Functional Programming

Lecture 4: Closures and lazy evaluation

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

Last lecture

- Functions with variable number of arguments
- Higher order functions
  - map, foldr, foldl, filter, compose

Binding scopes

A portion of the source code where a value is bound to a given name

- Lexical scope
  - functions use bindings available where defined
- Dynamic scope
  - functions use bindings available where executed

Ben Wood's slides

Eval

Eval is a function defined in the top level context

```
(define x 5)
(define (f y)
  (eval '(+ x y)))
```

Fails because of lexical scoping

Otherwise the following would be problematic

```
(define (eval x)
      (eval-expanded (macro-expand x)))
```

Eval in R5RS

```
(eval expression environment)
```

Where the environment is one of

- `(interaction-environment)` (global defines)
- `(scheme-report-environment x)` (before defines)
- `(null-environment x)` (only special forms)

None of them allows seeing local bindings

Cons

```
(define (my-cons x y)
  (lambda (m)
    (m x y)))

(define (my-car p)
  (p (lambda (x y) x)))

(define (my-cdr s)
  (p (lambda (x y) y)))
```

Currying

Transforms function to allow partial application

not curried $f: A \times B \rightarrow C$

curried $f: A \rightarrow (B \rightarrow C)$

not curried $f: A \times B \times C \rightarrow D$

curried $f: A \rightarrow (B \rightarrow (C \rightarrow D))$

Picture: Haskell Curry from Wikipedia

Currying

```
(define (my-curry1 f)
  (lambda args
    (lambda rest (apply f (append args rest)))))

(define (my-curry f)
  (define (c-wrap f stored-args)
    (lambda args
      (let ((all (append stored-args args)))
        (if (procedure-arity-includes?
                                f (length all))
            (apply f all)
            (c-wrap f all)))))
  (c-wrap f '()))
```

Lazy if using and/or

```
(define (my-if t a b)
        (or (and t a) b))

(define (my-lazy-if t ac bc)
  (or (and t (ac)) (bc)))

(my-lazy-if (< 1 2)
      (lambda () (print "true"))
      (lambda () (print "false"))
)
```

Syntax macros

```
(define-syntax macro-if
  (syntax-rules ()
     ((macro-if t a b)
         (my-lazy-if
             t
             (lambda () a)
             (lambda () b)
         )
)))
```

Lazy evaluation

`lambda ()` can be used to delay evaluation

Delayed function is called "thunk"

Useful for

- Lazy evaluation of function argument (call by name)
- Streams – potentially infinite lists

Delay / Force

```
(define-syntax w-delay
  (syntax-rules ()
     ((w-delay expr) (lambda () expr))))

(define-syntax w-force
  (syntax-rules ()
     ((w-force expr) (expr))))
```

Streams

```
(define (ints-from n)
  (cons n
        (w-delay (ints-from (+ n 1)))))

(define nat-nums
      (ints-from 1))

(define (first l) (car l))
(define (rest l)
  (if (pair? (cdr l)) (cdr l)
      (w-force (cdr l))))
```

Lazy map

```
(define (lazy_map f list)
  (if (null? list) list
      (cons (f (car list))
            (w-delay (lazy_map f
                               (rest list))))
   )
)
```

Home assignment 2

Population evaluator

- evaluate a collection of programs for the robot
- (evaluate

  <prgs>

  <pairs>

  <threshold>

  <stack_size>)

Summary

- Binding scopes
  - Lexical vs. dynamic
- Closures
  - Code + environment pointer
  - Way to "store data" in a function
  - Tool for lazy evaluation
- Streams