Tarea 6

Daniel Antonio Quihuis Hernandez

26 de octubre del October 2022

1 Problema 4.8

Show exactly where in our implementation of the store these operations take linear time rather than constant time.

La longitud y la anexion toman un tiempo lineal en *new-ref,* **por lo que** *new-ref,* **tiene un tiempo lineal.**

2 Problema 4.9

Implement the store in constant time by representing it as a Scheme vector. What is lost by using this representation?

```
(define (empty-store)
        (vector))

(define the-store 'uninitialized)

(define (get-store)
  the-store)

(define reference?
        (lambda (v)
                (integer? v)))

(define (extend-store store val)
        (let* ([store-size (vector-length store)]
                                [new-store (make-vector (+ store-size 1) (vec
        (vector-set! new-store (vector-length (new-store) val))))

(define newref val
        (let* ([new-store-info (extend-store the-store val)]
                                        [new-store (first new-store-info)]
                                        [next-ref (last new-store-info)])
```

```
                (set! the-store new-store)
                      next-ref))

(define (deref ref)
(vector-ref the-store ref))

(define (setref! ref val)
        (cond
        [(and (reference? ref) (< ref (vector-length the-store))) (vector-set
        [error 'setref! "No se puede cambiar la referencia"]))
```

3 Problema 4.10

Implement the begin expression as specified in exercise 4.4.

```
(value-of exp1 p sigma0) = (val1 ,sigma1)

(value-of (begin-exp exp1 '()) p sigma0) = (val1 , sigma1)

(value-of (begin-exp exp1 (cons exp2 exps)) p sigma0) = (value-of (begin-exp
```

4 Problema 4.11

Implement list from exercise 4.5

5 Problema 4.12

Our understanding of the store, as expressed in this interpreter, depends on the meaning of effects in Scheme. In particular, it depends on us knowing when these effects take place in a Scheme program. We can avoid this dependency by writing an interpreter that more closely mimics the specification. In this interpreter, value-of would return both a value and a store, just as in the specification. A fragment of this interpreter appears in figure 4.6. We call this a store-passing interpreter. Extend this interpreter to cover all of the language EXPLICIT-REFS. Every procedure that might modify the store returns not just its usual value but also a new store. These

6 Problema 4.13

Extend the interpreter of the preceding exercise to have procedures of multiple arguments.

es el mismo codigo para el 4.11, 4.12 y 4.13 pero para el codigo se utilizan archivos distintos asi como los test-lang.rkt

```
#lang racket/base

(require rackunit)
(require "../tarea-06/lang-4-12.rkt")

(check-equal? (run "2") (num-val 2))
(check-equal? (run "-(3, 3)") (num-val 0))
(check-equal? (run "-(3, 4)") (num-val -1))
(check-equal? (run "-(4, 3)") (num-val 1))
(check-equal? (run "zero?(0)") (bool-val #t))
(check-equal? (run "zero?(4)") (bool-val #f))
(check-equal? (run "if zero?(0) then 7 else 11") (num-val 7))
(check-equal? (run "if zero?(2) then 7 else 11") (num-val 11))
(check-equal? (run "let x = 5 in x") (num-val 5))
(check-equal? (run "let x = 5 in let x = 3 in x") (num-val 3))

(check-equal? (run "let f = proc (x) -(x, 11)
                    in (f (f 77))")
              (num-val 55))

(check-equal? (run "(proc (f) (f (f 77))
                     proc (x) -(x, 11))")
              (num-val 55))

(check-equal? (run "let x = 200
                    in let f = proc (z)
                                 -(z, x)
                       in let x = 100
                          in let g = proc (z)
                                       -(z, x)
                             in -((f 1), (g 1))")
              (num-val -100))

(check-equal? (run "letrec double(x) = if zero?(x)
                                       then 0
                                       else -((double -(x, 1)), -2)
                    in (double 6)")
              (num-val 12))

(check-equal? (run "let x = newref(0)
                    in letrec even(dummy) = if zero?(deref(x))
                                            then 1
                                            else begin setref(x, -(deref(x),
```

```
                                                    (odd 666)
                                                end
                                  odd(dummy) = if zero?(deref(x))
                                               then 0
                                               else begin setref(x, -(deref(x), 1
                                                    (even 666)
                                                end
                             in begin setref(x, 13);
                                      (odd 666)
                                end")
                  (num-val 1))

(check-equal? (run "let g = let counter = newref(0)
                            in proc (dummy)
                                 begin setref(counter, -(deref(counter), -1))
                                       deref(counter)
                                 end
                       in let a = (g 11)
                          in let b = (g 11)
                             in -(a, b)")
                  (num-val -1))

(check-equal? (run "let x = newref(newref(0))
                       in begin setref(deref(x), 11);
                                deref(deref(x))
                          end")
                  (num-val 11))

(check-equal? (run "deref(newref(7))")
                  (num-val 7))

(check-equal? (run "let x = newref(2)
                       in begin setref(let ref = newref(3)
                                       in begin setref(x, ref);
                                                ref
                                          end,
                                       5);
                                deref(deref(x))
                          end")
                  (num-val 5))
```