CS450

Structure of Higher Level Languages

Lecture 31: Dynamic binding

Tiago Cogumbreiro

Today we will learn...

- Revisit dynamic binding
- Dynamic binding to control globals
- Dynamic binding to control testing

Dynamic scoping in Racket

`parameterize`

Static versus dynamic scoping

Static Scoping

> **Static binding:** variables are captured at creation time

```
(define x 1)

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

(define (g)
  (define x 20)
  (define y 3)
  (f (+ x y)))

(check-equal? (g) (+ 23 1))
```

Dynamic Scoping

> **Dynamic binding:** variables depends on the calling context

```
(define x 1)

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

(define (g)
  (define x 20)
  (define y 3)
  (f (+ x y)))
; NOT VALID RACKET CODE
(check-equal? (g) (+ 23 20))
```

Why dynamic scoping?

1. A controlled way to represent global variables
2. A technique to make code testable

Dynamic scoping example

Dynamic scoping In Racket

```
(define x (make-parameter 1))
(define (f y) (+ y (x)))

(define (g)
  (parameterize ([x 20])
    (define y 3)
    (f (+ (x) y))))

(check-equal? (g) (+ 23 20))
```

Pseudo-Racket dynamic scoping

```
(define x 1)

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

(define (g)
  (define x 20)
  (define y 3)
  (f (+ x y)))
; NOT VALID RACKET CODE
(check-equal? (g) (+ 23 20))
```

- Function `make-parameter` returns a reference to a dynamically scoped memory-cell
- Calling a parameter without parameter returns the contents of the memory-cell
- Use `parameterize` to overwrite the memory-cell

Dynamic binding

Globals

Dynamic binding: controlled globals

> We can define different globals in different contexts.

```
(define buff (open-output-string))
(parameterize ([current-output-port buff])
  ; In this context, the standard output is a string buffer.
  (display "hello world!"))
(check-equal? (get-output-string buff) "hello world!")
```

Racket uses parameters to allow extending the behavior of many features:

- command line parameters
- standard output stream (known as a port)
- formatting options (eg, default implementation to print structures)

Dynamic binding

Testing

Dynamic binding: making code testable

> Consider an excerpt of Homework 5. We would like to be able to test each function independently. How?

```
(define (s:eval-exp mem env exp)
  (define (on-app mem env exp)
    ;; ...
    ;; Eb \Downarrow Eb vb
    (s:eval-term mem3 Eb (s:lambda-body lam)))
  (cond
    ;; ...
    [(s:apply? exp) (on-app mem env exp)]

(define (s:eval-term mem env term)
  (cond
    ; ...
    [else (s:eval-exp mem env term)]))
```

Dynamic binding: making code testable

- In Homework 4, we added a function parameter to test `r:eval` independently from `r:subst`.
- This extra function parameter was confusing to some students.
- This choice made the function interface more verbose than needed.
- More arguments, more chance of mistakes! Do we call `subst` or `s:subst`?

How can we use dynamic binding

to improve the testing design of `r:eval`?

Dynamic binding: making code testable

- Create a parameter per global function that you want to make testable
- Internal calls should target the ***parameter*** and not the global variable

Before

```
(define (r:eval subst exp)
  (cond
    [...
     (define eb' (subst eb x va))
     ...]))
```

Dynamic binding: making code testable

- Create a parameter per global function that you want to make testable
- Internal calls should target the ***parameter*** and not the global variable

Before

```
(define (r:eval subst exp)
  (cond
    [...
     (define eb' (subst eb x va))
     ...]))
```

After

```
(define r:subst-impl
    (make-parameter r:subst))

(define (r:eval exp)
  (cond
    [...
     (define eb' ((r:subst-impl) eb x va))
     ...]))
```

Dynamic binding: making code testable

> Consider an excerpt of Homework 5. We would like to be able to test each function independently. How?

```
(define (s:eval-exp mem env exp)
  (define (on-app mem env exp)
    ; ...
    ((s:eval-term-impl) mem3 Eb (s:lambda-body lam)))
  (cond ; ...
    [(s:apply? exp) (on-app mem env exp)]
(define s:eval-exp-impl (make-parameters s:eval-exp))

(define (s:eval-term mem env term)
  (cond ; ...
    [else ((s:eval-exp-impl) mem env term)]))
(define s:eval-term-impl (make-parameters s:eval-term))
```

Dynamic binding: making code testable

Usage example:

```
(parameterize ([s:eval-expr-impl (lambda (mem env expr) (s:number 10))])
  ; Now x is evaluated to (s:number 10) and y evaluates to (s:number 10)
  (eval-term? '[x y] 10))
```

We can test eval-term without implementing eval-exp!

This testing technique is known as ***mocking***.