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

UMass Boston

Rule `E-while-t`

Math

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

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 $(m, p\ \texttt{while}(e)\{p\}) \Downarrow m'$
- Evaluation above becomes a define
- Why are we returning `m'`, because $(m, \texttt{while}(e)\{p\}) \Downarrow m'$

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.