Functional Programming

Lecture 5: Imperative aspects of Scheme

Viliam Lisý
Rostislav Horčík
Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague
viliam.lisy@fel.cvut.cz
xhorcik@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 s)
  (cond ((null? s) '())
        (else
         (cons
          (f (first s))
          (w-delay (lazy-map f (rest s)))))))
```

Stream map

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

Implicitly defined streams

```
(define ones (cons 1
                   (w-delay ones)))
(define (sadd s1 s2) (smap + s1 s2))

(define nats (cons 1
  (w-delay (sadd ones nats))))

(define fibs (cons 0 (cons 1
  (w-delay (sadd fibs (rest fibs))))))
```

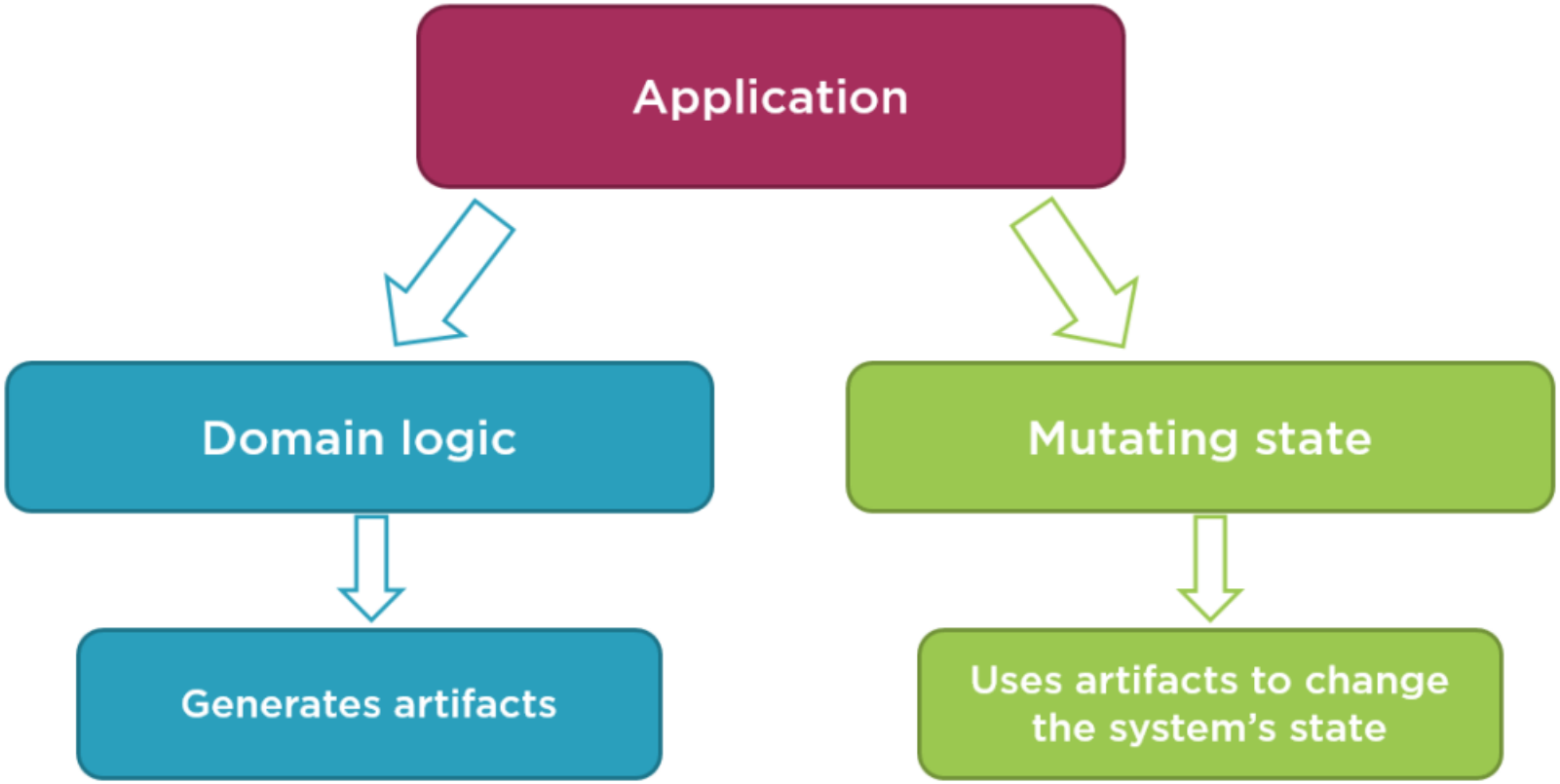
Imperative aspects of scheme

Until now, we did not need any mutable states

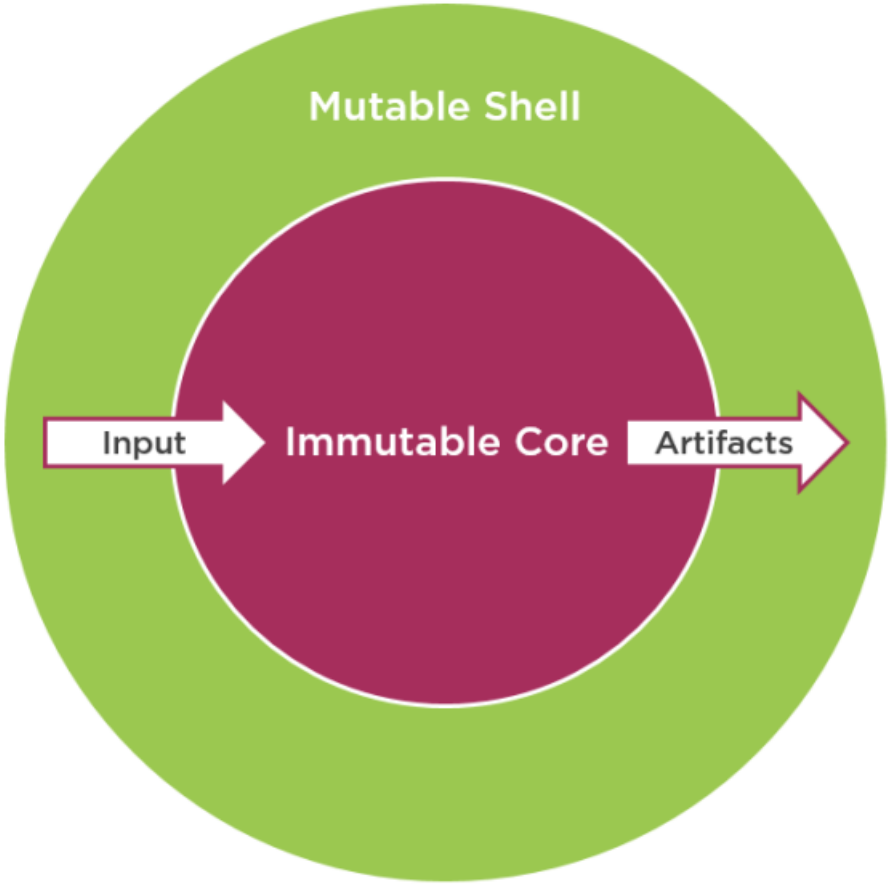
- Fully thread safe
- No exact (defensive) copies
- No temporal coupling

Do not use in assignments!!!

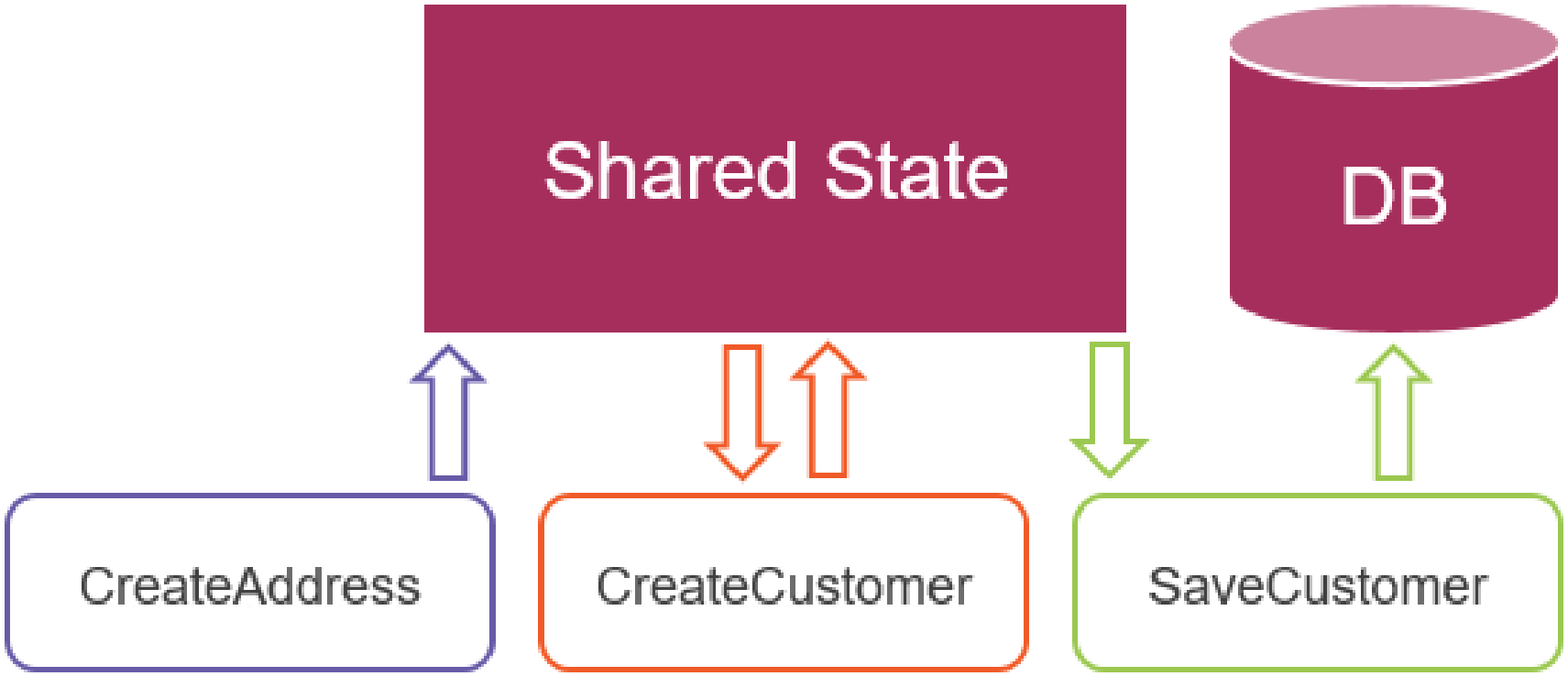
Immutable architecture



Immutable architecture



Temporal coupling



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 x 1)
(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)) counter)

> (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 :

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

Vectors

- Heterogeneous objects
- Indexed by integers, starting form 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

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

Letrec

Sometimes, we need all names available in the expressions

```
(letrec
    ((fact (lambda (n)
             (if (= n 1)
                 1
                 (* n (fact (- n 1)))))))
  (fact 10))
```

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

```
x1,...,xn
```

Closures in Impure Languages

Closures store data with functions

```
(define (count-clo)
  (let ((i 0))
    (lambda ()
      (begin (set! i (+ i 1))
             i))))
```

Random

```
(define random
  (let ((a 69069)
        (b 1)
        (m (expt 2 32))
        (seed 20200323))
    (lambda args
      (if (null? args)
          (begin
           (set! seed
                 (modulo (+ (* a seed) b) m))
           (/ seed m))
          (set! seed (car args))))))
```

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)
  - I/O operations
  - objects with states
  - …