Programming Languages and Paradigms
COMP 3109

Lecture 2: Functional Programming

Why Functional?

- ***Fun***ctional Programming is ***Fun***!
- Reflects the elegance of mathematics
- Pros
  - Simpler to learn
  - Higher productivity
  - Higher reliability
- Cons
  - Slower to execute
  - Inadequate for some problems

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—hence the notation's name.
  – Lambda calculus was the inspiration for functional programming
  – one uses it to compute by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions

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

Functional Programming Concepts

- Necessary features, many of which are missing in some imperative languages
  - 1st class and high-order functions
  - serious polymorphism
  - powerful list facilities
  - recursion
  - structured function returns
  - fully general aggregates
  - garbage collection

Functions

- Function f has a domain D and a range R
- Signature of a function f: $D \rightarrow R$
- Function f is a subset of $D \times R$ such that if (x,y_1) and (x,y_2) in f, then $y_1=y_2$.
- All elements $x \in D$ are associated unique elements $y \in R$, written as $f(x)$.
- Example: f: $\Re \rightarrow \Re$
  $x \rightarrow (x+1)/3$

Expressions

- Expressions are compositions of functions
- Constants are functions with no domain
- Example

  append(append(“a”,”b”),”c”)

  if(x,if(y,1,2),3)
- Recursive Example
  - Apply f: $x \rightarrow (x+1)/3$ to itself: $f(f(f(\ldots)))$

Functional Programming

- A functional program is an expression E
- Expression E represents program and input
- Expression is rewritten E by rewrite rules
  - Reductions replace sub-expression P of E by P' according to rewrite rules
  - Schematic notation:
    $E[P] \Rightarrow E[P']$
    where $P \Rightarrow P'$ holds according to rewrite rules.
- Reduction is repeated until no reductions are applicable
- Result E^* is called Normal Form

Example

- Algebraic Expression $(1+2)*(3+2)$
- Rewrite Rules (aka. Reduction System)
  - $a+b \Rightarrow c$ where c is addition of a and b
  - $a*b \Rightarrow c$ where c is multiplication of a and b
- Reductions / Normal Form

Step	E	P	P'	
1	(1+2)*(3+2)	1+2	3	
2	3*(3+2)	3+2	5	
3	3*5	3*5	15	←Normal Form

Church-Rosser Property

- Normal form is independent of the order of the evaluation of sub-expression
- Example:
  - $(1+2)*(3+2) \Rightarrow 3*(3+2) \Rightarrow 3*5 \Rightarrow 15$
  - $(1+2)*(3+2) \Rightarrow (1+2)*5 \Rightarrow 3*5 \Rightarrow 15$

Lambda Calculus

- Expressions in Lambda Calculus may consist of
  - Variables $V=\{v_1,v_2,\ldots\}$
  - Anonymous functions (aka. Abstraction)
    - Function $\lambda x.M[x]$ denotes the function $x \rightarrow M[x]$
    - The period separates the parameter from the function body
    - Example: $f(x)=x$ is written as $\lambda x.x$
  - Function applications
    - Expression $F\ A$ denotes function application of F to A.
    - Semantics: $(\lambda x.M[x])$N is $M[x:=N]$ where $[x:=N]$ denotes substitution of N for x
    - Function application is called ***Beta-Reduction***!

Variables in Lambda Calculus

- Bound Variables
  - Abstraction "binds" variables
  - Example: $(\lambda x.zx)$ binds variable x but not z
- Free Variables
  - Variables, which are not bound are "free"
  - Example: $(\lambda x.y)$ makes variable y free

Inductive Definition

- Lambda expressions are elements of set Λ.
- Set Λ is defined inductively:

  R1) if $x \in V$, then $x \in \Lambda$,

  R2) if $x \in V$ and $N \in \Lambda$, then $(\lambda x.N) \in \Lambda$,

  R3) if $M \in \Lambda$ and $N \in \Lambda$, then $(MN) \in \Lambda$,

  R4) nothing else is in set Λ.
- Set $V=\{v_1,v_2,\ldots\}$ is the set of variables.
- Example
  - Is expression $((\lambda x.(\lambda y.(yx)))z$ in Λ?

Multiple Parameters / Currying

- *Problem:*
  - Lambda allows only a single argument for a function.
- A function with more than one argument is represented by ***Currying***
- *Idea:*
  - An n argument function returns an (n-1) argument function, which returns an (n-2) argument function, etc.

Currying Example (cont'd)

- Example
  - Average two numbers, i.e., $f(x,y)=(x+y)/2$
  - Lambda Calculus (with arithmetic), i.e., $(\lambda x.(\lambda y.\ (x+y)/2)$
- Evaluation:
  $((\lambda x.(\lambda y.(x+y)/2))\ 5\ 7) \Rightarrow$
  $((\lambda y.(5+y)/2)\ 7) \Rightarrow$
  $(5+7)/2 \Rightarrow$
  6
- Function is **partially evaluated**
  - First function returns $(\lambda y.(5+y)/2)$
  - 7 is applied to result of first function
  - (5+7)/2 is evaluated and returned

Notation

- Too many parentheses
  - use implicit parenthesis rules to make life simpler
- For function application we use association to the left
  - Expression
    $F\ M_1\ M_2 \ldots M_k$
    denotes
    $(\ldots((F\ M_1)\ M_2) \ldots M_k)$
- For function abstraction we use association to the right
  - Expression
    $\lambda x_1 x_2 \ldots x_k.f(x_1,x_2,\ldots,x_k)$
    or
    $\lambda x_1.\lambda x_2.\ \ldots\ .\lambda x_k.f(x_1,x_2,\ldots,x_k)$
    denotes
    $(\lambda x_1.\ (\lambda x_2.(\ldots(\lambda x_k.f(x_1,x_2,\ldots,x_k))\ldots)))$

Reductions

- Beta Reduction: Function Application
  $(\lambda x.M[x])N$ is $M[x:=N]$
- Alpha Reduction: Renaming of Parameters
  $(\lambda x.M[x])N$ is $(\lambda y.M[y])N$
  by applying replacement $M[x:=y]$
- Eta Reduction: Simplification
  $(\lambda x.(fx))$ is f

Untyped Lambda Calculus

- Lambda Calculus does not have primitives
  - No numbers,
  - No arithmetic operations,
  - No aggregated data structures (structs, classes, etc.)
  - No control flow structures (only recursion!)
- However, it is computationally equivalent to a Turing Machine
- How can we present data types?
  - Data types can only expressed by functions

Macros

- Lambda Expressions are assigned to labels
- Instead of spelling out lambda expressions labels can be used
- Improves readability!
- Example
  - Macro Definition: $Ident = (\lambda x.x)$
  - Macro Usage: $(Ident\ y) \Rightarrow ((\lambda x.x)\ y) \Rightarrow y$

Boolean & If

- Boolean constants
  - true = (λx.(λy.(x)))
  - false = (λx.(λy.(y)))
- Semantics of a conditional functions
  - Expression: (if <cond> <f1> <f2>)
  - If predicate <cond> is true return result of <f1> otherwise <f2>
- If-Macro
  - if = (λf.f)

Example

(if true a b) ⇒

((λf.f) true a b) ⇒	;expansion of if
((((λf.f) true) a) b) ⇒	;associates to the left
((((λf.f) (λx.(λy.(x)))) a) b) ⇒	;expansion of true
((((λx.(λy.(x)))) a) b) ⇒	;evaluation of (λf.f)
((λy.(a)) b) ⇒	; evaluation of true
a	

Extensions

- Logical NOT
  - not = λf.λx.λy.(f y x)
- Example
  - (not true) =

((λf.λx.λy.(f y x)) (λx.(λy.(x)))) ⇒	; expansion
((λf.λx.λy.(f y x)) (λa.(λb.(a)))) ⇒	; renaming/conflict
(λx.λy.((λa.(λb.(a))) y x)) ⇒	; evaluation of f
(λx.λy.(λb.(y)) x) ⇒	; evaluation of a
λx.λy.y = false	

Extensions (cont'd)

- Logical OR
  - or = λf.λg.λx.λy.(f x (g x y))
- Logical AND
  - and = λf.λg.λx.λy.(f (g x y) y)
- How to prove it?
  - Evaluate for all four combinations, e.g.,

    (or true true) = true
    (or false true) = true
    (or true false) = true
    (or false false) = false

Recursive Data Types

- How to present non-negative integers in Lambda Calculus?
- Non-negative integers can be defined inductively, i.e.,
  - Basis Clause
    - 0 is a number and in the set of natural numbers
  - Inductive Clause
    - For any element x in the non-negative integers, x+1 is element of the natural numbers.
  - Extremal Clause
    - Nothing is in the set of non-negative integers unless it is obtained by the inductive clause and basis clause
- Every number has a successor (or child)
- Without the extremal clause 0.5, 1.5, etc. would be in the set of natural numbers

Natural Numbers in Lambda

- Natural Numbers in Lambda Calculus have two constructors
  - Zero
  - Successor, i.e. give me next number
- Representation of Zero
    - zero = λx.λy.y
    - s = λx.λy.λz. (y (x y z))
- Numbers
    - zero = λx.λy.y
    - 1 = (s zero) = λx.λy.(x y)
    - 2 = (s s zero) = λx.λy.(x x y)
    - ….
    - n = (s s …. zero)
    - = λx.λy.(x…{n-2 times}…(xy)….)

Programming Languages and Paradigms

Computations

- Arithmetic addition
  add =(λnfx.f (n f x))
- Arithmetic multiplication
  mult =(λmnf.m(nf))
- Proofs are more elaborate than logical AND and OR

Programming Languages and Paradigms

LISP

- LISP is an old programming language
- Invented in 1958 by John McCarthy
- Was very popular in the AI boom
- Is a functional programming language
- Is a practical implementation of Lambda Calculus

Programming Languages and Paradigms

LiSP - means List Processing

- LISP has atoms
  - Numbers, eg. 10
  - Identifiers, eg. Foo
  - Strings, eg. “filename”
- LISP has lists
  - can contain other lists
  - can contain atoms
  - can be empty
- Syntax:
  - <object> := <atoms> | <list>
  - <list> := “(“ {<object>} “)”

List Examples in LISP

- (1 2 3)
- ()
- (+ 1 2)
- (* (+ 1 2) (- 2 3))
- (sq 1 2)
- (setq a 100)
- (defun sq(n) (* n n))
- (let ((a 6)) a)
- (if t 5 6)
- (cons 5 6)
- (cons (cons 6 7))

Concept of LISP

- LISP has as data structure model
  - Lists
  - Atoms
- Even programs are written as lists
- No other data structures exist

Evaluation

- Prefix notation of function calls as lists
  - Operation is first element
  - Second and following elements are arguments
  - (<operation><arg_1> …. <arg_n>)
- Examples
  - (+ 4 2)
  - (+ 3 (+3 2))
  - (sq (* 4 2))

Numerical Functions

- Numerical operations
  - (+ 1 2 3)
  - (- 1 2)
  - (* 1 3 4)
  - (/ 2 2)
- Square Root (sqrt x)
- Base Exponent (expt x y)
- Trigonometric Functions, (sin x) …
- Absolute Value (abs x)
- Modulo (mod x y)
- Rounding (round x)

Interaction

- Interaction with lisp is done in a
  - ***read-eval-print loop***
- Loop consists of following steps
  - Parse input and construct LISP object
  - Evaluate LISP object to produce output
  - Print output object
- Example:

  ➢(+ 1 2)

  3

Variables

- Variables can be defined by
  (setq<var><value>)
- Semantics
  <var> = <value>
- Occurrence of variable symbol replaces variable symbol by the value of the variable
- Examples

  ➢(setq a (+ 5 3))

  8

  ➢A

  8

Programming Languages and Paradigms

Quote

- If lists should not be evaluated use function quote
- Example
  - ➢ (setq a (+ 1 2))

  3
  - ➢ (setq a (quote (+ 1 2)))

  (+ 1 2)
- There is a short form, just used ‘
- Example
  - ➢ (setq a ‘(+ 1 2))

  (+ 1 2)

Programming Languages and Paradigms

Conditional Function

- Definition

(if <cond><true-value><false-value>)

- Boolean values in LISP are given by two symbols
  - symbol **nil** (is equal to empty list ie. ()) represents *false*
  - **T** represents *true*
- Example
  - ➢ (if nil 1 2)

  2
  - ➢ (if (= 10 10) 1 2)

  1
  - ➢ (if () 1 2)

  2

Programming Languages and Paradigms

Predicates

- Type Checking Predicates
  - (atom x) checks whether x is not a list, ie. number, symbol, string
  - (intergerp x) checks whether x is an integer
  - (numberp x) checks whether x is a number
  - (stringp x) checks whether x is a string
- Numeric Predicates
  - (oddp x) checks whether x is integer and odd
  - (evenp x) checks wheter x is integer and even
- Equality
  - (equal x y) checks equality
  - (eql x y) checks identity
  - (= x y) check numerical equality
- Logical operators
  - (or x y) for logical or
  - (and x y) for logical and

Programming Languages and Paradigms

Functions

- Function Declaration
(defun <name> (arg1 … argn) body)
- (defun <name> (arg1 … argn) body) **denotes**
  (setq <name> '(lambda (arg_1 … arg_n) body))
- Example

```
➢ (defun factorial (x)
    (if (= x 0)
      1
      (*  x (factorial (- x 1)))
    )
  )
FACTORIAL
➢ (factorial 4)
24
```

Programming Languages and Paradigms

Bindings

- Definition
  (let ((<$name_1$><$value_1$>) ….. (<$name_n$><$value_n$>)) body)
- Example

```
➢ (let ((a 3) (b 4) (c 5))
    (+ (* a b) c)
  )
➢ a
```

Error: The variable A is unbound.

- Let allows local bindings of variables.
- Bindings might be nested
- Example (inner most binding for variable is taken)

```
➢ (let ((a 3))
    (let ((a 5))
      a))
5
```

Programming Languages and Paradigms

List Construction

- Construction with **cons**
  (cons <element> <list>)
  returns a new list with <element> as first element followed by elements by <list>
- Examples

```
➢ (cons 1 nil)
(1)
➢ (cons 'a '(bc))
(a bc)
```

- Construction with list
  (list <$elem_1$> … <$elem_n$>)
  assembles new list with elements specified as arguments
- Example

```
➢ (list 1 2 3)
(1 2 3)
```

List Access

- Access first element
  (first <list>)
- Example
  - ➢ (first '(a bc))
  a
- Access list without first element
  (rest <list>)
- Example
  - ➢ (rest '(a bc))
  (bc)