Functional Languages

CSE 307 – Principles of Programming Languages

Stony Brook University

http://www.cs.stonybrook.edu/~cse307

Historical Origins

- The imperative and functional models grew out of work undertaken Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, etc. ~1930s
  - different formalizations of the notion of an algorithm, or effective procedure, based on automata, symbolic manipulation, recursive function definitions, and combinatorics
- These results led Church to conjecture that any intuitively appealing model of computing would be equally powerful as well
  - this conjecture is known as Church's thesis

Historical Origins

- Turing's model of computing was the Turing machine a sort of pushdown automaton using an unbounded storage "tape"
  - the Turing machine computes in an imperative way, by changing the values in cells of its tape – like variables just as a high level imperative program computes by changing the values of variables

Historical Origins

- Church's model of computing is called the lambda calculus
  - based on the notion of parameterized expressions with each parameter introduced by an occurrence of the letter λ.
  - Lambda calculus was the inspiration for functional programming.
  - Computation by substitution of parameters into expressions, just as computation by passing arguments to functions.
  - Constructive proof that transforms input into output.

Lambda Calculus

- λ = lambda
- lambda terms consist of:
  - variables (a)
  - lambda abstraction (λa.t)
  - application (t s)
- Variables can be bound by lambda abstractions or free:
  - Example: in λa.ab, a is bound, b is free.

Lambda Calculus

- alpha equivalence: $\lambda a.a = \lambda b.b$
- beta substitution: $(\lambda a.aa)\ b = bb$
  - problem: what happens if we substitute a free variable into a place where it would be bound?
  - Example: (ya.(yb.ab)) b c
    - wrong: (yb.bb) c

      cc
    - right: use alpha equivalence to ensure this doesn't happen.

      (ya.(yd.ad)) b c

      (yd.bd) c

      bc

Functional Programming Concepts

- Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are an attempt to realize Church's lambda calculus in practical form as a programming language
- The key idea: do everything by composing functions
  - no mutable state
  - no side effects
- So how do you get anything done in a functional language?
  - Recursion takes the place of iteration
  - First-call functions take value inputs
  - Higher-order functions take a function as input

Functional Programming Concepts

- Necessary features, many of which are missing in some imperative languages:
  - high-order functions
  - powerful list facilities
  - structured function returns
  - fully general aggregates
  - garbage collection

Functional Programming

- LISP family of programming languages:
  - Pure (original) Lisp
  - Interlisp, MacLisp, Emacs Lisp
  - Common Lisp
  - Scheme
    - All of them use s-expression syntax: (+ 1 2).
- LISP is old - dates back to 1958 - only Fortran is older.
- Anything in parentheses is a function call (unless quoted)
  - (+ 1 2) evaluates to 3
- ((+ 1 2)) <- error, since 3 is not a function.
  - by default, s-expressions are evaluated. We can use the quote special form to stop that: (quote (1 2 3))
    - short form: '(1 2 3) is a list containing +, 1, 2

Functional Programming Concepts

- Pure Lisp is purely functional; all other Lisps have imperative features
- All early Lisps dynamically scoped
  - Not clear whether this was deliberate or if it happened by accident
- Scheme and Common Lisp are statically scoped
  - Common Lisp provides dynamic scope as an option for explicitly-declared special functions
  - Common Lisp now THE standard Lisp
    - Very big; complicated

A Review/Overview of Scheme

- Interpreter runs a read-eval-print loop
- Things typed into the interpreter are evaluated (recursively) once
- Names: Scheme is generally a lot more liberal with the names it allows:
  - foo? bar+baz - <--- all valid names.
  - x$_%L&=*! <--- valid name
    - names by default evaluate to their value

A Review/Overview of Scheme

- Conditional expressions:
  - (if a b c) = if a then b else c
  - Example: (if (< 2 3) 4 5) ⇒ 4
  - Example 2: only one of the sub-expressions evaluates (based on if the condition is true): (if (> a b) (- a 100) (- b 100))

- Imperative stuff
  - assignments
  - sequencing (begin)
  - iteration
  - I/O (read, display)

A Review/Overview of Scheme

- Lamba expressions:
  - (lambda (x) (* x x))
  - We can apply one or more parameters to it:

    ((lambda (x) (* x x)) 3 3)

    (* 3 3)

    9
- Bindings: (let ((a 1) (b 2)) (+ a b))
  - in let, all names are bound at once. So if we did:

    (let ((a 1) (b a)) (+ a b))

    - we'd get name from outer scope. It prevents recursive calls.
  - letrec puts bindings into effect while being computed (allows for recursive calls):

    (letrec ((fac (lambda (x) (if (= x 0) 1 (* x (fac (- x 1))))))) (fac 10))

A Review/Overview of Scheme

- Define binds a name in the global scope:

```
(define square (lambda (x) (* x x)))
```

- Lists:
  - pull apart lists:

```
(car '(1 2 3)) -> 1
(cdr '(1 2 3)) -> (2 3)
(cons 1 '(2 3)) -> (1 2 3)
```

- Equality testing:
  - (= a b) <- numeric equality
  - (eq? 1 2) <- shallow comparison
  - (equal? a b) <- deep comparison

A Review/Overview of Scheme

- Control-flow:
  - (begin (display "foo") (display "bar"))
- Special functions:
  - eval = takes a list and evaluates it.

    A list: '(+ 1 2) -> (+ 1 2)

    Evaluation of a list: (eval '(+ 1 2)) -> 3
  - apply = take a lambda and list: calls the function with the list as an argument.

A Review/Overview of Scheme

- Evaluation order:
  - applicative order:
    - evaluates arguments before passing them to a function:

((lambda (x) (* x x)) (+ 1 2))

((lambda (x) (* x x) 3)

(* 3 3)

9

  - normal order:
    - passes in arguments before evaluating them:

((lambda (x) (* x x)) (+ 1 2))

(* (+ 1 2) (+ 1 2))

(* 3 3)

9

  - Note: we might want normal order in some code.

(if-tuesday (do-tuesday)) // do-tuesday might print something and we want it only if it's Tuesday

A Review/Overview of Scheme

- ((lambda (x y) (if x (+ y y) 0) t (* 10 10))
- Applicative order:

((lambda (x y) (if x (+ y y)) t 100)

(if t (+ 100 100) 0)

(+ 100 100)

200

- (four steps !)
- Normal Order:

(if t (+ (* 10 10) (* 10 10)) 0)

(+ (* 10 10) (* 10 10))

(+ 100 (* 10 10))

(+ 100 100)

200

- (five steps !)

A Review/Overview of Scheme

- What if we passed in nil instead?
- ((lambda (x y) (if x (+ y y) 0) nil (* 10 10))
- Applicative:

((lambda (x y) (if x (+ y y)) nil 100)

(if nil (+ 100 100) 0)

0

- (three steps!)
- Normal

(if nil (+ (* 10 10) (* 10 10)) 0)

0

- (two steps)
- Both applicative and normal order can do extra work!
- Applicative is usually faster, and doesn't require us to pass around closures all the time.

A Review/Overview of Scheme

- Strict vs Non-Strict:
  - We can have code that has an undefined result.
    - (f) is undefined for

      (define f (lambda () (f))) - infinite recursion

      (define f (lambda () (/ 1 0)) - divide by 0.
  - A pure function is:
    - strict if it is undefined when any of its arguments is undefined,
    - non-strict if it is defined even when one of its arguments is undefined.
  - Applicative order == strict.
  - Normal order == can be non-strict.
  - ML, Scheme (except for macros) == strict.
  - Haskell == nonstrict.

A Review/Overview of Scheme

- Lazy Evaluation:
  - Combines non-strictness of normal-order evaluation with the speed of applicative order.
  - Idea: - Pass in closure. - Evaluate it once. - Store result in memo. - Next time, just return memo.
  - Example 1: ((lambda (a b) (if a (+ b b) nil)) t (expensivefunc))

    (if t (+ (expensivefunc) (expensivefunc)) nil)

    (+ (expensivefunc) (expensivefunc))

    (+ 42 (expensivefunc)) <- takes a long time.

    (+ 42 42) <- very fast.

    84
  - Example2: ((lambda (a b) (if a (+ b b) nil)) nil (expensivefunc))

    (if nil (+ (expensivefunc) (expensivefunc)) nil)

    nil ➔ never evaluated expensivefunc! win!

Currying

Named for Haskell Curry

- Example: let a function add that take two arguments:

int add(int a, int b) { return a + b; }

- with the type signature:

(int, int) -> int , i.e., takes 2 integers, returns an int.

- We can curry this, to create a function with signature:

int -> (int -> int)

- using the curried version:

f = add(1)

print f(2)

-> prints out 3.

- Really useful in practice, even in non-fp languages.
- Some languages use currying as their main function-calling semantics (ML): **fun add a b : int = a + b;** ML's calling conventions make this easier to work with: **add 1**

  add 1 2 (There's no need to delimit arguments.)

Pattern Matching

- It's common for FP languages to include pattern matching operations:
  - matching on value,
  - matching on type,
  - matching on structure (useful for lists).
- ML example:

```
fun sum_even l =
    case l of
            nil => 0
          | b :: nil => 0
          | a :: b :: t => h + sum_even t;
```

Memoization

- Caching Results of Previous Computations (LISP):

```
(defun fib (n) (if (<= n 1) 1 (+ (fib (- n 1)) (fib (- n 2)))))
(setf memo-fib (memo #'fib))
(funcall memo-fib 3)
 => 3
(fib 5)
=> 8
(fib 6)
=> 13)
```

LISP

```
(+ 2 2)
 => 4
(+ 1 2 3 4 5 6 7 8 9 10)
 => 55
(- (+ 9000 900 90 9) (+ 5000 500 50 5))
 => 4444)
(append '(Pat Kim) '(Robin Sandy))
 => (PAT KIM ROBIN SANDY)
'(pat Kim)
 => (PAT KIM))
```

LISP

```
(setf p '(John Q Public))
(first p))
(rest p))
(second p))
(third p))
(fourth p))
(length p))
(setf names '((John Q Public) (Malcolm X) (Miss Scarlet))
(first (first names))
 => JOHN)
(apply #'+ '(1 2 3 4))
 => 10
```

LISP

```
(remove 1 '(1 2 3 2 1 0 -1))
 => (2 3 2 0 -1)
```

- Destructive lists:

```
(setq x '(a b c))
(setq y '(1 2 3))
(nconc x y)
 => (a b c 1 2 3)
x
 => (a b c 1 2 3)
y
 => (1 2 3)
```

Functional Programming in Perspective

- Advantages of functional languages
  - lack of side effects makes programs easier to understand
  - lack of explicit evaluation order (in some languages) offers possibility of parallel evaluation (e.g. MultiLisp)
  - lack of side effects and explicit evaluation order simplifies some things for a compiler
  - programs are often surprisingly short
  - language can be extremely small and yet powerful

Functional Programming in Perspective

- Problems
  - difficult (but not impossible!) to implement efficiently on von Neumann machines
    - lots of copying of data through parameters
    - frequent procedure calls
    - heavy space use for recursion
    - requires garbage collection
    - requires a different mode of thinking by the programmer
    - difficult to integrate I/O into purely functional model

Functional Programming in Perspective

- Other languages are embracing and integrating the concepts of Functional Programming:
- Java 7 - Higher Order Functions:
  - Types: #(int(int, int))
  - Methods: Math#add(int, int) – static
    Math#add(int, int) – dynamic method
    Math#() - constructor
  - If an interface contains one method, then a method with the right signature can be an instance that implements that interface: button.addActionListener(this#onButton(ActionEvent))
  - Also adds inner methods, anonymous inner methods.