CS450

Structure of Higher Level Languages

Lecture 13: Shared mutable state and immutability

Tiago Cogumbreiro

Policy on academic honesty

1. Any two students sharing code in a submission **will void both submissions**.
2. Any repeated incident will be reported to the department, and the student **will fail the course with an F**.

Students may choose to withdraw any homework submission that has not been voided.

Please read on...

Acknowledging Intellectual Debts. Nurit Haspel, Ethan Bolker, Carl Offner.

Student conduct

> Students are required to adhere to the University Policy on Academic Standards and Cheating, to the University Statement on Plagiarism and the Documentation of Written Work, and to the Code of Student Conduct as delineated in the catalog of Undergraduate Programs. The Code is available online at: `www.umb.edu/life_on_campus/policies/code/`

Anti-Cheating Software

Moss Results

Thu Mar 21 09:40:13 PDT 2019

Options -l scheme -m 10

Scheme files

[How to Read the Results | Tips | FAQ | Contact | Submission Scripts | Credits]

File 1	File 2	Lines Matched
assignment_154296_export/submission .rkt (58%)	assignment_154296_export/submission_ .rkt (60%)	97
assignment_154296_export/submission .rkt (52%)	assignment_154296_export/submission_ .rkt (51%)	85
assignment_154296_export/submission .rkt (50%)	assignment_154296_export/submission_ .rkt (47%)	74
assignment_154296_export/submission .rkt (99%)	assignment_154296_export/submission_ .rkt (99%)	113
assignment_154296_export/submission .rkt (24%)	assignment_154296_export/submission_ .rkt (34%)	66
assignment_154296_export/submission .rkt (32%)	assignment_154296_export/submission_ .rkt (39%)	45
assignment_154296_export/submission .rkt (20%)	assignment_154296_export/submission_ .rkt (29%)	61
assignment_154296_export/submission .rkt (20%)	assignment_154296_export/submission_ .rkt (30%)	61
assignment_154296_export/submission .rkt (28%)	assignment_154296_export/submission_ .rkt (35%)	35
assignment_154296_export/submission .rkt (27%)	assignment_154296_export/submission_ .rkt (35%)	33

Cheating sample

assignment_154296_export/submission[redacted].rkt (99%)		assignment_[redacted]_export/submissio (99%)
2-114		2-108

assignment_154296_export/submission_ rkt

```
#lang racket

(require "ast.rkt")
(require "hw1.rkt")
(require rackunit)
(provide (all-defined-out))

;use if for number 1
;

;; Exercise 1.a: Read-write cell
;; Solution has 3 lines.
(define (rw-cell x)
        ((lambda (a)
                (lambda (b) (if (equal? 1 (length b))
                                (rw-cell (car b))
                                a)))
        x))


;; Exercise 1.b: Read-only cell
;; Solution has 4 lines.
(define (ro-cell x)
        ((lambda (a)
                (lambda (b) (if (equal? 1 (length b))
                                (ro-cell a)
```

assignment_154296_export/submission_ .rkt

```
#lang racket

(require "ast.rkt")
(require "hw1.rkt")
(require rackunit)
(provide (all-defined-out))

;; Exercise 1.a: Read-write cell
;; Solution has 3 lines.
(define (rw-cell x)
        ((lambda (a)
                (lambda (b) (if (equal? 1 (length b))
                                (rw-cell (car b))
                                a)))
        x))


;; Exercise 1.b: Read-only cell
;; Solution has 4 lines.
(define (ro-cell x)
        ((lambda (a)
                (lambda (b) (if (equal? 1 (length b))
                                (ro-cell a)
                                a)))
        x))
```

Today we will...

1. Study adding `define` to our language
2. Introduce mutation in an immutable setting
3. Introduce Racket's contracts

λ_D-calculus: λ-calculus with definitions

Syntax

$$t ::= e \mid t ; t \mid (\texttt{define}\ x\ e)$$

$$e ::= v \mid x \mid (e_1\ e_2) \mid \lambda x.t \qquad v ::= n \mid (E, \lambda x.t) \mid \texttt{void}$$

- New grammar rule: *terms*
- A program is now a non-empty sequence of terms
- Since we are describing the *abstract* syntax, there is no distinction between a basic and a function definition
- Since evaluating a definition returns a void, we need to update values

Values

> We add `void` to values.

$$v ::= n \mid (E, \lambda x.t) \mid \mathtt{void}$$

Racket implementation

```
;; Values
(define (s:value? v) (or (s:number? v) (s:closure? v) (s:void? v)))
(struct s:number (value) #:transparent)
(struct s:closure (env decl) #:transparent)
(struct s:void () #:transparent)
```

Expressions

Expressions remain unchanged.

$$e ::= v \mid x \mid (e_1\ e_2) \mid \lambda x.t$$

Racket implementation

```
(define (s:expression? e) (or (s:value? e) (s:variable? e) (s:apply? e) (s:lambda? e)))
(struct s:variable (name) #:transparent)
(struct s:apply (func args) #:transparent)
(struct s:lambda (params body) #:transparent)
```

Terms

- We implement terms below.

$$t ::= e \mid t; t \mid (\texttt{define}\ x\ e)$$

Racket implementation

```
(define (s:term? t) (or (s:expression? t) (s:seq? t) (s:define? t)))
(struct s:seq (fst snd) #:transparent)
(struct s:define (var body) #:transparent)
```

The body of a function declaration is a single term

The body is no longer a list of terms!

- A sequence is not present in the concrete syntax, but it simplifies the implementation and formalism (see reduction)

Parsing datum into AST terms

- Our parser handles multiple terms in the body of a function declaration.
- Function `s:parse1` parses a single term.

```
(check-equal?
  (s:parse1 '(lambda (x) x y z))
  (s:lambda (list (s:variable 'x))
    (s:seq (s:variable 'x)
      (s:seq (s:variable 'y) (s:variable 'z)))))
```

Parsing datum into AST terms

- The body of a function can have one or more definitions, values, or function calls.

```
(check-equal?
  (s:parse1 '(lambda (x) (define x 3) x))
  (s:lambda (list (s:variable 'x))
    (s:seq (s:define (s:variable 'x) (s:number 3)) (s:variable 'x))))
```

Parsing datum into AST terms

- Parsing supports function definitions.
- Function `s:parse` can parse a sequence of terms, which corresponds to a Racket program.

```
(check-equal?
  (s:parse '[(define (f x) x) (f 1)])
  (s:define (s:variable 'f) (s:lambda (list (s:variable 'x)) (s:variable 'x))))
```

λ_D-calculus: λ-calculus with definitions

Semantics

$$\frac{e \Downarrow_E v}{e \Downarrow_E (E, v)} \quad \texttt{(E-exp)}$$

$$\frac{e \Downarrow_E v}{(\texttt{define}\ x\ e) \Downarrow_E (E[x \mapsto v], \texttt{void})} \quad \texttt{(E-def)}$$

$$\frac{t_1 \Downarrow_{E_1} (E_2, v_1) \qquad t_2 \Downarrow_{E_2} (E_3, v_2)}{t_1; t_2 \Downarrow_{E_1} (E_3, v_2)} \quad \texttt{(E-seq)}$$

- Evaluating a define *extends* the environment with a new binding
- Sequencing must thread the environments

Please, use your Rule Sheet in the following examples:

$$v \Downarrow_E v \qquad (\texttt{E-val})$$

$$x \Downarrow_E E(x) \qquad (\texttt{E-var})$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \qquad (\texttt{E-lam})$$

$$\frac{e_f \Downarrow_E (E_b, \lambda x.t_b) \qquad e_a \Downarrow_E v_a \qquad t_b \Downarrow_{\mathbf{E_b[x \mapsto v_a]}} v_b}{(e_f\ e_a) \Downarrow v_b} \qquad (\texttt{E-app})$$

$$\frac{e \Downarrow_E v}{e \Downarrow_E (E, v)} \qquad (\texttt{E-exp})$$

$$\frac{e \Downarrow_E v}{(\texttt{define}\ x\ e) \Downarrow_E (E[x \mapsto v], \texttt{void})} \qquad (\texttt{E-def})$$

$$\frac{t_1 \Downarrow_{E_1} (E_2, v_1) \qquad t_2 \Downarrow_{E_2} (E_3, v_2)}{t_1; t_2 \Downarrow_{E_1} (E_3, v_2)} \qquad (\texttt{E-seq})$$

Evaluating define

Example 1

- Consider the following program

```
(define a 20)
(define b (lambda (x) a))
(b 1)
```

- What is the output of this program?

Evaluating define

Example 1

- Consider the following program

```
(define a 20)
(define b (lambda (x) a))
(b 1)
```

- What is the output of this program? The output is: `20`

Let us try and evaluate this program with our λ_D semantics!

Example 1: step 1

Input

```
Environment: []
Term: (define a 20)
```

Example 1: step 1

Input

```
Environment: []
Term: (define a 20)
```

Output

```
Environment: [ (a . 20) ]
Value: #<void>
```

Evaluating

Example 1: step 1

Input

```
Environment: []
Term: (define a 20)
```

Output

```
Environment: [ (a . 20) ]
Value: #<void>
```

Evaluating

$$\frac{20 \Downarrow_{\{\}} 20 \quad \texttt{(E-val)}}{(\texttt{define}\ a\ 20) \Downarrow_{\{\}} (\{a : 20\}, \texttt{void})}\ \texttt{E-def}$$

Example 1: step 2

Input

```
Environment: [ (a . 20) ]
Term: (define b (lambda (y) a))
```

Example 1: step 2

Input

```
Environment: [ (a . 20) ]
Term: (define b (lambda (y) a))
```

Output

```
Environment: [
  (a . 20)
  (b . (closure [(a . 20)] (lambda (y) a)))
]
Expression: #<void>
```

Example 1: step 2

Input

```
Environment: [ (a . 20) ]
Term: (define b (lambda (y) a))
```

Output

```
Environment: [
  (a . 20)
  (b . (closure [(a . 20)] (lambda (y) a)))
]
Expression: #<void>
```

Evaluating

$$\dfrac{\lambda y.a \Downarrow_{\{a:20\}} (\{a:20\}, \lambda y.a) \quad \textbf{(E-lam)}}{(\texttt{define}\ b\ \lambda y.a) \Downarrow_{\{a:20\}} (\{a:20, b:(\{a:20\}, \lambda y.a)\}, \texttt{void})}\ \texttt{E-def}$$

Example 1: step 3

Input

```
Environment: [
  (a . 20)
  (b . (closure [(a . 20)] (lambda (y) a)))
]
Term: (b 1)
```

Example 1: step 3

Input

```
Environment: [
  (a . 20)
  (b . (closure [(a . 20)] (lambda (y) a)))
]
Term: (b 1)
```

Output

```
Environment: [
  (a . 20)
  (b . (closure [(a . 20)] (lambda (y) a)))
]
Expression: 20
```

Evaluation

Example 1: step 3

Input

```
Environment: [
  (a . 20)
  (b . (closure [(a . 20)] (lambda (y) a)))
]
Term: (b 1)
```

Output

```
Environment: [
  (a . 20)
  (b . (closure [(a . 20)] (lambda (y) a)))
]
Expression: 20
```

Evaluation

$$\dfrac{\dfrac{\dfrac{E(b) = (\{a:20\}, \lambda y.a)}{b \Downarrow_E (\{a:20\}, \lambda y.a)}\textbf{E-var} \quad \dfrac{}{1 \Downarrow_E 1}\textbf{E-val} \quad \dfrac{F(a) = 20}{a \Downarrow_F 20}\textbf{E-var}}{(b\ 1) \Downarrow_E 20}\textbf{E-app}}{(b\ 1) \Downarrow_E (E, 20)}\textbf{E-exp}$$

where

$$E = \{a:20, b:(\{a:20\}, \lambda y.a)\}$$
$$F = E[y \mapsto 1] = \{a:20, b:(\{a:20\}, \lambda y.a), y:1\}$$

Evaluating define

Example 2

Consider the following program

```
(define b (lambda (x) a))
(define a 20)
(b 1)
```

What is the output of this program?

Evaluating define

Example 2

- Consider the following program

```
(define b (lambda (x) a))
(define a 20)
(b 1)
```

- What is the output of this program? The output is: `20`

Let us try and evaluate this program with our λ_D semantics!

Example 2: step 1

Input

```
Environment: []
Term: (define b (lambda (y) a))
```

Example 2: step 1

Input

```
Environment: []
Term: (define b (lambda (y) a))
```

Output

```
Environment: [
  (b . (closure [] (lambda (y) a))
]
Expression: #<void>
```

Evaluation

Example 2: step 1

Input

```
Environment: []
Term: (define b (lambda (y) a))
```

Output

```
Environment: [
  (b . (closure [] (lambda (y) a))
]
Expression: #<void>
```

Evaluation

$$\dfrac{\lambda y.a \Downarrow_{\{\}} (\{\}, \lambda y.a) \quad \texttt{(E-lam)}}{(\texttt{define}\ b\ \lambda y.a) \Downarrow_{\{\}} (\{b : (\{\}, \lambda y.a)\}, \texttt{void})}\ \texttt{E-def}$$

Example 2: step 2

Input

```
Environment: [
  (b . (closure [] (lambda (y) a))
]
Term: (define a 20)
```

Example 2: step 2

Input

```
Environment: [
  (b . (closure [] (lambda (y) a))
]
Term: (define a 20)
```

Output

```
Environment: [
  (a . 20)
  (b . (closure [] (lambda (y) a))
]
Expression: #<void>
```

Evaluation

Example 2: step 2

Input

```
Environment: [
  (b . (closure [] (lambda (y) a))
]
Term: (define a 20)
```

Output

```
Environment: [
  (a . 20)
  (b . (closure [] (lambda (y) a))
]
Expression: #<void>
```

Evaluation

$$\dfrac{20 \Downarrow_{\{b:(\{\},\lambda y.a)\}} 20 \quad \texttt{(E-val)}}{(\texttt{define}\ a\ 20) \Downarrow_{\{b:(\{\},\lambda y.a)\}} (\{b : (\{\}, \lambda y.a), a : 20\}, \texttt{void})}\ \texttt{E-def}$$

Example 2: step 3

Input

```
Environment: [
  (a . 20)
  (b . (closure [] (lambda (y) a))
]
Term: (b 1)
```

Example 2: step 3

Input

```
Environment: [
  (a . 20)
  (b . (closure [] (lambda (y) a))
]
Term: (b 1)
```

Output

```
Environment: [
  (a . 20)
  (b . (closure [] (lambda (y) a))
]
Expression: error! a is undefined
```

Insight

> When creating a closure we copied the existing environment, and therefore any future updates are forgotten.

The semantics of λ_D is not enough! We need to introduce a notion of **mutation**.

How do implement mutation without mutable constructs?

Shared "mutable" state with immutable data-structures

Why, though?

Benefits

- A necessity if we use a language without mutation (such as Haskell)
- Parallelism: A great way to implement fast and safe data-structures in concurrent code (look up copy-on-write)
- Development: Controlled mutation improves code maintainability
- Memory management: counters the problem of circular references (notably, useful in C++ and Rust, see example)

> Encoding shared mutable state with immutable data-structures is a great skill to have.

Heap

> We want to design a data-structure that represents a *heap* (a shared memory buffer) that allows us to: *allocate* a new memory cell, *load* the contents of a memory cell, and *update* the contents of a memory cell.

Constructors

- `empty-heap` returns an empty heap
- `(heap-alloc h v)` creates a new memory cell in heap `h` whose contents are value `v`
- `(heap-put h r v)` updates the contents of memory handle `r` with value `v` in heap `h`

Selectors

- `(heap-get h r)` returns the contents of memory handle `r` in heap `h`

Heap usage

```
(define h empty-heap)           ; h is an empty heap
(define r (heap-alloc h "foo")) ; stores "foo" in a new memory cell
```

> What should the return value of `heap-alloc`?

- Should `heap-alloc` return a copy of `h` extended with "foo"? But then, how to we access the memory cell pointing to "foo"?
- Should `heap-alloc` return a handle to the new memory cell? But, since there is no mutation, how can we access the new heap?

Heap usage

```
(define h empty-heap)          ; h is an empty heap
(define r (heap-alloc h "foo")) ; stores "foo" in a new memory cell
```

What should the return value of `heap-alloc`?

- Should `heap-alloc` return a copy of `h` extended with "foo"? But then, how to we access the memory cell pointing to "foo"?
- Should `heap-alloc` return a handle to the new memory cell? But, since there is no mutation, how can we access the new heap?

Function `heap-alloc` must return a *pair* `eff` that contains the new heap and the memory handle.

```
(struct eff (state result) #:transparent)
```

Heap usage example

Spec

```
(define h1 empty-heap)           ; h is an empty heap
(define r (heap-alloc h1 "foo")) ; stores "foo" in a new memory cell
(define h2 (eff-state r))
(define x (eff-result r)) ;
(check-equal? "foo" (heap-get h2 x)) ; checks that "foo" is in x
(define h3 (heap-put h2 x "bar"))    ; stores "bar" in x
(check-equal? "bar" (heap-get h3 x)) ; checks that "bar" is in x
```

Handles must be unique

We want to ensure that the handles we create are **unique**, otherwise allocation could overwrite existing data, which is undesirable.

Spec

```
(define h1 empty-heap)           ; h is an empty heap
(define r1 (heap-alloc h1 "foo")) ; stores "foo" in a new memory cell
(define h2 (eff-state r1))
(define x (eff-result r1))
(define r2 (heap-alloc h2 "bar")) ; stores "foo" in a new memory cell
(define h3 (eff-state r2))
(define y (eff-result r2))
(check-not-equal? x y)  ; Ensures that x ≠ y
(check-equal? "foo" (heap-get h3 x))
(check-equal? "bar" (heap-get h3 y))
```

How can we implement a memory handle?

A simple heap implementation

- Let a handle be an integer
- Recall that the heap only grows (no deletions)
- A the handle matches the number of elements already present in the heap
- When the heap is empty, the first handle is 0, the second handle is 1, and son..

Heap: A solution

- We use a hash-table to represent the heap because it has a faster random-access than a linked-list (where lookup is linear on the size of the list).
- We wrap the hash-table in a struct, and the handle (which is a number) in a struct, for better error messages. And because it helps maintaining the code.

```
(struct heap (data) #:transparent)
(define empty-heap (heap (hash)))
(struct handle (id) #:transparent)
(struct eff (state result) #:transparent)
(define (heap-alloc h v)
  (define data (heap-data h))
  (define new-id (handle (hash-count data)))
  (define new-heap (heap (hash-set data new-id v)))
  (eff new-heap new-id))
(define (heap-get h k)
  (hash-ref (heap-data h) k))
(define (heap-put h k v)
  (define data (heap-data h))
  (cond
    [(hash-has-key? data k) (heap (hash-set data k v))]
    [else (error "Unknown handle!")]))
```

Contracts

Contracts

> Adding some sanity to highly dynamic code.

- Design-by-contract: idea pioneered by Bertrand Meyer and pushed in the programming language **Eiffel**, which was recognized by ACM with the Software System Award in 2006.
- Contracts are pre- and post-conditions each unit of code must satisfy (*e.g.*, a function)
- In some languages, notably F* and Dafny, pre- and post-conditions are checked at compile time!

Bibliography

Design by Contract, in Advances in Object-Oriented Software Engineering, eds. D. Mandrioli and B. Meyer, Prentice Hall, 1991.

Contracts in Racket

> Use `define/contract` rather than `define` to test the validity of each parameter and the return value.

- The → operator takes a predicate for each argument and one predicate for the return value For instance: `(-> symbol? real? string?)` declares that the first parameter is a symbol, the second parameter is numeric, and the return value is a string.

Example

```
(define/contract (f x y)
  ; Defines the contract
  (-> symbol? real? string?)
  (format "(~a, ~a)"))
```

Contracts examples

Read up on Racket's manual entry on: data-structure contracts

- `real?` for numbers
- `any/c` for any value
- `list?` for a list
- `listof number?` for a list that contains numbers
- `cons?` for a pair
- `(or/c integer? boolean?)` either an integer or a boolean
- `(and/c integer? even?)` an integer that is an even number
- `(cons/c number? string?)` a pair with a number and a string
- `(hash/c symbol? number?)` a hash-table where the keys are symbols and the keys are numbers