Functional Languages

4190.310
Programming Languages
Spring 2014

Lecture 05

Reading assignments: Chapter 10

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

Turing Machine

- 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

Lambda Calculus

- 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

Necessary Features in Functional Programming

- Many of which are missing in some imperative languages

- 1st class and high-order functions
- Serious polymorphism
- Powerful list facilities
- Structured function returns
- Fully general aggregates
- Garbage collection

How do You Get Anything Done?

- Recursion (especially tail recursion) takes the place of iteration
- In general, you can get the effect of a series of assignments

```
x := 0       ...
x := expr1   ...
x := expr2   ...
```

- From f3(f2(f1(0))), where each f expects the value of x as an argument, f1 returns expr1, and f2 returns expr2

Tail Recursion

- Tail call
  - Suppose function g calls function f
  - A call to f in the body of g is a tail call if g returns the result of calling f without any further computation
- Tail recursion
  - A function is tail recursive if all recursive calls in the body of f are tail calls to f

```
int g(int n) {
    if (n == 0)
        return f(x);
    else
        return (f(x) + n);
}
```

```
int f(int n) {
    if (n == 0)
        return f(x);
    else
        return f(n-1);
}
```

Recursion

- Recursion even does a nifty job of replacing looping

```
x := 0; i := 1; j := 100;
while i < j do
  x := x + i*j; i := i + 1;
  j := j - 1
end while
return x
```

- Becomes f(0, 1, 100), where
  f(x, i, j) == if i < j then f (x+i*j, i+1, j-1) else x

Higher-order Functions

- Thinking about recursion as a direct, mechanical replacement for iteration, however, is the wrong way to look at things
  - One has to get used to thinking in a recursive style

- Even more important than recursion is the notion of higher-order functions
  - Take a function as argument, or return a function as a result
  - Great for building things

Variants of Lisp

- Pure (original) Lisp
- Interlisp, MacLisp, Emacs Lisp
- Common Lisp
- Scheme

Variants of Lisp (contd.)

- 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 statically scoped
  - Common Lisp provides dynamic scope as an option for explicitly-declared special functions
  - Common Lisp now THE standard Lisp
    - Very big; complicated (The Ada of functional programming)

Variants of Lisp (contd.)

- Scheme is a particularly elegant Lisp

- Other functional languages
  - ML
  - Miranda
  - Haskell
  - FP

- Haskell is the leading language for research in functional programming

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 (provided you don't blow it in other ways)
- Programs are often surprisingly short
- Language can be extremely small and yet powerful

Problems of Functional Languages

- 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

Problems of Functional Languages (contd.)

- 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