CSC 520, Spring 2020

Principles of Programming Languages

Michelle Strout

The University of Arizona

Version: 1

March 4, 2020

New Evaluation Rules

$$\frac{x \in \operatorname{dom} \rho \qquad \rho(x) \in \operatorname{dom} \sigma}{\langle \mathsf{VAR}(x), \rho, \sigma \rangle \Downarrow \langle \sigma(\rho(x)), \sigma \rangle} \quad (\textsc{Var})$$

$$\frac{x \in \operatorname{dom} \rho \qquad \rho(x) = \ell \qquad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \mathsf{SET}(x, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma'\{\ell \mapsto v\} \rangle} \quad (\textsc{Assign})$$

Semantics of Lambda

THE UNIVERSITY OF ARIZONA
Computer Science

Key Issue: Values of free variables

Static scoping:

Where `lambda` occurs, "look outward" for ρ;

Capture that ρ for future reference.

$$\frac{}{\langle \mathrm{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle (\!| \mathrm{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho |\!), \sigma \rangle} \quad (\textsc{MkClosure})$$

Create closure in C implementation of `eval` by

```
case LAMBDAX:
    return mkClosure(e->u.lambdax, env);
```

Plan

THE UNIVERSITY OF ARIZONA.
Computer Science

- **Announcements**
  - HW5 is due ~~Friday~~ Saturday
  - Study guide
- **Last time**
  - Scheme Semantics
    - Stores
    - Lambdas evaluate to closures (will finish today)
    - Application (will finish today)
- **Today**
  - Scheme wrap up
  - ML intro

Function Application (mistake in notes now fixed)

- **Which "even" is referenced with f is called?**

```
(val even (lambda (x) (= 0 (mod x 2))))

(val f (lambda (y) (if (even y) 5 15)))

(val even 3)

(f 10)
```

$$\frac{x \in \operatorname{dom} \rho \qquad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \mathrm{VAL}(x, e), \rho, \sigma \rangle \to \langle \rho, \sigma'\{\rho(x) \mapsto v\} \rangle} \quad (\textsc{DefineOldGlobal})$$

$$\frac{x \notin \operatorname{dom} \rho \qquad \ell \notin \operatorname{dom} \sigma \qquad \langle e, \rho\{x \mapsto \ell\}, \sigma\{\ell \mapsto \text{unspecified}\} \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \mathrm{VAL}(x, e), \rho, \sigma \rangle \to \langle \rho\{x \mapsto \ell\}, \sigma'\{\ell \mapsto v\} \rangle} \quad (\textsc{DefineNewGlobal})$$

Applying Closures

THE UNIVERSITY OF ARIZONA
Computer Science

Captured environment for free variables

Arguments for bound variables (≡ formal parameters)

$$\frac{\begin{array}{c}\langle e, \rho, \sigma\rangle \Downarrow \langle (\!|\mathsf{LAMBDA}(\langle x_1, \ldots, x_n\rangle, e_c), \rho_c |\!), \sigma_0\rangle \\ \langle e_1, \rho, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ \ell_1, \ldots, \ell_n \notin \operatorname{dom} \sigma_n \\ \langle e_c, \rho_c\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\}\rangle \Downarrow \langle v, \sigma'\rangle\end{array}}{\langle \mathsf{APPLY}(e, e_1, \ldots, e_n), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad (\textsc{ApplyClosure})$$

```
nl = f.u.closure.lambda.formals;
return eval(f.u.closure.lambda.body,
            bindalloclist(nl, vl, f.u.closure.env));
```

Why not use rho_c to evaluate actuals?

- **Key idea: formal parameters are given new locations**

```
(val x 7)
(val f (lambda (x) (lambda (y) (+ x y))))
(val f3 (f 3))
(f3 x)
```

Locations in Closures

THE UNIVERSITY OF ARIZONA
Computer Science

Key is shared mutable state.

```
-> (val resettable-counter-from
     (lambda (n)
        (list2
          (lambda () (set n (+ n 1)))
          (lambda () (set n 0)))))
-> (val twenty  (resettable-counter-from 20))
-> ((car twenty))
21
-> ((car twenty))
22
-> ((cadr twenty))
0
-> ((car twenty))
1
```

Closure Optimizations

THE UNIVERSITY OF ARIZONA
Computer Science

- Only copy part of rho mappings for free variables in lambda expression
- Keep closures on the stack
- Share closures
- Eliminate closures (when functions don't escape)

uscheme and the Five Questions

Abstract syntax: imperative core, `let`, `lambda`

Values: S-expressions
(especially `cons` cells, function closures)

Environments:
A name stands for a mutable location holding value

Evaluation rules: `lambda` captures environment

Initial basis: yummy higher-order functions

Common Lisp, Scheme

- **Advantages**
  - High-level data structures
  - Automatic memory management (garbage collection)
  - Programs as data!
  - Hygenic macros for extending the language
  - Used in AI applications
- **Disadvantages**
  - Hard to talk about data
  - Hard to detect errors at compile time
- **All about the lambda**
  - Major win
  - Real implementation cost (heap allocation)

Common List, Scheme

THE UNIVERSITY OF ARIZONA
Computer Science

- **Advantages**
  - High-level data structures
  - Automatic memory management (garbage collection)
  - Programs as data!
  - Hygenic macros for extending the language
  - Used in AI applications
- **Disadvantages**
  - Hard to talk about data
  - Hard to detect errors at compile time
- **All about the lambda**
  - Major win
  - Real implementation cost (heap allocation)

Scheme as it really is:
Macros, Cond exp, Mutation, …

THE UNIVERSITY OF ARIZONA
Computer Science

- **Macros!**
  - A Scheme program is just another S-expression
  - Function define-syntax manipulates syntax at compile time
  - Macros are hygienic–name classes are impossible
  - Let, and, many others implemented as macros

Scheme as it really is:
Macros, Cond exp, Mutation, …

- **Real Scheme: Conditionals**

```
(cond (c1 e1)   ; if c1 then e1
      (c2 e2)   ; else if c2 then e2
       ...          ...
      (cn en)) ; else if cn then en

; Syntactic sugar---'if' is a macro:
(if e1 e2 e3) == (cond (e1 e2) (#t e3))
```

Scheme as it really is: Macros, Cond exp, Mutation, …

The University of Arizona
Computer Science

- **Real Scheme: Mutation**
  - Not only variables can be mutated
  - Mutate heap-allocated cons cell

```
(set-car! '(a b c) 'd)  ➔  (d b c)
```

ML Overview

- **Designed for programs, logic, symbolic data**
- **Theme: Precise ways to describe data**
- **ML = uScheme + pattern matching + exceptions + static types**
- **Three new ideas**
  - (1) Pattern matching is big and important. You might really like it.
  - (2) Exceptions are easy
  - (3) Static types get two to three weeks in their own right

uScheme -> ML Rosetta Stone

uScheme	SML
`(cons x xs)`	`x :: xs`
`'()`	`[]`
`'()`	`nil`
`(lambda (x) e)`	`fn x => e`
`(lambda (x y z) e)`	`fn (x, y, z) => e`
`|| &&`	`andalso orelse`
`(let* ([x e1]) e2)`	`let val x = e1 in e2 end`

```
(let* ([x1 e1]                 let val x1 = e1
       [x2 e2]                     val x2 = e2
       [x3 e3]) e)                 val x3 = e3
                               in  e
                               end
```

Example: The length function

THE UNIVERSITY OF ARIZONA.
Computer Science

- **Algebraic laws**

```
length []          = 0
length (x::xs)     = 1 + length xs
```

- **Code**

```
fun length []          = 0
|   length (x::xs)     = 1 + length xs
```

- **Notice**
  - No parentheses! (Yay!)
  - Function application by juxtaposition
  - Infix operators
  - Function application has higher precedence than any infix operator
  - Compiler checks all the cases