CS450

Structure of Higher Level Languages

Lecture 12: Implementing λ_E-Racket with environments

Tiago Cogumbreiro

Homework 4

Deadline: March 26, Tuesday 5:30pm EST

Today we will...

1. Motivate the need for environments
2. Introduce the λ_E-calculus formally
3. Discuss the implementation details of the λ_E-Racket
4. Discuss test-cases

In this unit we learn about...

- Implementing a formal specification
- Growing a programming language interpreter

Recall the λ-calculus

Syntax

$$e ::= v \mid x \mid (e_1\ e_2) \qquad v ::= n \mid \lambda x.e$$

Semantics

$$v \Downarrow v \ \texttt{(E-val)}$$

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

A complexity analysis on function-call

Let us focus consider our implementation of Micro-Racket, and draw our attention to function substitution.

Given a function call $(e_f\ e_a)$

1. We evaluate e_f down to a function $(\lambda(x)\ e_b)$
2. We evaluate e_a down to a value v_a
3. We evaluate $e_b[x \mapsto v_a]$ down to a value v_b

> What is the complexity of the substitution operation $[x \mapsto v_a]$?

A complexity analysis on function-call

Let us focus consider our implementation of Micro-Racket, and draw our attention to function substitution.

Given a function call $(e_f \; e_a)$

1. We evaluate e_f down to a function $(\lambda(x) \; e_b)$
2. We evaluate e_a down to a value v_a
3. We evaluate $e_b[x \mapsto v_a]$ down to a value v_b

> What is the complexity of the substitution operation $[x \mapsto v_a]$?

The run-time grows **linearly** on the size of the expression, as we must replace x by v_a in every sub-expression of e_b.

Can we do better?

Can we do better?

Yes, we can sacrifice some **space**

to improve the run-time **speed**.

Decreasing the run time of substitution

Idea 1: Use a lookup-table to bookkeep the variable bindings

Idea 2: Introduce closures/environments

λ_E-calculus: λ-calculus with environments

We introduce the evaluation of expressions down to values, parameterized by environments:

$$e \Downarrow_E v$$

The evaluation takes two arguments: an expression e, and an environment E. The evaluation returns a value v.

Attention!

Homework Assignment 4:

- Evaluation $e \Downarrow_E v$ is implemented as function `(s:eval env exp)` that returns a value `s:value`, an environment `env` is a `hash`, and expression `exp` is an `s:expression`.
- functions and structs prefixed with `r:` correspond to the λ-Racket language (Section 1).
- functions and structs prefixed with `s:` correspond to the λ_E-Racket language (Section 2)

λ_E-calculus: λ-calculus with environments

Syntax

$$e ::= v \mid x \mid (e_1\ e_2) \mid \lambda x.e \qquad v ::= n \mid (E, \lambda x.e)$$

Semantics

$$v \Downarrow_E v \qquad \texttt{(E-val)}$$

$$x \Downarrow_E E(x) \qquad \texttt{(E-var)}$$

$$\lambda x.e \Downarrow_E (E, \lambda x.e) \qquad \texttt{(E-clos)}$$

$$\frac{e_f \Downarrow_E (E_b, \lambda x.e_b) \qquad e_a \Downarrow_E v_a \qquad e_b \Downarrow_{\mathbf{E_b[x \mapsto v_a]}} v_b}{(e_f\ e_a) \Downarrow_E v_b} \qquad \texttt{(E-app)}$$

Overview of λ_E-calculus

Notable differences

1. Declaring a function is an *expression* that yields a function value (a closure), which packs the environment at creation-time with the original function declaration.
2. Calling a function unpacks the environment E_b from the closure and extends environment E_b with a binding of parameter x and the value v_a being passed

Environments

> An environment E maps variable bindings to values.

Constructors

- Notation $\emptyset$ represents the empty environment (with zero variable bindings)
- Notation $E[x \mapsto v]$ extends an environment with an new binding (overwriting any previous binding of variable x).

Accessors

- Notation $E(x) = v$ looks up value v of variable x in environment E

Implementing the new AST

Implementing the new AST

Values

$$v ::= n \mid (E, \lambda x.e)$$

Racket implementation

```
(define (s:value? v) (or (s:number? v) (s:closure? v)))
(struct s:number (value) #:transparent)
(struct s:closure (env decl) #:transparent)
```

Implementing the new AST

Expressions

$$e ::= v \mid x \mid (e_1\ e_2) \mid \lambda x.e$$

Racket implementation

```
(define (s:expression? e) (or (s:value? e) (s:variable? e) (s:apply? e) (s:lambda? e)))
(struct s:lambda (params body) #:transparent)
(struct s:variable (name) #:transparent)
(struct s:apply (func args) #:transparent)
```

How can we represent environments in Racket?

Hash-tables

TL;DR: A data-structure that stores pairs of key-value entries. There is a lookup operation that given a key retrieves the value associated with that key. Keys are unique in a hash-table, so inserting an entry with the same key, replaces the old value by the new value.

- Hash-tables represent a (partial) injective function.
- Hash-tables were covered in CS310.
- Hash-tables are also known as maps, and dictionaries. We use the term hash-table, because that is how they are known in Racket.

Hash-tables in Racket

Constructors

1. Function `(hash k1 v1 ... kn vn)` a hash-table with the given key-value entries. Passing zero arguments, `(hash)`, creates an empty hash-table.
2. Function `(hash-set h k v)` copies hash-table `h` and adds/replaces the entry `k v` in the new hash-table.

Accessors

- Function `(hash? h)` returns `#t` if `h` is a hash-table, otherwise it returns `#f`
- Function `(hash-count h)` returns the number of entries stored in hash-table `h`
- Function `(hash-has-key? h k)` returns `#t` if the key is in the hash-table, otherwise it returns `#f`
- Function `(hash-ref h k)` returns the value associated with key `k`, otherwise aborts

Hash-table example

```
(define h (hash))                          ; creates an empty hash-table
(check-equal? 0 (hash-count h))            ; we can use hash-count to count how many entries
(check-true (hash? h))                     ; unsurprisingly the predicate hash? is available

(define h1 (hash-set h "foo" 20))          ; creates a new hash-table where "foo" is bound to 20
(check-equal? (hash "foo" 20) h1)          ; (hash-set (hash) "foo" 20) = (hash "foo" 20)

(define h2 (hash-set h1 "foo" 30))
(check-equal? (hash "foo" 30) h2)          ; in h2 "foo" is the key, and 30 the value
(check-equal? 30 (hash-ref h2 "foo"))      ; ensures that hash-ref retrieves the value of "foo"
(check-equal? (hash "foo" 20) h1)          ; h1 remains the same
```

Encoding environments with hash-tables

- How can we encode an empty environment $\emptyset$:

Encoding environments with hash-tables

- How can we encode an empty environment $\emptyset$: `(hash)`
- How can we encode a lookup $E(x)$:

Encoding environments with hash-tables

- How can we encode an empty environment $\emptyset$: `(hash)`
- How can we encode a lookup $E(x)$: `(hash-ref E x)`
- How can we encode environment extension $E[x \mapsto v]$:

Encoding environments with hash-tables

- How can we encode an empty environment $\emptyset$: `(hash)`
- How can we encode a lookup $E(x)$: `(hash-ref E x)`
- How can we encode environment extension $E[x \mapsto v]$: `(hash-set E x v)`

Test-cases

Function `(check-s:eval? env exp val)` is given in the template to help you test effectively your code.

> The use of `check-s:eval` is **optional**. You are encouraged to play around with `s:eval` directly.

1. The first parameter is an S-expression that represents an *environment*. The S-expression must be a list of pairs representing each variable binding. The keys must be symbols, the values must be serialized λ_E values

```
[] ; The empty environment
[ (x . 1) ]  ; An environment where x is bound to 1
[ (x . 1) (y . 2) ] ; An environment where x is bound to 1 and y is bound to 2
```

2. The second parameter is an S-expression that represents the a valid λ_E **expression**
3. The third parameter is an S-expression that represents a valid λ_E **value**

Serialized expressions in λ_E

> Each line represents a **quoted** expression as a parameter of function `s:parse-ast`. For instance, `(s:parse-ast '(x y))` should return `(s:apply (s:variable 'x) (list (s:variable 'y)))`.

```
1                              ; (s:number 1)
x                              ; (s:variable 'x)
(closure [(y . 20)] (lambda (x) x))
; (s:closure
;    (hash (s:variable 'y) (s:number 20))
;    (s:lambda (list (s:variable 'x)) (list (r:variable 'x))))
(lambda (x) x)                 ; (s:lambda (list (s:variable 'x)) (list (s:variable 'x)))
(x y)                          ; (s:apply (s:variable 'x) (list (s:variable 'y)))
```

Test cases

```
; x is bound to 1, so x evaluates to 1
(check-s:eval? '[(x . 1)] 'x 1)
; 20 evaluates to 20
(check-s:eval? '[(x . 2)] 20 20)
; a function declaration evaluates to a closure
(check-s:eval? '[] '(lambda (x) x) '(closure [] (lambda (x) x)))
; a function declaration evaluates to a closure; notice the environment change
(check-s:eval? '[(y . 3)] '(lambda (x) x) '(closure [(y . 3)] (lambda (x) x)))
; because we use an S-expression we can use brackets, curly braces, or parenthesis
(check-s:eval? '{(y . 3)} '(lambda (x) x) '(closure [(y . 3)] (lambda (x) x)))
; evaluate function application
(check-s:eval? '{} '((lambda (x) x) 3)  3)
; evaluate function application that returns a closure
(check-s:eval? '{} '((lambda (x) (lambda (y) x)) 3)  '(closure {[x . 3]} (lambda (y) x)))
```