1)
  ;; allocate x with 1
  (define eff-x (heap-alloc h1 1))
  (define x (eff-result eff-x))
  (define h2 (eff-state eff-x))

  ;; allocate y with 2
  (define eff-y (heap-alloc h2 2))
  (define y (eff-result eff-y))
  (define h3 (eff-state eff-y))

  ;; allocate z with (+ x y)
  (heap-alloc h3 (+ (heap-get h3 y) (heap-get h3 x))))
  (define (run-state h op) (eff-state (op h)))
  (define H (heap (hash (handle 0) 1 (handle 1) 2 (handle 2) 3)))
  (check-equal? (run-state empty-heap prog1) H)

CS450 ☽ Mark and sweep; sets; refactoring evaluation ☽ Lecture 27 ☽ Tiago Cogumbreiro
Effectful operations

- An **effectful operation** takes a state and returns an effect `eff` that pairs some state with some result. An effectful operation is parameterized by the state type and by the result type.
- Below we define two effectful operations where the state is a heap.

**Add**

```
(define (num x)
    (lambda (h)
        (heap-alloc h x)))
```

**Alloc**

```
(define (add x y)
    (lambda (h)
        (heap-alloc h (+ (heap-get h y) (heap-get h x)))))
```

**Did you know?**

- The state (heap) is a parameter, so that we can combine effectful operations.
- Functions `num` and `add` each returns an effectful operation
Sequencing effectful operations

Example

```
(define (prog2 h1)
  ;; allocate x with 1
  (define eff-x ((num 1) h1))
  (define x (eff-result eff-x))
  (define h2 (eff-state eff-x))
  ;; allocate x with 2
  (define eff-y ((num 2) h2))
  (define y (eff-result eff-y))
  (define h3 (eff-state eff-y))
  ;; allocate y with (+ x y)
  ((add x y) h3))
```

The bind operator

```
(define (bind o1 o2)
  (lambda (h1)
    (define eff-r (o1 h1))
    (define r (eff-result eff-r))
    (define h2 (eff-state eff-r))
    ((o2 r) h2))
)
```

We highlight in yellow an example of redundant code. Function `bind` abstracts away the redundant code.
Abstracting with bind

Before

(define (prog2 h1)
  ;; allocate x with 1
  (define eff-x ((num 1) h1))
  (define x (eff-result eff-x))
  (define h2 (eff-state eff-x))
  ;; allocate x with 2
  (define eff-y ((num 2) h2))
  (define y (eff-result eff-y))
  (define h3 (eff-state eff-y))
  ;; allocate y with (+ x y)
  ((add x y) h3))

Using the bind operator we remove redundant code. You can think of bind as a variable declaration, akin to an effectful define.

After

(define prog3
  ;; allocate x with 1
  (bind (num 1)
    (lambda (x)
      ;; allocate x with 2
      (bind (num 2)
        (lambda (y)
          ;; allocate y with (+ x y)
          (add x y)))))))