CS450

Structure of Higher Level Languages

Lecture 27: From spec to code; mark and sweep; sets

Tiago Cogumbreiro

From spec to code

Implementing a big-step operational semantics

In Racket

Introducing μ-JavaScript

Syntax

$$p ::= x = e; \mid \texttt{while}\ (e)\{p\} \mid \texttt{console.log}(e); \mid p\ p$$

Example

```
x = 10;
while (x) {
    console.log(x);
    x = x - 1;
}
```

```
$ node foo.js
10
9
8
7
6
5
4
3
2
1
```

Introducing μ-JavaScript

Syntax

```
x = 10;
while (x) {
    console.log(x);
    x = x - 1;
}
```

```
(j:seq
  (j:assign (r:variable 'x) (r:number 10))
  (j:while (r:variable 'x)
      (j:seq
        (j:log (r:variable 'x))
        (j:assign (r:variable 'x)
            (r:apply
              (r:variable '-)
              (list
                (r:variable 'x)
                (r:number 1)))))))
```

Semantics of μ-JavaScript

Math

$$(m, p) \Downarrow m'$$

- an input map m
- an input program p
- an output map m'

Racket

```
(define/contract (j:eval vars prog)
  (→ hash? j:program? hash?)
```

- vars is m, implemented with hash
- prog the input program p
- The return value is going to be m'

Why do we return a heap m'?

μ-JavaScript programs mutate the heap with assignments, so the evaluation needs to return the updated heap.]

Rule E-assign

Math

$$\frac{e \Downarrow_m v}{(m, x = e) \Downarrow m[x \mapsto v]}$$

Racket

```
[(j:assign? prog)
  (define v (r:eval vars (j:assign-exp prog)))
  (hash-set vars (j:assign-var prog) v)]
```

Why?

- `(j:assign? prog)` because we are specifying that the input program must be an assignment: $(m, x = e) \Downarrow m[x \mapsto v]$
- `(define v (r:eval (j:assign-exp prog)))`
  Each evaluation above the fraction, eg $e \Downarrow v$, becomes a define
- since we are using hash-tables and $m[x \mapsto v]$ updates the map, thus `(hash-set m x v)`
- above the fraction we keep intermediate computations and constraints
- Notice that in the code we have `vars` but in the rule we have m

Rule E-log

Math

$$\frac{e \Downarrow_m v \qquad \log(v)}{(m, \texttt{console.log}(e);) \Downarrow m}$$

Racket

```
[(j:log? prog)
  (define v (r:eval vars (j:log-exp prog)))
  (displayln v)
  m]
```

Why?

- `(j:log? prog)` because we are specifying that the input program must be an assignment: $(m, \texttt{console.log}(e);) \Downarrow m$
- `(displayln v)`
  In the formalism we have an abstract function to print out the value, $\log(v)$

Rule E-seq

Math

$$\frac{(m_1, p_1) \Downarrow m_2 \qquad (m_2, p_2) \Downarrow m_3}{(m_1, p_1\ p_2) \Downarrow m_3}$$

Racket

```
[(j:seq? prog)
  (define p1 (j:seq-left prog))
  (define p2 (j:seq-right prog))
  (define m2 (j:eval vars p1))
  (define m3 (j:eval m2 p2))
  m3]
```

Why?

- `(define p1 (j:seq-left prog))` to improve readability we define p_1 in Racket. Variable p_1 is defined in:
  $(m_1, p_1\ p_2) \Downarrow m_3$
- Notice that m_1 is `m` in Racket

Rule `E-while-f`

Math

$$\frac{e \Downarrow_m 0}{(m, \texttt{while}(e)\{p\}) \Downarrow m}$$

Racket

```
[(j:while? prog)
  (define v (r:eval vars (j:while-exp prog)))
  (cond [(equal? v 0) vars]
        [else ... ])]
```

Why?

- `(j:while? prog)` because: $(m, \texttt{while}(e)\{p\}) \Downarrow m$
- `(define v (r:eval (j:while-exp prog)))`
  Each evaluation above the fraction, eg $e \Downarrow 0$, becomes a define
- `(cond [(equal v 0) m]` because we are constraining the result $e \Downarrow 0$.
- Why are we returning `vars`?
  Because $(m, \texttt{while}(e)\{p\}) \Downarrow m$

Rule E-while-t

Math

$$\frac{e \Downarrow_m v \qquad v \neq 0 \qquad (m, p\ \mathbf{while}(e)\{p\}) \Downarrow m'}{(m, \mathbf{while}(e)\{p\}) \Downarrow m'}$$

Racket

```
[(j:while? prog)
  (define v (r:eval vars (j:while-cond prog)))
  (cond [(equal? v 0) vars]
        [else
          (define m'
            (r:eval vars (j:seq (j:while-body prog) prog)))
          m'])]
```

Why?

- `(j:seq (j:while-body prog) prog)` because $(m, p\ \mathbf{while}(e)\{p\}) \Downarrow m'$
- Evaluation above becomes a define
- Why are we returning `m'`, because $(m, \mathbf{while}(e)\{p\}) \Downarrow m'$

The implementation

```
(define/contract (j:eval vars prog)
  (→ hash? j:program? hash?)
  (cond
    [(j:assign? prog) (hash-set vars (j:assign-var prog) (r:eval vars (j:assign-exp prog)))]
    [(j:seq? prog) (j:eval (j:eval vars (j:seq-left prog)) (j:seq-right prog))]
    [(j:log? prog) (displayln (r:eval vars (j:log-exp prog))) vars]
    [(j:while? prog)
      (define v (r:eval vars (j:while-exp prog)))
      (cond [(equal? v 0) vars]
            [else (j:eval vars (j:seq (j:while-body prog) prog))])]))
```

Do we need to check the result of evaluating e_f?

$$\frac{e_f \Downarrow \lambda x.e_b \qquad e_a \Downarrow v_a \qquad e_b[x \mapsto v_a] \Downarrow v_b}{(e_f\ e_a) \Downarrow v_b}\ \texttt{(E-app)}$$

You may, but you do not need to. No other rule checks the result of evaluating e_f, thus you can ***assume*** it is a lambda. We are not interested in invalid inputs.

Garbage Collection
and our implementation of Heap

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

UMASS BOSTON

- 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

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
    ; 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 $\{x \mid x \in s_1 \wedge x \notin s_2\}$

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