Functional Languages

Hwansoo Han
Historical Origins

- Imperative and functional models
  - 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

- A conjecture known as Church-Turing thesis
  - If an algorithm (a process that terminates) exists, there is an equivalent Turing machine, $\lambda$-calculus, and recursion
  - Equivalent computability among models
Turing Machine

- Turing’s model of computing
  - Pushdown automaton + an unbounded storage “tape”
  - 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
Lambda Calculus

- Church’s model of computing
  - Notion of parameterized expressions
    - With each parameter introduced by an occurrence of the letter $\lambda$ - hence the notation’s name.
  - $\lambda$-calculus was the inspiration for functional programming
  - Compute by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions

- Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are attempts to realize Church's $\lambda$-calculus in practical form as a programming language
Programs as Mathematical Proofs

- To prove a proposition, mathematicians use a proof
  - A program is a constructive proof for any given inputs
    - *Constructive* proof – one that shows how to obtain a mathematical object with some desired property
    - *Non-constructive* proof – one that merely shows that such an object must exist by contradiction, counting, reduction to known theorem whose proof is also non-constructive

- Logic programming is also tied to the notion of constructive proofs, but at a more abstract level:
  - Logic programmer writes a set of *axioms* that allow the *computer* to discover a constructive proof for each particular set of inputs
Functional Programming Concepts

- Do everything by composing functions, which have
  - No notion of internal state, and thus no side effects

- Features, many of which are often missing in imperative languages
  - First-class function values and high-order functions
    - *First-class* function can be a parameter, a return value, assigned to a variable
    - *High-order* function takes a function as a parameter, and return a function as a result
  - Extensive polymorphism
  - List types and operators
  - Structured function returns
  - Fully general aggregates for structured objects
  - Garbage collection
Recursion in Functional Languages

- How get anything done in a functional language?
  - Recursion (especially tail recursion) takes the place of iteration
  - In general, you can get the effect of a series of assignments

\[
\begin{align*}
  x &:= 0 \quad \ldots \\
  x &:= \text{expr1} \quad \ldots \\
  x &:= \text{expr2} \quad \ldots 
\end{align*}
\]

from \( f3(f2(f1(0))) \), where each \( f \) expects the value of \( x \) as an argument, \( f1 \) returns \( \text{expr1} \), and \( f2 \) returns \( \text{expr2} \)
Recursion in FL (cont’d)

- Recursion even does a nifty job of replacing a looping

```java
x = 0; i = 1; j = 100;
while (i < j) {
    x = x + i*j;
    i = i + 1;
    j = j - 1
}
return x
```

becomes

\[ f(0,1,100) \text{, where} \]

\[ f(x, i, j) = \begin{cases} 
    f(x+i*j, i+1, j-1) & \text{if } i < j \\
    x & \text{else}
\end{cases} \]
Scheme – a Lisp descendant

- Scheme is a particularly elegant Lisp
  - Interpreter runs a read-eval-print loop
  - Things typed are evaluated recursively once
  - Anything in parentheses is a function call (unless quoted)
    - Parentheses are not just grouping
    - Adding a level of parentheses changes meaning

\[(+ 3 4) \Rightarrow 7\]
\[7 \Rightarrow 7\]

(load “my_scheme_prog”) \Rightarrow evaluate from the loaded file

\[((+ 3 4)) \Rightarrow ERROR \text{ eval: 7 is not a procedure}\]
Scheme

- **Values**
  - Boolean values: `#t` and `#f`
  
- **Numbers**

- **Quoting**
  
  - `(quote (+ 3 4)) ⇒ (+ 3 4) ; quoted ( ) is not function
  - `'+(3 4) ⇒ (+ 3 4)` ; ’ is the same as quote

- **Conditional Expression**
  
  - `(if (< 2 3) 4 5) ⇒ 4`
  - `(cond ((< 3 2) 1)
    ((< 4 3) 2)
    (else 3)) ⇒ 3`
Lambda expressions - function

(lambda (a b) (if (< a b) a b))

Nested scopes with let and letrec (for recursion)

(let (list_of_pairs the_rest) ; list_of_pairs: var for the_rest
(let ((a 3)
    (b 4)
    (square (lambda (x)(* x x)))
    (plus +))
  (sqrt (plus (square a) (square b)))))) ⇒ 5.0 ; sqrt is a built-in func

(let ((a 3)) ; scope of binding produced by let is for the 2nd arg only
  (let ((a 4)
    (b a)) ; a takes the value of outer a – names are visible at the end
  (+ a b))) ⇒ 7
Scheme (cont’d)

- **Equivalence**

  (if (= n 1) 1 3)

  (eq? a b) ; a and b refer to the same object?
  (eqv? a b) ; a and b are semantically equivalent? (in different locations)
  (equal? a b) ; a and b are the same recursive structure?

- **List operators**

  (car '(2 3 4)) ⇒ 2
  (cdr '(2 3 4)) ⇒ (3 4) (cdr '(2)) ⇒ ()
  (cons 2 '(3 4)) ⇒ (2 3 4) (cons 2 3) ⇒ (2 . 3) ; an improper list

- **Type predicates**

  (boolean? x) (char? x) (string? x) (symbol? x)
  (number? x) (pair? x) (list? x) (complex? x)
  (rational? x) (real? x) (integer? x) ...
Imperative stuff

- Vector type
  - To access an arbitrary element in a sequence
  - Indexed by an integer, like an array, but elements can be heterogeneous types

Assignments: 
(set! x 3)  (set-car! l '(c d))  (set-cdr! l '(e))

Sequencing: 
(begin
  (display “hi”)
  (display “mom”))

Iteration (for-each)

I/O (read, display)
Evaluation Order Revisited

- **Applicative order** (as in Scheme)
  - What you're used to in imperative languages
  - Usually faster

- **Normal order**
  - Like *call-by-name* – don't evaluate arg until you need it
  - Sometimes faster

<table>
<thead>
<tr>
<th>applicative order</th>
<th>normal order</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(double (* 3 4))</code></td>
<td><code>(double (* 3 4))</code></td>
</tr>
<tr>
<td>⇒ <code>(double 12)</code></td>
<td>⇒ <code>(+ (* 3 4) (* 3 4))</code></td>
</tr>
<tr>
<td>⇒ <code>(+ 12 12)</code></td>
<td>⇒ <code>(+ 12 (* 3 4))</code></td>
</tr>
<tr>
<td>⇒ 24</td>
<td>⇒ <code>(+ 12 12)</code></td>
</tr>
<tr>
<td></td>
<td>⇒ 24</td>
</tr>
</tbody>
</table>
Strict vs. Non-strict Functions

- Strict function vs. non-strict function
  - A *strict* function requires all arguments to be well-defined (successfully evaluated), so applicative order can be used
    - Scheme, ML

- A *non-strict* function does not require all arguments to be well-defined (some may fail to be evaluated at run-time); it requires normal-order evaluation (lazy evaluation)
  - Miranda, Haskel
Higher-Order Functions

- Higher-order functions
  - Takes a function as argument or return a function as a result

- Map
  
  \[(\text{map} \times \langle 2\ 4\ 6 \rangle \ \langle 3\ 5\ 7 \rangle) \Rightarrow \langle 6\ 20\ 42 \rangle]\n
- Reduce (folding)

  \[
  \text{(define fold (lambda (f i 1)}
  \[
  \quad \text{(if (null? 1) i ; i is commonly the identity element for f}
  \[
  \quad \quad \text{(f (car 1) (fold f i (cdr 1))))}}
  \]
  \]
  \[
  \text{(define total (lambda (l) (fold + 0 l))}}
  \]
  \[
  \text{(total \langle 1\ 2\ 3\ 4\ 5 \rangle) \Rightarrow 15}
  \]
  \[
  \text{(define total-all (lambda (l) (map total l))}}
  \]
  \[
  \text{(total-all \langle\langle 1\ 2\ 3\ 4\ 5 \rangle\ (2\ 4\ 6\ 8\ 10 \rangle\ (3\ 6\ 9\ 12\ 15)) \rangle \Rightarrow \langle 15\ 30\ 45 \rangle}
  \]
Functional Programming: Pros

- Advantages
  - 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: Cons

Problems

- Difficult – but not impossible! – to implement efficiently on von Neumann machines
- Lots of copying of data through parameters
- Apparent need to create a whole new array in order to change one element
- Heavy use of pointers (space/time and locality problem)
- 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
Guile

- Scheme interpreter in Linux
  - Install guile from the “Interpreters”

```
$ guile
guile> (+ 3 4)
7

guile> (map * '(1 2 3 4) '(1 2 3 4))
(1 4 9 16)

guile> (load “my_prog.scm”) ; load insert func. from file

guile> (insert ‘a ‘(b c d))

guile> (quit)
$
```