Tarea 06 - EXPLICIT REFS

Enrique Giottonini, Miguel Navarro

Octubre 26, 2022

Exercise 4.8 [⋆]

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

- En *newref-exp* se utiliza *append* el cual tiene un tiempo de ejecución dependiente de la cantidad de todos los argumentos en la lista (exceptuando el último) por lo que new-ref tiene tiempo de ejecución lineal. Esto siempre sucede.

- En *deref-exp* se utiliza *list-ref*, el cual tiene un tiempo de ejecución lineal por lo que *deref-exp* tiene un tiempo de ejecución lineal. Esto sucede siempre y cuando el *store* no es vacío y la referencia es válida.

- En *setref-exp* se utiliza *list-set* el cual tiene un tiempo de ejecución lineal por lo que *setref-exp* tiene tiempo de ejecución lineal. Esto sucede siempre y cuando el *store* no es vacío y la referencia es válida.

Exercise 4.9 [⋆]

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

El código se vuelve más complicado de entender, además por el uso de la estructura para el vector es más complicado manejar múltiples *stores*.

Exercise 4.10 [⋆]

Implement the begin expression as specified in exercise 4.4.

Sintáxis Concreta y Abstracta

Expression ::= begin Expression {; Expression}*

(begin-exp exp exps)

Semántica

```
(value-of (begin-exp exp exps) env σ) =
(if (null? exps)
        (value-of exp env σ)
        (value-of (begin-exp (first exps) (rest exps)) env σ1))
```

Donde σ_1 es el *store* resultante de:

```
(value-of exp env σ)
```

Exercise 4.11 [⋆]

Implement list from exercise 4.5.

Sintáxis Concreta y Abstracta

Expression ::= list (Expression*)

(list exps)

Semántica

```
(value-of (list-exp exps) env σ) =
(if (null? exps)
        ((null-val), σ)
        ((pair-val (cons val1 val2)), σ2)))
```

Donde:

```
(value-of (first exps) env σ) = (val1, σ1)
(value-of (list-exp rest exps) env σ1) = (val2, σ2)
```

Exercise 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 are packaged in a data type called answer. Complete this definition of value-of.

Desde un principio la implementamos de esta forma, por tanto las pruebas y la implementación están explicit-refs.rkt, y test-explicit-refs.rkt.

Exercise 4.13 [⋆⋆⋆]

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

Sintáxis Concreta y Abstracta

Expression ::= proc {Expression}$^+$ Expression
Expression ::= (Expression {Expression}$^+$)

(proc-exp params body)
(call-exp fun args)

Semántica

(value-of (proc-exp params body) env σ) = ((proc-val f), σ))

where:

f = (procedure p_1 p' env)
params = (p_1 p_2 p_3 ... p_n)
(proc-exp params body) = (proc-exp p_1 (proc-exp p_2 (... (proc-exp p_n body))))
p' = (proc-exp p_2 (proc-exp p_3 (... (proc-exp p_n body)))

(value-of (call-exp fun args) env s) = (val_n, s_n)

where:
args = (a_1 a_2 ... a_n)
(call-exp fun args) = (call-exp (call-exp (... (call-exp fun a_1) ...) a_{n-1}) a_n))))

Usando las definiciones que ya teniamos anteriormente, sabemos que:

```
(value-of (call-exp rator rand) env σ)
 = (let ([proc (expval->proc val_1)]
         [arg val_2])
     (apply-procedure proc arg σ_2))
```

where:
(val_1, σ_1) = (value-of rator env σ)
(val_2, σ_2) = (value-of rand env σ_1)

```
(define (apply-procedure proc val σ)
  (unless (procedure? proc)
    (error 'value-of "no es un procedimiento: ~e" proc))
  (let ([var (procedure-var proc)]
        [body (procedure-body proc)]
        [saved-env (procedure-saved-env proc)])
    (value-of body (extend-env var val saved-env) σ)))
```

(value-of (call-exp fun a_1) env σ) = (val_1, σ_1)
(value-of (call-exp (proc->expval val_1) a_2) env σ_1) = (val_2, σ_2)
...
(value-of (call-exp (proc->expval val_{n-1}) a_n) env σ_{n-1}) = (val_n, σ_n)