CS450

Structure of Higher Level Languages

Lecture 12: Function calls with environments

Tiago Cogumbreiro

Lexical Scope

Lexical Scope

- ***Binding:*** association between a variable and a value.
- ***Scope*** of a binding: the text where occurrences of this name refer to the binding
- ***Lexical (or static) scope***: the innermost lexically-enclosing construct declaring that variable

> **Did you know?** In Computer Science, ***static analysis*** corresponds to analyzing the source code, without running the program.

```
(define (f)
  (define x 10) ; visible: f
  (define y 20) ; visible: f, f.x
  (+ x y))      ; visible: f, f.x, f.y

; visible: f
(define x 1)
; visible: f, x
(define y (+ x 1))
; visible: f, x, y
(check-equal? (f) 30) ; yields (+ f.x f.y)
```

Dynamic Scope

Lexical scope vs dynamic scope

- Lexical scoping is the default in all popular programming languages
- With lexical scoping, we can analyze the source code to identify the scope of ***every*** variable
- With lexical scoping, the programmer can reason about each function independently

What is a dynamic scope?

- Variable scope depends on the calling context
- Renders all variables global

> appeared in McCarthy's Lisp 1.0 as a bug and became a feature in all later implementations, such as MacLisp, Gnu Emacs Lisp.

Moreau, L. Higher-Order and Symbolic Computation (1998) 11: 233. DOI:10.1023/A:1010087314987

```
;; NOT VALID RACKET CODE!!!
(define (f) x)

(define (g x) (f))
(check-equal? (g 10) 10)

(define x 20)
(check-equal? (f) 20)
```

Example

- What is the result of evaluating `(g)`?

```
(define x 1)

(define (f y) (+ y x))

(define (g)
  (define x 2)
  (define y 3)
  (f (+ x y)))

(check-equal? (g) ???)
```

Example

- What is the result of evaluating `(g)`?

```
(define x 1)

(define (f f:y) (+ f:y x))

(define (g)
  (define g:x 2)
  (define g:y 3)
  (f (+ g:x g:y)))

(check-equal? (g) 6)
```

Why lexical scoping?

- Lexical scoping is important for using functions-as-values
- To implement our Mini-Racket we will need to implement lexical scoping

Example

- What is the result of evaluating `(g)`?

```
(define (g) x)

(define x 10)

(check-equal? (g) ??)
```

Example

What is the result of evaluating `(g)`?

```
(define (g) x)
; (g) throws an error here
(define x 10)

(check-equal? (g) 10)
```

We can define a function `g` that refers to an undefined variable `x`; variable `x` must be defined before calling `g`.
In Racket, variable definition produces a side-effect, as the definition of `x` impacted a previously defined function `g`. ***In Module 5, we implement the semantics of `define`.***

Accessing variables outside a function

The body of a function can refer to variables defined outside of that function.

> It can access variables is defined outside of the function, but where exactly?

The function's body can access any variable that is accessible/visible when the function is **defined**, which is known as the **lexical scope**.

In the following example, the function returns `3` and not `10`, even though variable `x` is now `10`.

```
; For a given x create a new function that always returns x
(define (getter x) (lambda () x))
(define get3 (getter 3)) ; At creation time, x = 3
(define x 10)
(check-equal? 3 (get3)) ; At call time, x = 10
```

Function closures

Recall that functions capture variables

Function closure

- **A function closure is the return value of function declaration** (***i.e.***, the function ***value***)
- **Definition:** A function closure is a pair that stores a function declaration and its lexical environment (***i.e.***, the state of each variable captured by the function declaration)
- The technique of creating a function closure is used by compilers/interpreters to represent function values

Recall that function declarion $\neq$ function definition:

- Function declaration: `(` `lambda` `(` *`variable*`* `)` *`term+`*`)`
- Function definition: `(` `define` `(`*`variable+`* `)` *`term+`* `)`

Now we know what a function closure is

1. How to compute the variables in a closure
2. When to set the values of each variable in a closure

Function closures: captured variables

> ***It is crucial for us to know how variables are captured in Racket.***

Given an expression the set of free variables can be defined inductively:

- When the expression is a variable `x`, the set of free variables is `{ x }`.
- When the expression is a `(lambda (x) e)`, the set of free variables is that of expression `e` minus variable `x`.
- When the expression is a function application `(e1 e2)`, the set of free variables is the union of the set of free variables of `e1` and the set of free variables of `e2`.

Captured variables: Given an expression `(lambda (x) e)` a function closure ***captures*** the set of free variables of expression `(lambda (x) e)`.

Captured variables examples

Let us compute fv `(lambda (x) (+ x y))`:

1. The free-variables of a λ are the free variables of the body of the function minus parameter x.

fv `(lambda (x) (+ x y))` = fv `(+ x y)` $\setminus\{x\}$

2. We are now in a case of function application, which is the union of the free variables of each of its sub expressions.

fv `(+ x y)` $\setminus\{x\} = \big(\mathrm{fv}(+) \cup \mathrm{fv}(x) \cup \mathrm{fv}(y)\big) \setminus \{x\}$

4. Finally, we reach the case where each argument of `free-vars` is a variables.

$$\big(\mathrm{fv}(+) \cup \mathrm{fv}(x) \cup \mathrm{fv}(y)\big) \setminus \{x\} = \big(\{+\} \cup \{x\} \cup \{y\}\big) \setminus \{x\} = \{+, x, y\} \setminus \{x\} = \{+, y\}$$

What creates an environment?

Definition: At any execution point there is an environment, which maps each variable to a value.

What creates environments:

- Each branch inside a `cond` creates an environment
- The body of a function creates an environment

What updates an environment:

- The arguments of a `lambda` are added to the function's body environment
- A `(define x e)` updates the current environment by adding/updating variable `x` and setting it to the value that results from evaluating `e`

Example 1: capture an argument

> The `lambda` is capturing `x` as the parameter of `getter` at creation time, so when we call `(getter3)` we get `(lambda () 3)`.

```
(define (getter x)
  (lambda () x)) ; getter:x

(define get3 (getter 3)) ; getter:x = 3; (lambda () getter:x)
(check-equal? 3 (get3))
```

Example 3: `cond` starts a new scope

> Function `getter` captured `x` at the outermost scope (the `x` captured at function declaration time). Inside the branches of `cond` we have ***a new scope***, which means that `getter` is unaffected by the redefinition of `x`.

```
(define (getter) x) ; root.x
(define x 10)         ; root.x = 10
; Each branch of the cond creates a new environment
; so it does not affect getter
(cond [#t (define x 20) (check-equal? 10 (getter))])
(check-equal? 10 (getter))
```

Example 3: define shadows parameters

> Function `getter` returns variable `x` from the environment of function `f`. When calling `f 20` the last value of variable `x` in the scope of `f` is `10`, due to `(define x 10)`, which overwrites the function's parameter `x=20`.

```
(define (f x)
  (define (getter) x) ; f.x = ?
  (define x 10)       ; f.x = 10
  getter)

(define g (f 20))
(check-equal? 10 (g))
```

The λ-calculus is slow

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 `(e:eval env exp)` that returns a value `e:value`, an environment `env` is a `hash`, and expression `exp` is an `e:expression`.
> - functions and structs prefixed with `s:` correspond to the λ_S language (Section 1).
> - functions and structs prefixed with `e:` correspond to the λ_E language (Section 2)

UMass Boston

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

Church's encoding

Chuch's encoding

- Alonzo Church created the λ-calculus
- Church's Encoding is a treasure trove of λ-calculus expressions: it shows how natural numbers can be encoded
- Let us go through Church's encoding of booleans
- Examples taken from Colin Kemp's PhD thesis (page 17)

Encoding Booleans with λ-terms

Why? Because you will be needing test-cases.

```racket
(require rackunit)
(define ns (make-base-namespace))
(define (run-bool b) (((eval b ns) #t) #f))

; True
(define TRUE '(lambda (a) (lambda (b) a)))
(define FALSE '(lambda (a) (lambda (b) b)))
(define (OR a b) (list (list a TRUE) b))
(define (AND a b) (list (list a b) FALSE))
(define (NOT a) (list (list a FALSE) TRUE))
(define (EQ a b) (list (list a b) (NOT b)))
; Test
(check-equal?
    (run-bool (EQ TRUE (OR (AND FALSE TRUE) TRUE)))
    (equal? #t (or (and #f #t) #t)))
```

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)))
```

UMass Boston

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)))
```