CSCI-344
Programming Language Concepts (Section 3)

Lecture 10

Continuations, μScheme Semantics

Instructor: Hossein Hojjat

September 21, 2016

Where we are

Done:

- Functions as first-class citizens
- Higher-order functions for Lists

This session:

- A Taste of Continuation-Passing Style (CPS)
  - Yet another advantage of Higher-order functions
- μScheme Operational Semantics

Continuations

- Consider the absolute value function

```
(define abs (x) (if (< x 0) (* -1 x) x) )
```

Continuations

- Consider the absolute value function

```
(define abs (x) (if (< x 0) (* -1 x) x) )
```

- To evaluate `abs` for a given value we need:
    1. Finish evaluating the condition subexpression `(< x 0)`
    2. Then evaluate the appropriate branch `(* -1 x)` or `x`

Continuations

- Consider the absolute value function

```
(define abs (x) (if (< x 0) (* -1 x) x) )
```

- To evaluate `abs` for a given value we need:
  1. Finish evaluating the condition subexpression `(< x 0)`
  2. Then evaluate the appropriate branch `(* -1 x)` or `x`
- **Continuation** of the subexpression `(< x 0)` is the rest of computation that will come after evaluation of subexpression

Continuations

- Consider the absolute value function

```
(define abs (x) (if (< x 0) (* -1 x) x) )
```

- To evaluate `abs` for a given value we need:
  1. Finish evaluating the condition subexpression `(< x 0)`
  2. Then evaluate the appropriate branch `(* -1 x)` or `x`
- **Continuation** of the subexpression `(< x 0)` is the rest of computation that will come after evaluation of subexpression

```
(define abs (x)
  ( (lambda (y) (if y (* -1 x) x))  (< x 0)))
```

- `(lambda (y) (if y (* -1 x) x))` is the continuation of `(< x 0)` in the function `abs`

Continuations

- At any point during evaluation, **continuation** is the part of the computation that still remains to be done
- A **continuation** is a function from the result of the subexpression to the final result of the whole computation

Continuations

- At any point during evaluation, **continuation** is the part of the computation that still remains to be done
- A **continuation** is a function from the result of the subexpression to the final result of the whole computation

Example

- Question: What is the continuation of `(+ 3 4)` in `(+ 1 (+ 2 (+ 3 4)))`?

Continuations

- At any point during evaluation, **continuation** is the part of the computation that still remains to be done
- A **continuation** is a function from the result of the subexpression to the final result of the whole computation

Example

- Question: What is the continuation of `(+ 3 4)` in `(+ 1 (+ 2 (+ 3 4)))`?
- Answer: `(lambda (x) (+ 1 (+ 2 x)))`

Continuation-Passing Style (CPS)

```
(define add2 (n) (+ n 2))
(define add5 (m) (+ 3 (add2 m)))
```

Continuation-Passing Style (CPS)

```
(define add2 (n) (+ n 2))
(define add5 (m) (+ 3 (add2 m)))
```

- Consider the continuation of the expression `(add2 m)`:

```
(define add2 (n) (+ n 2))
(define add5 (m) ((lambda (r) (+ 3 r)) (add2 m)))
```

Continuation-Passing Style (CPS)

```scheme
(define add2 (n) (+ n 2))
(define add5 (m) (+ 3 (add2 m)))
```

- Consider the continuation of the expression `(add2 m)`:

```scheme
(define add2 (n) (+ n 2))
(define add5 (m) ((lambda (r) (+ 3 r)) (add2 m)))
```

- We can pass continuation as an extra parameter to `add2`
- Delegates the responsibility of calling continuation to function itself

```scheme
(define add2 (n k) (k (+ n 2)))
(define add5 (m) (add2 m (lambda (r) (+ 3 r))))
```

Continuation-Passing Style (CPS)

```
(define add2 (n) (+ n 2))
(define add5 (m) (+ 3 (add2 m)))
```

- Consider the continuation of the expression `(add2 m)`:

```
(define add2 (n) (+ n 2))
(define add5 (m) ((lambda (r) (+ 3 r)) (add2 m)))
```

- We can pass continuation as an extra parameter to `add2`
- Delegates the responsibility of calling continuation to function itself

```
(define add2 (n k) (k (+ n 2)))
(define add5 (m) (add2 m (lambda (r) (+ 3 r))))
```

- Function `add2` is said to be written in **continuation-passing style**

Continuation-Passing Style (CPS)

Continuation-passing style

A style of programming where every function is explicitly passed its continuation (i.e., another function to which it should send its result)

Continuation-passing style

A style of programming where every function is explicitly passed its continuation (i.e., another function to which it should send its result)

Original `length` function:

```
(define length (xs)
 (if (null ? xs) 0
   (+ 1 (length (cdr xs)))))
```

CPS variant of `length` function:

```
(define length (xs k)
 (if (null? xs) (k 0)
   (length (cdr xs) (lambda (x) (k (+ 1 x))))))
```

Why Continuations?

- Continuation-passing style makes the control structure of the program explicit
- This can be very useful in a variety of applications
- For example, you can pass several continuations to a function

```
(define divide (a b success fail)
  (if (= b 0)
    (fail 0)
    (success (/ a b)))
)
```

- CPS has similarities to the "goto" statement in some imperative languages
- We can use continuations to implement backtracking search
    - For example, see the SAT solver example in book

μScheme Operational Semantics

What's New

Key changes in uScheme compared to Impcore:

- New constructs: **`let`**, **`lambda`** and application
- New values: cons-cells and functions (closures)
- Impcore uses three environment to bind:
    1. functions,
    2. global variables,
    3. local variables

  μScheme uses only a single environment ρ
- Environments of μScheme get copied, a binding in an environment never gets mutated
    - Environment maps names to mutable locations, not values

Locations

- Why can't we use values instead of locations in an environment?

Locations

- Why can't we use values instead of locations in an environment?

```
(define f (x)
    (lambda () (set x (+ x 1))))
(val g (f 5))
```

- Evaluation of `(f 5)` creates a closure
  - Closure initially: $\langle\!|$ `(lambda () (set x (+ x 1)))`, $\{$`x` $\mapsto 5\}$ $|\!\rangle$

Locations

- Why can't we use values instead of locations in an environment?

```
(define f (x)
     (lambda () (set x (+ x 1))))
(val g (f 5))
```

- Evaluation of `(f 5)` creates a closure
  - Closure initially: $\langle\!|$ `(lambda () (set x (+ x 1)))`, $\{$`x` $\mapsto 5\}$ $|\!\rangle$
- We need to change the environment in closure after each call of `g`

```
-> (g)
6
-> (g)
7
```

- Environment points to the location ℓ of `x` instead of its value

$\langle\!|$ `(lambda () (set x (+ x 1)))`, $\{$`x` $\mapsto \ell\ \}$ $|\!\rangle$

New Judgment

New Constructs (Abstract Syntax)

$$
\begin{array}{lcll}
\mathsf{Exp} & = & \mathrm{LET} & (\mathsf{Namelist}, \mathsf{Explist}, \mathsf{Exp}) \\
 & | & \mathrm{LAMBDA} & (\mathsf{Namelist}, \mathsf{Exp}) \\
 & | & \mathrm{APPLY} & (\mathsf{Exp}, \mathsf{Explist})
\end{array}
$$

New Judgment

New Constructs (Abstract Syntax)

$$\begin{array}{rcll} \mathsf{Exp} & = & \mathrm{LET} & (\mathsf{Namelist}, \mathsf{Explist}, \mathsf{Exp}) \\ & | & \mathrm{LAMBDA} & (\mathsf{Namelist}, \mathsf{Exp}) \\ & | & \mathrm{APPLY} & (\mathsf{Exp}, \mathsf{Explist}) \end{array}$$

New Evaluation Judgment

$$\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$$

- ρ never changes
- ρ maps a name to a mutable location
- σ is the store (content of every location)

New Judgment

New Constructs (Abstract Syntax)

$$\begin{array}{lcll} \mathsf{Exp} & = & \mathrm{LET} & (\mathsf{Namelist}, \mathsf{Explist}, \mathsf{Exp}) \\ & | & \mathrm{LAMBDA} & (\mathsf{Namelist}, \mathsf{Exp}) \\ & | & \mathrm{APPLY} & (\mathsf{Exp}, \mathsf{Explist}) \end{array}$$

New Evaluation Judgment

$$\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$$

- ρ never changes
- ρ maps a name to a mutable location
- σ is the store (content of every location)

Some intuitions for a compiler (interpreter) writer:

- ρ models the compiler's symbol table
- σ models the contents of registers and memory

μScheme Semantics: Variables

- Looking up a variable doesn't change the store

$$\frac{x \in \mathit{dom}(\rho) \qquad \rho(x) \in \mathit{dom}(\sigma)}{\langle \mathrm{VAR}(x), \rho, \sigma \rangle \Downarrow \langle \sigma(\rho(x)), \sigma \rangle} \text{ (Var)}$$

μScheme Semantics: Assignment

$$\frac{x \in \mathit{dom}(\rho) \qquad \rho(x) = \ell \qquad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \mathrm{SET}(x, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma'\{\ell \mapsto v\} \rangle} \text{ (Assign)}$$

μScheme Semantics: Assignment

$$\frac{x \in \textit{dom}(\rho) \qquad \rho(x) = \ell \qquad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \mathrm{SET}(x, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma'\{\ell \mapsto v\} \rangle} \text{ (Assign)}$$

- What changes are captured in σ'?
- What changes are captured in $\sigma'\{\ell \mapsto v\}$?
- What would happen if we used σ instead of σ' in the conclusion?
- What would happen if we used a fresh ℓ?
- Some other ℓ in the range of ρ?

μScheme Semantics: Lambda Expression

- Wrap the current environment along with the lambda expression in a closure

$$\frac{x_1, \cdots, x_n \text{ all distinct}}{\langle \mathrm{LAMBDA}(\langle x_1, \cdots, x_n\rangle, e), \rho, \sigma\rangle \Downarrow \langle (\!|\mathrm{LAMBDA}(\langle x_1, \cdots, x_n\rangle, e), \rho|\!), \sigma\rangle} \text{ (MkClosure)}$$

μScheme Semantics: Function Application

$$
\frac{\begin{array}{c}
\langle e, \rho, \sigma \rangle \Downarrow \langle (\!|\mathrm{LAMBDA}(\langle x_1, \cdots, x_n \rangle, e_c), \rho_c|\!), \sigma_0 \rangle \\
\langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\
\vdots \\
\langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\
\ell_1, \cdots, \ell_n \notin \mathit{dom}(\sigma_n) \qquad \text{(and all distinct)} \\
\langle e_c, \rho_c\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle
\end{array}}{\langle \mathrm{APPLY}(e, e_1, e_2, \ldots, e_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \text{ (ApplyClosure)}
$$

μScheme Semantics: Function Application

$$\frac{\begin{array}{c}\langle e, \rho, \sigma\rangle \Downarrow \langle (\!|\mathrm{LAMBDA}(\langle x_1, \cdots, x_n\rangle, e_c), \rho_c|\!), \sigma_0\rangle \\ \langle e_1, \rho, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ \ell_1, \cdots, \ell_n \notin \mathit{dom}(\sigma_n) \qquad \text{(and all distinct)} \\ \langle e_c, \rho_c\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\}\rangle \Downarrow \langle v, \sigma'\rangle\end{array}}{\langle \mathrm{APPLY}(e, e_1, e_2, \ldots, e_n), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \text{ (ApplyClosure)}$$

- What if we used σ instead of σ_0 in evaluation of e_1?
- What if we used σ instead of σ_0 in evaluation of arguments?
- What if we used ρ_c instead of ρ in evaluation of arguments?
- What if we did not require $\ell_1, \cdots, \ell_n \notin \mathit{dom}(\sigma)$?
- What is the relationship between ρ and σ?

μScheme Semantics: Let Expression

$$\frac{\begin{array}{c} x_1,\ \cdots,x_n \text{ all distinct} \\ \sigma_0 = \sigma \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \ell_1, \cdots, \ell_n \notin \mathit{dom}(\sigma_n) \qquad \text{(and all distinct)} \\ \langle e, \rho\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\} \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \mathrm{LET}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \text{ (ApplyClosure)}$$

Summary

- Scheme: Lambda expressions, Closures, Currying, Algebraic Laws, Higher-order functions, Continuations

Next Lecture

- New Language, New Concepts!
- "Programming in Standard ML" (Robert Harper) (Chapters 1 - 13)
- PL:BPC Chapter 5