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...


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

<table>
<thead>
<tr>
<th>File 1</th>
<th>File 2</th>
<th>Lines Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (58%)</td>
</tr>
<tr>
<td></td>
<td>assignment_154296_export/submission</td>
<td>rkt (52%)</td>
</tr>
<tr>
<td></td>
<td>assignment_154296_export/submission</td>
<td>rkt (50%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (99%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (99%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (24%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (32%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (20%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (20%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (28%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (27%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (58%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (52%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (50%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (99%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (99%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (24%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (32%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (20%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (20%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (28%)</td>
</tr>
<tr>
<td>assignment_154296_export/submission</td>
<td>assignment_154296_export/submission</td>
<td>rkt (27%)</td>
</tr>
</tbody>
</table>

rkt (60%)     97
rkt (51%)     85
rkt (47%)     74
rkt (99%)     113
rkt (99%)     66
rkt (34%)     66
rkt (32%)     45
rkt (29%)     61
rkt (30%)     61
rkt (35%)     35
rkt (35%)     33
cheating sample

---

```
#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))
  )

;; 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 (car b))
      a))
  )
```

---
Today we will...

1. Study adding `define` to our language
2. Introduce mutation in an immutable setting
3. Introduce Racket's contracts
$\lambda_D$-calculus: $\lambda$-calculus with definitions

Syntax

$\begin{align*}
t ::=& e \mid t; t \mid (\text{define } x \ e) \\
e ::=& v \mid x \mid (e_1 \ e_2) \mid \lambda x.\ t \quad v ::=& n \mid (E, \lambda x.\ t) \mid \text{void}
\end{align*}$

- 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.

\[ v ::= n \mid (E, \lambda x.t) \mid \text{void} \]

Racket implementation

```racket
;; 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

```racket
(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.

\[ t ::= e \mid t; t \mid (\text{define } x \ e) \]

Racket implementation

\[(\text{define } (s:\text{term? } t) (\text{or } (s:\text{expression? } t) (s:\text{seq? } t) (s:\text{define? } t)))\]
\[(\text{struct } s:\text{seq } (\text{fst } \text{snd}) \text{#:transparent})\]
\[(\text{struct } s:\text{define } (\text{var } \text{body}) \text{#:transparent})\]

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.

```scheme
(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.

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

\[
\begin{align*}
\frac{e \Downarrow_E v}{e \Downarrow_E (E, v)} \quad (E-\text{exp}) \\
\frac{\text{define } x e \Downarrow_E (E[x \mapsto v], \text{void})}{E \Downarrow_E (E, v)} \quad (E-\text{def}) \\
\frac{t_1 \Downarrow_{E_1} (E_2, v_1) \quad t_2 \Downarrow_{E_2} (E_3, v_2)}{t_1; t_2 \Downarrow_{E_1} (E_3, v_2)} \quad (E-\text{seq})
\end{align*}
\]

- Evaluating a define extends the environment with a new binding
- Sequencing must thread the environments
Please, use your Rule Sheet in the following examples:

\[ v \Downarrow_E v \quad \text{(E-val)} \]

\[ x \Downarrow_E E(x) \quad \text{(E-var)} \]

\[ \lambda x.t \Downarrow_E (E, \lambda x.t) \quad \text{(E-lam)} \]

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

\[ e \Downarrow_E v \quad \text{(E-exp)} \]

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

\[
\frac{t_1 \Downarrow_{E_1} (E_2, v_1) \quad t_2 \Downarrow_{E_2} (E_3, v_2)}{t_1; t_2 \Downarrow_{E_1} (E_3, v_2)} \quad \text{(E-seq)}
\]
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

\[
\begin{align*}
\text{(define } a & \ 20) \\
\text{(define } b & \ (\lambda(x) \ a)) \\
(b \ 1)
\end{align*}
\]

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

Let us try and evaluate this program with our $\lambda_D$ 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

\[
\begin{align*}
20 & \Downarrow \emptyset 20 \quad (E\text{-val}) \\
\text{(define a 20)} & \Downarrow \emptyset (\{a : 20\}, \text{void}) \quad E\text{-def}
\end{align*}
\]
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

\[
\frac{\lambda y.a \Downarrow_{\{a:20\}} \{a : 20\}, \lambda y.a} {\text{(define } b \lambda y.a) \Downarrow_{\{a:20\}} \{a : 20, b : (\{a : 20\}, \lambda y.a)\}, \text{void}}
\]

E-def

CS450 _SHARED MUTABLE STATE AND IMMUTABILITY  LECTURE 13  Tiago Cogumbleiro
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
```
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

\[
E(b) = \{a : 20\}, \lambda y.a
\]
\[
E \var{b} \downarrow_E \{a : 20\}, \lambda y.a
\]

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 $\lambda_D$ 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

\[
\begin{align*}
\lambda y.a \downarrow {} & (\{\}, \lambda y.a) \quad \text{(E-lam)} \\
(\text{define } b \lambda y.a) \downarrow {} & (\{b : (\{\}, \lambda y.a)\}, \text{void}) \quad \text{E-def}
\end{align*}
\]
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

\[
\frac{20 \downarrow \{ b : \{ \}, \lambda y. a \}}{\text{(define a 20)} \downarrow \{ b : \{ \}, \lambda y. a \} \ (\{ b : \{ \}, \lambda y. a \}, a : 20\}, \text{void})}
\]
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 $\lambda_D$ 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 do 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

\[(\text{define } h \text{ empty-heap}) \quad \text{; } h \text{ is an empty heap} \]
\[(\text{define } r (\text{heap-alloc } h \text{ "foo"})) \quad \text{; } \text{stores "foo" in a new memory cell} \]

What should the return value of \text{heap-alloc}?

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

Function \text{heap-alloc} must return a \textit{pair eff} that contains the new heap and the memory handle.

\[(\text{struct eff (state result)} \,:\text{transparent})\]
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 \neq 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 so on..
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

- 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

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