CS450

Structure of Higher Level Languages

Lecture 20: Homework 4 preparation

Tiago Cogumbreiro

Today we will...

- Go through the implementation of language λ_E
- Write some examples that manipulate hash-tables
- Go through some examples of λ_E programs

Implementing the new AST

Implementing the new AST

Values

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

Racket implementation

```
(define (e:value? v) (or (e:number? v) (e:closure? v)))
(struct e:number (value) #:transparent)
(struct e: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 (e:expression? e) (or (e:value? e) (e:variable? e) (e:apply? e) (e:lambda? e)))
(struct e:lambda (params body) #:transparent)
(struct e:variable (name) #:transparent)
(struct e: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

Test-cases

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

> The use of `check-e:eval` is **optional**. You are encouraged to play around with `e: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 `e:parse-ast`. For instance, `(e:parse-ast '(x y))` should return `(e:apply (e:variable 'x) (list (e:variable 'y)))`.

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

Test cases

```
; x is bound to 1, so x evaluates to 1
(check-e:eval? '[(x . 1)] 'x 1)
; 20 evaluates to 20
(check-e:eval? '[(x . 2)] 20 20)
; a function declaration evaluates to a closure
(check-e:eval? '[] '(lambda (x) x) '(closure [] (lambda (x) x)))
; a function declaration evaluates to a closure; notice the environment change
(check-e: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-e:eval? '{(y . 3)} '(lambda (x) x) '(closure [(y . 3)] (lambda (x) x)))
; evaluate function application
(check-e:eval? '{} '((lambda (x) x) 3)  3)
; evaluate function application that returns a closure
(check-e:eval? '{} '((lambda (x) (lambda (y) x)) 3)  '(closure {[x . 3]} (lambda (y) x)))
```