



Functional Programming

Lecture 5: Imperative aspects of Scheme

Viliam Lisý

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

viliam.lisy@fel.cvut.cz

Last lecture

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

Streams recap.

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

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

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

Stream map

```
(define (smap proc . args)
  (if (null? (car args)) '()
      (cons
        (apply proc
                (map first args))
        (w-delay (apply smap
                        proc
                        (map rest args))
                )))
  )
```

Recursive Streams

```
(define ones (cons 1 (w-delay ones)))  
(define (add-streams s1 s2)  
        (stream-map + s1 s2))  
(define integers  
  (cons 1  
        (w-delay (add-streams  
                  ones integers))))  
(define fibs  
  (cons 0 (cons 1  
              (w-delay (add-streams  
                        (rest fibs) fibs))))))
```

Imperative aspects of scheme

Until now, we did not need any mutable states

- Fully thread safe

- No exact (defensive) copies

- No temporal coupling

Do not use in assignments!!!

Set!

```
(set! id expr)
```

Assigns the value of `expr` to variable `id`

The `id` has to be already defined

It is not the same as redefining the variable

```
(define (foo)
  (define x 4)
  x)
```

```
(define (bar)
  (set! x 4)
  x)
```

We can have a state now!

```
(define counter 1)
```

```
(define (inc-counter)  
  (set! counter (+ counter 1)))
```

```
> (inc-counter)
```

```
> (inc-counter)
```

```
> (inc-counter)
```

Promise

Our weak delay/force may evaluate many times

State can be used to save the evaluated result

Delay with memoization :

```
(make-promise (lambda () expression))
(define (make-promise proc)
  (let ((already-run? #f)
        (result #f))
    (lambda ()
      (if already-run? result
          (begin (set! result (proc))
                 (set! already-run? #t)
                 result))))))
(define (force p) (p))
```

Vectors

- Heterogeneous objects
- Indexed by integers, starting from 0
- Typically faster and smaller than lists

```
(vector obj ...)
```

```
(make-vector k)
```

```
(make-vector k fill)
```

```
(vector-ref vector k)
```

```
(vector-set! vector k obj)
```

```
(list->vector list)
```

Iteration

```
(do ((<variable1> <init1> <step1>) ...)
    (<test> <expression> ...)
    <command> ...)
```

Create a vector initialized 0...length-1

```
(let ((vec (make-vector 5)))
  (do ((i 0 (+ i 1)))
      ((= i 5) vec)
      (vector-set! vec i i)))
```


Letrec

```
(letrec ((x1 e1) ... (xn en)) body)
```

Is a macro expanding to

```
(let ((x1 undefined) ... (xn undefined))  
  (let ((t1 e1) ... (tn en))  
    (set! x1 t1 )  
    ...  
    (set! xn tn ) )  
  body)
```

All expressions must evaluate without evaluating

x_1, \dots, x_n

Closures in Impure Languages

Closures store data with functions

```
(define (counter)
  (let ((i 0))
    (lambda ()
      (begin (set! i (+ 1 i))
              i))))
```

Random

```
(define random
  (let ((a 69069) (c 1)
        (m (expt 2 32)) (seed 19380110))
    (lambda new-seed
      (if (pair? new-seed)
          (set! seed (car new-seed))
          (set! seed
                (modulo (+ (* seed a) c) m)))
        (/ seed m))))
```

"Classes and objects"

Encapsulate data and related functions

Closures combine data with functions

Assignment allows modifying the data

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
                balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                        m))))
  dispatch)
```

Summary

- We do not need to modify the state
- It breaks nice properties of FP
- It can sometimes be useful
 - random access in $O(1)$
 - objects
 - ...