(More) Programming Language Fundamentals

September 27th, 2011
Melanie Kambadur

Adv. Topics in Programming Languages and Compilers,
Prof. Aho
My Background

• Grew up in Bloomington, IN
• B.S. in CS @ IU, 2007-2010
• Started PhD @ CU, Fall 2010
• Advisor Martha Kim
• Computer Arch. Lab
• Interested in: performance, compilers, esp. for parallel & heterogeneous archs
Indiana Programming Languages

- Four dedicated faculty members
- Several others with PL ties
- ~ 16 students (mostly PhDs)
- Weekly Seminars
- Usually around 4 classes per semester, e.g.
  - PL Principles (ugrad/grad)
  - PL Foundations
  - Compilers (ugrad/grad)
  - Domain-Specific Languages and Compilers: Performance meets Productivity
  - Reversible and Quantum Computing
  - Language-Based Approaches to Security
  + Intro. Class for ugrads is taught in Scheme
Today’s Topics

• Review: Types, Evaluation

• Lambda Calculus

• Brief Introduction to Scheme

• Recursion (non-tail vs. tail)

• Scoping and Closures

• Continuations and Continuation Passing Style
Type Systems

WHAT  “A type is metadata about a chunk of memory that classifies the kind of data stored there. This classification usually implicitly specifies what kinds of operations may be performed on the data.”  [1]

WHY  abstract, document, optimize, safety  [2]

HOW  primitives (num, char), containers (array, hash table), user-defined (class)  [1]
Dynamic vs. Static

• Dynamic Typing
  – Most type checks performed at run time

• Static Typing
  – Most type checks performed at compile time
Dynamic vs. Static

- **Dynamic Typing**
  - Most type checks performed at run time
  - Ex. Python, JavaScript, Matlab, Scheme

- **Static Typing**
  - Most type checks performed at compile time
  - Ex. C, Java, Pascal, Ocaml, Haskell
Dynamic vs. Static

• Dynamic Typing
  – Most type checks performed at run time
  – Ex. Python, JavaScript, Matlab, Scheme
  – Advantages: faster compilation, generic code
• Static Typing
  – Most type checks performed at compile time
  – Ex. C, Java, Pascal, Ocaml, Haskell
  – Advantages: errors, optimizable
• Disagreement over which has fastest develop.+debug cycle.
Weak vs. Strong

• Weak Typing
  – Implicit type conversion is allowed

• Strong Typing
  – Operation legality is based on type
Weak vs. Strong

• **Weak Typing**
  – Implicit type conversion is allowed
  – Ex. C, C++, JavaScript

• **Strong Typing**
  – Operation legality is based on type
  – Ex. Scheme, Java, Python, Haskell
Weak vs. Strong

- **Weak Typing**
  - Implicit type conversion is allowed
  - Ex. C, C++, JavaScript
  - Advantages: easier to write programs?

- **Strong Typing**
  - Operation legality is based on type
  - Ex. Scheme, Java, Python, Haskell
  - Advantages: safer? (behavior guaranteed)
Evaluation Order

WHAT  Rules for the evaluation of expressions.

WHY   Consistency in results of running code. Most languages use strict evaluation (not Haskell!)\(^3\)

HOW   Different strategies, and sometimes languages use more than one...
Evaluation Order

- **applicative eval.**: arguments of a function are evaluated from left to right; terms are reduced as much as possible before applying a function (strict)
- **normal order eval.**: functions are applied before arguments are evaluated (non-strict)
- **eager eval.**: an expression is evaluated as soon as possible (strict)
- **lazy eval.**: an expression’s evaluation is delayed until its value is needed so that sometimes repeated evaluations are avoided (non-strict)
Evaluation Order

• **call by value**: evaluates arguments to a procedure before applying the procedure and applies the procedure to the values of these arguments (strict)
  – most common eval strategy, e.g. C, Scheme, Java
• **call by reference**: a function receives an implicit reference to the variable used as argument, rather than a copy of its value. (strict)
  – e.g. ampersand in C
• **call by name**: arguments are not evaluated, just substituted into a function, and are evaluated when reached in the function. (non-strict)
• **call by need**: if an argument is evaluated once, we store it for future use. (non-strict, lazy evaluation is one type of call-by need implementation)
Lambda Calculus\textsuperscript{[4]}

WHAT
Introduced in 1930s by Church as a mathematical system for defining computable functions.

WHY
“A formal system for function definition, function application, and recursion.”\textsuperscript{[5]}
A model for functional programming languages.

HOW
Simple grammar, operations
λ Calc Grammar

Expression → Abstraction | Application | (expr) | var | constant
Abstraction → λ var . expr
Application → expr expr
Expression

Expr $\rightarrow$ abstr. $|$ appl. $|$ $(expr)$ $|$ var $|$ const.

- A program which when evaluated returns a result consisting of another lambda-calculus expression.
Abstraction

\[ \text{Abstr} \to \lambda \text{var} \cdot \text{Expr} \]

- An expression defining a function
- \text{var} is the formal parameter and \text{expr} the body
- \( \lambda \text{var} \cdot \text{Expr} \) binds \text{var} in \text{expr}. 
Application

Appl \rightarrow expr \ expr

• An expression followed by an expression: If \( e \) is a function and \( f \) an expression, then \( ef \) is a function application.
• in \((\lambda x.y)z\), we are applying the function \( \lambda x.y \) to the argument \( z \).
• Function application is left associative and application binds tighter than period
• \((\lambda x. \lambda y. xy) \lambda z.z = (\lambda x. (\lambda y. xy)) \lambda z.z\)
Free and Bound Variables

- All vars are local to function defs
- In the function $\lambda x.x$ the variable $x$ is **bound**
- In $(\lambda x.x)y$, $x$ is bound and $y$ is **free**
- In $(\lambda x.x y)(\lambda y.y)$, is $y$ free or bound?
- An expression with no free vars is **closed**
\( \lambda \) Calc Reductions

- \( \alpha \)-reduction: consistent renaming of variables
  - \( \lambda x.x \equiv \lambda y.y \equiv \lambda z.z \)
  - Can’t rename free vars or rename bound vars to names taken by free vars

Definitions from [6]
\( \lambda \) Calc Reductions

• \( \alpha \)-reduction: consistent renaming of variables
  – \( \lambda x.x \equiv \lambda y.y \equiv \lambda z.z \)
  – Can’t rename free vars or rename bound vars to names taken by free vars

• \( B \)-reduction: application of functions
  – \([y/x]e\) means \( y \) is subst. for all \( x \)’s in \( e \).
  – Ex. \((\lambda x.x)y \rightarrow [y/x]x = y\) (identity function)

Definitions from [6]
More $\lambda$ Calc

• $\eta$-conversion/ lambda abstraction: convert expressions to functions, or funcs to exprs
  – Reduction: $(\lambda x. Mx) \rightarrow_\eta M$
  – Abstraction: $(\lambda x. Mx) \leftarrow_\eta M$
More $\lambda$ Calc

- $\eta$-conversion/ lambda abstraction: convert expressions to functions, or funcs to exprs
  - Reduction: $(\lambda x. Mx) \rightarrow^\eta M$
  - Abstraction: $(\lambda x. Mx) \leftarrow^\eta M$

- Normal Form: An expr with no more possible beta reductions (not containing func. appl.)
  - Ex: $x, xe, \lambda x.e$ (where $e$ is in normal form)
Y Combinator

• One of many fixed point combinators.
• Recursion without special language support
Y Combinator

• One of many fixed point combinators.
• Recursion without special language support

\[ Y = \lambda f. (\lambda x.f (x x)) (\lambda x.f (x x)) \]

\[ Y \, g = (\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))) \, g \text{ (by definition of } Y) \]

\[ = (\lambda x . g (x x)) (\lambda x . g (x x)) \text{ (} \beta \text{-reduction of } \lambda f \text{: applied main function to } g) \]

\[ = (\lambda y . g (y y)) (\lambda x . g (x x)) \text{ (} \alpha \text{-conversion: renamed bound variable) \]

\[ = g ((\lambda x . g (x x)) (\lambda x . g (x x))) \text{ (} \beta \text{-reduction of } \lambda y \text{: applied left function to right function) \]

\[ = g (Y \, g) \text{ (by third equality)} \]

Example from [7]
Today’s Topics (10/4)

• Review: Brief Introduction to Scheme

• Recursion (non-tail vs. tail)

• Scoping and Closures

• Continuations and Continuation Passing Style
The Scheme Programming Language

WHAT  Functional language,
Descendant of LISP (yes, lots of parens)

WHY   Easy to think algorithmically,
Natural recursion,
Extensible through macros

HOW    examples to follow,
scheme.tar
Sidebar: side effects

- When an expression modifies state (outside of its own function).
- Examples:
  - Update a global variable
  - Print to I/O
  - Modify an input argument
- Sometimes line between imperative and functional languages
- Another distinguishing feature is support of first-class functions
### Scheme Implementation Details

- Governed by standards, up to R6RS (see [8])

<table>
<thead>
<tr>
<th>Introduced</th>
<th>1975</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Original Designers</strong></td>
<td>Guy L. Steele &amp; Gerald Jay Sussman</td>
</tr>
<tr>
<td><strong>Scope</strong></td>
<td>Lexical</td>
</tr>
<tr>
<td><strong>Typing</strong></td>
<td>Dynamic, Strong</td>
</tr>
<tr>
<td><strong>First-class functions?</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Memory Management</strong></td>
<td>Garbage Collection</td>
</tr>
</tbody>
</table>
"We were actually trying to build something complicated and discovered, serendipitously, that we had accidentally designed something that met all our goals but was much simpler than we had intended....we realized that the lambda calculus—a small, simple formalism—could serve as the core of a powerful and expressive programming language.”

—Sussman & Steele, *The First Report on Scheme Revisited*
Design Quotes

“Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.” –R6RS Report [8]
Play Along?

• Download: http://www.scheme.com/download/
  – Scroll down to: *Petite Chez Scheme*
  – Build is simple, .pkg/.exe is simpler

• In terminal, type `petite` to get to REPL

• For reference, see: http://www.scheme.com/tspl4/
Scheme Language Basics

**Constants**
- Booleans: \#t \#f
- Numbers: 1
- Data: `a
  `(1 2 3)
- String: “abc”

**Procedure Calls**
- (* 3 4)
- (map sqrt '(16 9))

**Lambda Expressions**
- (lambda (x) x)
- (lambda (x y) (+ x y))

**Assignments**
- (define a 2)
- (define identity (lambda (x) x))
More Language Basics

Conditionals
(if (> 3 2)
  'yes
  'no)
(cond ((> 3 3) 'gt)
  ((< 3 3) 'lt)
  (else 'eq))

Binding Constructs
(let ((x 2) (y 3))
  (* x y))

Predicates
(pair? '(a b c))
(even? 2)
(zero? 2)
\( \lambda \text{ Calc in Scheme} \)

Expression \( \rightarrow \) Abstraction \( | \) Application \( | \) `\langle \text{var} \rangle` \( | \) \langle constant\rangle (e.g. \(1, \#t, "abc"\))

Abstraction \( \rightarrow \) (\(\lambda\) (\(\langle \text{var} \rangle\) \(\langle \text{body} \rangle\))) \( | \) (\(\lambda\) (\(\langle \text{formals} \rangle\)) \(\langle \text{body} \rangle\))

Application \( \rightarrow \) (operator operand operand)
Currying

“The process of incrementally supplying arguments to a function.”[^9]

Multiple argument operation $\rightarrow$ single argument operation
Currying

“The process of incrementally supplying arguments to a function.”[9]

Multiple argument operation → single argument operation

> (* 2 4 )

8
Currying

“The process of incrementally supplying arguments to a function.”[⁹]

Multiple argument operation → single argument operation

> (* 2 4)
8
>(define (mult y)
   (lambda (x)
     (* y x)))
> ((mult 2) 4)
8
More λ Calc in Scheme

• Y Combinator:

```scheme
(define Y
  (lambda (f)
    (let ((g (lambda (h)
                     (lambda (x) ((f (h h)) x))))
         (g g)))
  See [10] to find out one way to derive
Free/Bound → Scope

- Lexical (Static): identifier bindings are local to functions
- Dynamic: identifiers have bindings on a global stack
Free/Bound $\rightarrow$ Scope

- **Lexical (Static):** identifier bindings are local to functions
- **Dynamic:** identifiers have bindings on a global stack
- **Example:** `scope.ss^{[11]}
  - $(n = \,?)$ $(n = \,?) \rightarrow$ lexical
  - $(n = \,?)$ $(n = \,?) \rightarrow$ dynamic
Free/Bound $\rightarrow$ Scope

• Lexical (Static): identifier bindings are local to functions

• Dynamic: identifiers have bindings on a global stack

• Example: `scope.ss^{[11]}
  - (n = 2) \quad (n = 2) \rightarrow$ lexical
  - (n = ?) \quad (n = ?) \rightarrow$ dynamic
Free/Bound → Scope

• Lexical (Static): identifier bindings are local to functions

• Dynamic: identifiers have bindings on a global stack

• Example: scope.ss\textsuperscript{[11]}
  
  – (n = 2) (n = 2) → lexical
  
  – (n = 7) (n = 2) → dynamic
Like LISP

• S-expressions (list data structures)

• List operations:
  – car → returns first element of the list
  – cdr → returns list minus the first element
  – cons → combines lists

• Example in REPL...
Simple Scheme Functions

Examples:

increment.ss

fact.ss
Recursion

WHAT Applying a function within itself

WHY Finite representation of infinite set, Divide and conquer

HOW Base case, reducing rules, examples: fib-iter.c, fib-rec.c
Tail Recursion

• If a procedure is going to return the same value it was called with, why call the procedure in the first place?
  – Tail calls: “when one procedure directly returns the result of invoking another procedure”[15]
  – Tail recursion: “when a procedure recursively tail-calls itself, directly or indirectly”[15]

• In Scheme implementations:
  – Tail calls must be GOTOs (tail call optimization)
  – tail recursion → indefinite iteration with no stack overflow
  – Example: fact-tail.ss
Non-Tail vs. Tail Stacks

fact (3)
  fact (3 1)
  fact (2 3)
  fact (1 6)
  fact (0 6)
  ret 6
  ret 6
  ret 6
  ret 6
  ret 6
ret 6
Non-Tail vs. **Tail** Stacks

```
fact (3)  
  fact (3 1)  
    fact (2 3)  
      fact (1 6)  
        fact (0 6)  
          ret 6  
          ret 6  
          ret 6  
          ret 6  
          ret 6  
          ret 6
```
Non-Tail vs. Tail Stacks

```
fact (3)
fact (3 1)
fact (2 3)
fact (1 6)
fact (0 6)
ret 6
ret 6
ret 6
ret 6
```

```
fact (3)
save(3 1), goto fact
save(2 3), goto fact
save(1 6), goto fact
save(0 6), goto fact
ret 6

Compiler saves space:
Only calling function’s addr is kept, nothing else on stack or heap.
```
Not Like LISP

• Data and functions share namespace:
  – No defun, setf and #’

• Order of evaluation not defined
  – Why? Legal to have expr in operator position as long as it evaluates to operator
  – You can force order (e.g. begin)
Not Like LISP

• Data and functions share namespace:
  – No defun, setf and #’

• Order of evaluation not defined
  – Why? Legal to have expr in operator position as long as it evaluates to operator
  – You can force order (e.g. begin)

★ Lexical Scope to properly represent λ calc
Advantage of Lexical Scope

• Behavior of variables is more predictable
  – Don’t have to anticipate all contexts

• Not possible for free var in procedure to refer to external bindings

• Any dynamic scope left in the real world?
Closures

WHAT  A procedure and its environment, (envr. has bindings for non-local vars)
Introduced by Sussman & Steele in [13]

WHY   A way to implement lexical scope with nested first class functions

HOW   When function runs, free vars look up closure environment
Continuations\textsuperscript{[14][15]}

WHAT
A semantic stack of what remains to be executed.
DS with stack, reg content, pc

WHY
Save execution state and return to it later (reification).
Use for threading, backtracking, co-routines.

HOW
Grab from machine (in Scheme, \texttt{call/cc}), or
Always pass along continuation
Continuation Example\textsuperscript{[15]}

Keep track of:

1) what to evaluate, and
2) what to do with the value $\rightarrow$ continuation

6 continuations waiting for:

$(\text{if} \ (\text{null?} \ x) \ (\text{quote} \ ()) \ (\text{cdr} \ x))$
Continuation Example\textsuperscript{[15]}

Keep track of:

1) what to evaluate, and
2) what to do with the value $\rightarrow$ continuation

6 continuations waiting for: the value of

$$(\text{if (null? x) (quote ()) (cdr x))}$$
Continuation Example$^{[15]}$

Keep track of:

1) what to evaluate, and
2) what to do with the value $\rightarrow$ continuation

6 continuations waiting for: the value of

$$(\text{if } (\text{null? } x) \ (\text{quote } ()) \ (\text{cdr } x))$$
Continuation Example\textsuperscript{[15]}

Keep track of :

1) what to evaluate, and
2) what to do with the value $\rightarrow$ continuation

6 continuations waiting for: the value of

$$(\text{if } (\text{null? } x) (\text{quote } ()) (\text{cdr } x))$$
Continuation Example\textsuperscript{[15]}

Keep track of:

1) what to evaluate, and
2) what to do with the value $\rightarrow$ continuation

6 continuations waiting for: the value of

\[(\text{if (null? x) (quote ()) (cdr x))}\]
Continuation Example\[^{15}\]

Keep track of:

1) what to evaluate, and
2) what to do with the value $\rightarrow$ continuation

6 continuations waiting for: the value of

$\text{(if (null? x) (quote ()) (cdr x))}$
Continuation Example\textsuperscript{[15]}

Keep track of:

1) what to evaluate, and
2) what to do with the value $\rightarrow$ continuation

6 continuations waiting for: the value of

$(\text{if (null? x) (quote ()) (cdr x))}$
Continuation Example

Keep track of:

1) what to evaluate, and
2) what to do with the value \(\rightarrow\) continuation

6 continuations waiting for:

\[
\text{(if (null? x) (quote ()) (cdr x))}
\]

Doesn’t count because it’s the same as the first
call/cc$^{[15]}$ $^{[16]}$

- Get the continuation of any expression

- Like setjmp() in C

- Many languages offer limited continuation support (breaks, returns)
call/cc examples\textsuperscript{[15]}

(call/cc
  (lambda (k)
    (* 5 4)))

(call/cc
  (lambda (k)
    (* 5 (k 4)))))

(+ 2
  (call/cc
    (lambda (k)
      (* 5 (k 4))))))
call/cc examples\textsuperscript{[15]}

(call/cc
  (lambda (k)
    (* 5 4))) \rightarrow 20

(call/cc
  (lambda (k)
    (* 5 (k 4))))

(+ 2
  (call/cc
    (lambda (k)
      (* 5 (k 4))))))

Continuation is obtained and bound to \(k\), but \(k\) is never used, so the value is simply the product of 5 and 4.
call/cc examples\textsuperscript{[15]}

(call/cc
  (lambda (k)
    (* 5 4)))

(call/cc
  (lambda (k)
    (* 5 (k 4)))))

(+ 2
  (call/cc
    (lambda (k)
      (* 5 (k 4)))))
call/cc examples

(call/cc
  (lambda (k)
    (* 5 4)))

(call/cc
  (lambda (k)
    (* 5 (k 4)))) \rightarrow 4

(+ 2
  (call/cc
    (lambda (k)
      (* 5 (k 4)))))

Continuation is invoked before the multiplication, so the value is the value passed to the continuation, 4.
call/cc examples$^{[15]}$

(call/cc
 (lambda (k)
  (* 5 4)))

(call/cc
 (lambda (k)
  (* 5 (k 4)))))

(+ 2
 (call/cc
  (lambda (k)
   (* 5 (k 4))))))
call/cc  examples\textsuperscript{[15]}

(call/cc
 (lambda (k)
  (* 5 4)))
(call/cc
 (lambda (k)
  (* 5 (k 4))))
(+ 2
 (call/cc
  (lambda (k)
   (* 5 (k 4))))) \rightarrow 6

Continuation includes the addition by 2; thus, the value is the value passed to the continuation, 4, plus 2.
## Continuation Passing Style (CPS)\(^{[15]}\)

<table>
<thead>
<tr>
<th><strong>WHAT</strong></th>
<th>Replacing continuations with explicit procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Introduced in the <em>Lambda Papers</em></td>
</tr>
</tbody>
</table>

| **WHY** | Pass more than one result to its continuation; |
|         | Take separate "success" and "failure" continuations; |
|         | Any program that uses call/cc can be rewritten in CPS without call/cc. |
Continuation Passing Style (CPS)\textsuperscript{[15,17]}

HOW

Represent continuations as functions that take one parameter, $v$ (the result)

All recursive functions take a continuation, $k$ that is applied when a result is ready

Example: $\text{fact-cps.ss}$
How to X-form to CPS\textsuperscript{[15]}

• When one procedure calls another with a non-tail call, the callee gets an implicit continuation that completes what’s left of the caller’s body and returns to the callee’s continuation

• If the call is a tail call, the callee simply receives the continuation of the caller

• You try: fib.ss $\rightarrow$ fib-cps.ss
Resources

- Free Petite Chez Scheme Download: [http://www.scheme.com/download/](http://www.scheme.com/download/)
- The original *Lambda Papers* by Guy Steele and Gerald Sussman: [http://library.readscheme.org/page1.html](http://library.readscheme.org/page1.html)
- *Essentials of Programming Languages, The Reasoned Schemer, Little Schemer* series, by Dan Friedman and others
References

4. “Notes on Lambda Calculus”, Al Aho, COMS E6998-2 Advanced Topics in Programming Languages and Compilers
8. R6RS, Sperber et al., http://www.r6rs.org/index.html