Lecture 11: Functional Programming Concepts

COMP 524 Programming Language Concepts
Stephen Olivier
February 26, 2009

Based on slides by A. Block, notes by N. Fisher, F. Hernandez-Campos, and D. Stotts

The University of North Carolina at Chapel Hill

Goals

- Discuss functional languages

Functional Features

- Most functional languages Provide
  - Functions as first-class values
  - Higher-order functions
  - List Type (operators on lists)
  - Recursion
  - Structured function return
  - Garbage collection
  - Polymorphism and type inference

So Why Functional?

- Teaches truly recursive algorithms
- Implicit Polymorphism
- Natural expressiveness for symbolic and algebraic computations
  - Algorithms clearly map to the code that implements them

History

- Lambda-calculus as semantic model (Church)
- LISP (1958, MIT, McCarthy)

```
(defun fib (n)
  (if (or (= n 0) (= n 1))
  1
  (+ (fib (- n 1))
     (fib (- n 2)))))
```

History

- Lisp
  - Dynamic Scoping
- Common Lisp (CL), Scheme
  - Static scoping
- ML
  - Typing, type inference, fewer parentheses
- Haskell, Miranda
  - pure functional

LISP Properties

- Homogeneity of programs and data
  - Programs are lists and a program can examine/change itself while running
- Self-Definition
  - Easy to write a Lisp interpreter in Lisp

Can Programming be Liberated from the von Neumann Style?

- This is the title of a lecture given by John Backus when he received the Turing Award in 1977.
- In this, he pointed out that the program should be abstract description of algorithm rather than sequences of changes in the state of the memory.
  - He called for raising the level of abstraction
  - A way to realize this goal is functional programming
- Programs written in modern functional programming languages are a set of mathematical relationships between objects
  - No explicit memory management takes place

Evaluation Order

- Functional programs are evaluated following a reduction (or evaluation or simplification) process
- There are two common ways of reducing expressions
  - Application order
    - Impatient evaluation
  - Normal order
    - Lazy evaluation

Applicative Order

- In applicative order, expressions at evaluated following the parsing tree (deeper expressions are evaluated first)

```
square (3 + 4)
= { definition of + }
   square 7
= {definition of square }
   7 * 7
= { definition of * }
  49
```

Normal Order

- In Normal order, expressions are evaluated only as their value is needed

```
square (3 + 4)
= { definition of square }
    (3 + 4) * (3 + 4)
= {definition of + applied to first term }
    7 * (3 + 4)
= {definition of + applied to second term }
  7 * 7
= { definition of * }
  49
```

Haskell Evaluation Order

- Haskell is a lazy functional programming language
  - Expressions are evaluated in normal order
  - Identical expressions are evaluated only once

```
square (3 + 4)
= { definition of square }
    (3 + 4) * (3 + 4)
= {definition of + applied both terms }
    7 * 7
= { definition of * }
  49
```

ML History

- ML Stands for “Meta-language”
- Developed in 1970s by Robert Milner at the University of Edinburgh
- Characteristics
  - Functional control structures
  - strict, formal semantics (provable correctness)
  - Strict polymorphic type system
    - Coercion not allowed
  - Still subject of active industry research
    - Microsoft promotes variant called F# for their .NET framework

Recursion

- Recursion requires no special syntax
- Recursion and logically-controlled iteration are equally powerful

$$gcd(a,b) = \begin{cases} a & \text{if} \quad a = b \\ gcd(a-b,b) & \text{if} \quad a > b \\ gcd(a,b-a) & \text{if} \quad a < b \end{cases}$$

Recursion

$$gcd(a,b) = \begin{cases} a & \text{if} \quad a = b \\ gcd(a-b,b) & \text{if} \quad a > b \\ gcd(a,b-a) & \text{if} \quad a < b \end{cases}$$

```
int gcd(int a,int b){
  if(a == b) return a;
  else if (a>b) return gcd(a-b,b);
  else return gcd(a,b);
}
```

Recursion

```
int gcd(int a,int b){
  while(a!=b){
    if(a>b) { a = a-b;}
    else {b = b-a;}
  }
  return a;
}
```

Iteration

Tail Recursion

- **Tail recursion** is when **no computation occurs after the recursive statement**.
- The advantage of tail recursion is that **space can be reused**.

```
int gcd(int a,int b){
  if(a == b) return a;
  else if (a>b) return gcd(a-b,b);
  else return gcd(a,b);
}
```

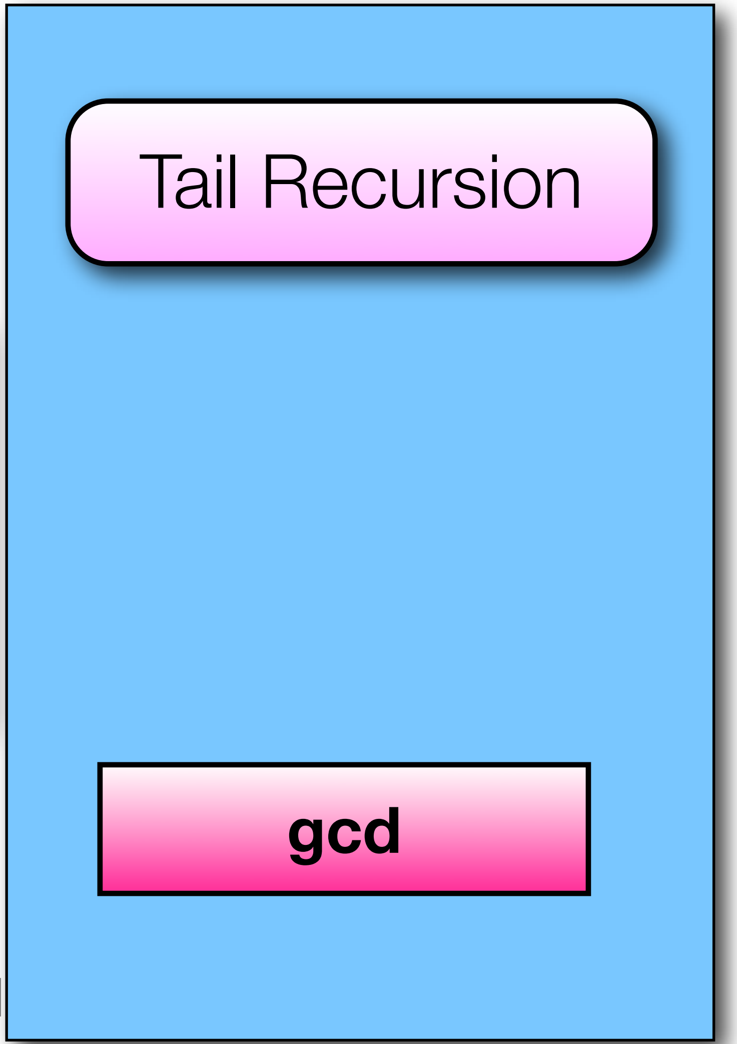
Tail Recursion

```
int gcd(int a,int b){
  int x;
  if(a == b) x=a;
  else if (a>b) x= gcd(a-b,b);
  else x= gcd(a,b);
  return x;
}
```

Not Tail Recursion

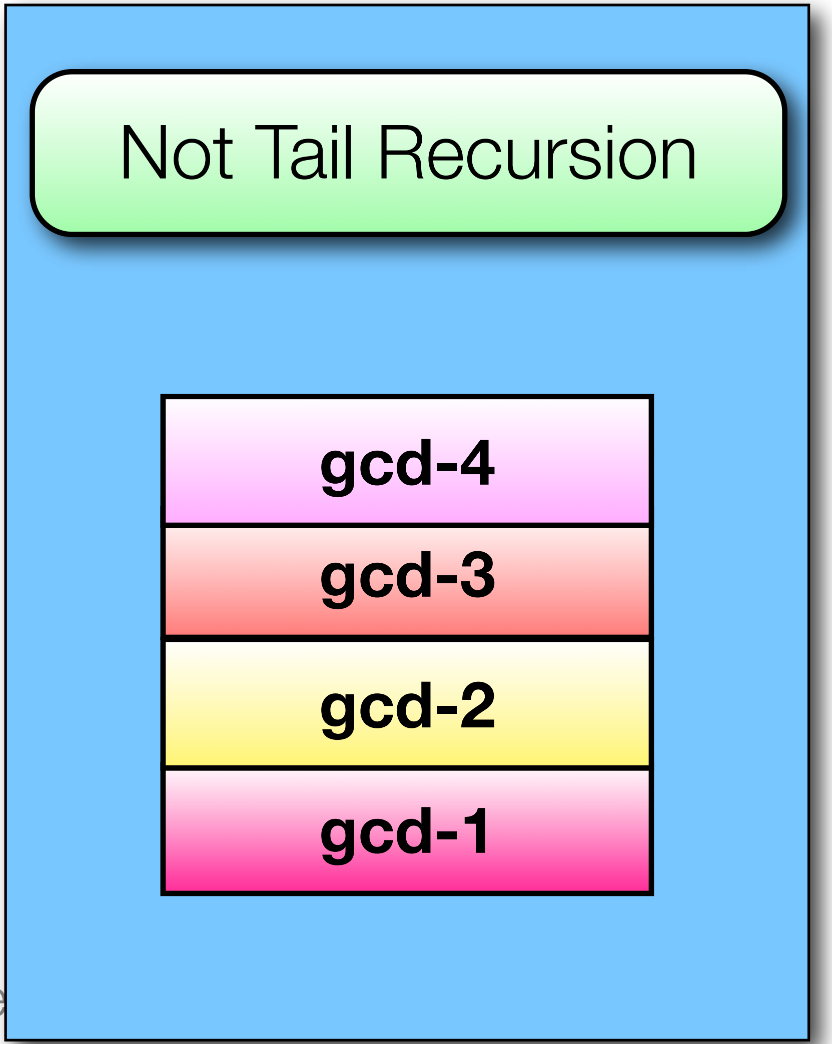
Tail Recursion

```
int gcd(int a,int b){
  if(a == b) return a;
  else if (a>b) return gcd(a-b,b);
  else return gcd(a,b);
}
```



Tail Recursion

```
int gcd(int a,int b){
  int x;
  if(a == b) x=a;
  else if (a>b) x= gcd(a-b,b);
  else x= gcd(a,b);
  return x;
}
```



Function Definitions in ML

- Tail-Recursive Functions:

```
fun fib(n)=
  let fun fib_helper(f1, f2, i) =
    if i = n then f2
    else fib_helper(f2, f1+f2, i+1)
  in
    fib_helper(0,1,0)
  end;
fib(7);
```

- On good implementations, equivalent in speed (and sometimes machine code) to iterative version!
- What is the **inferred** type of this function?

Types in ML

- Built-in Types:
  - Integer
  - Real
  - String
  - Char
  - Boolean
- From these we can construct
  - **Tuples**: Heterogeneous element types with finite fixed length
    - `(#“a”, 5, 3.0, “hello”, true): char *int *real*string*bool`
  - Lists:
    - `[5.0, 3.2, 6.7] : real list`
    - `[(# “a”, 7), (# “b”, 8)]: (char *int)list`
  - Functions
  - Records

Types inference in ML

- Everything is inferred; ML complains if anything is ambiguous.

```
fun circum(r) = r * 2.0 * 3.14159;
circum(7.0);
```

- What is the inferred type of **r**? Why?
- How about the function?
  - **r must be of type real**.
    - Can be explicit by defining fun `circum(r:real)`...
  - Type of function circum:
    - real->real

Polymorphism in ML

- Consider the following function in ML:

```
fun compare(x,p,q) =
  if x = p then
    if x = q then “both”
    else “first”
  else
    if x = q then “second”
    either “neither”
```

Polymorphism in ML

- Consider the following function in ML:

```
fun compare(x,p,q) =
  if x = p then
    if x = q then “both”
    else “first”
  else
    if x = q then “second”
    either “neither”
```

What is type of compare? x? p? q?

Polymorphism in ML

- Consider the following function in ML:

```
fun compare(x,p,q) =
  if x = p then
    if x = q then “both”
    else “first”
  else
    if x = q then “second”
    either “neither”
```

What is type of compare? x? p? q?

`` `a*`a*`a->string ``

All of these are valid:
compare(1,2,3);
compare(1,1,1);
let val t = (“larry”, “moe”, “curly”) in compare(t) end;

```
fun compare(x,p,q) =
  if x = p then
    if x = q then “both”
    else “first”
  else
    if x = q then “second”
    either “neither”
```

Type Checking

- ML verifies **type consistency**
- Set of constraints
  - All occurrences of same identifier (in same scope) have the same type.
  - In an `if...then..else... construct`, if condition must have type `bool`, and `then` and `else` must have same type.
  - Programmer defines functions have type `` `a->`b `` where `` `a `` is type of function parameters and `` `b `` is type of function return.
  - When function is called, the arguments passed and value returned must have same type as definition.
- Process of checking if two types are the same is called **unification**.

Lists in ML

- Heterogeneous & Homogeneous Lists operator:
  - Appending (joining) two lists:

```
[1,4]@[3,5] => [1,4,3,5]
(“hi”, 3.0)@(4, “bye”) => (“hi”, 3.0, 4, “bye”)
```

  - Prefixing a list with an item:

```
1::[2,7,9] => [1,2,7,9];
NOTE: [2,7,9]::1 is illegal (use [2,7,9]@[1] instead)
```

Lists in ML

- Heterogeneous & Hon
  - Appending (joining) two

```
[1,4]@[3,5] => [1,4,
("hi", 3.0)@(4, "bye
```

  - Prefixing a list with an i

```
1::[2,7,9] => [1,2,7,9]
NOTE: [2,7,9]::1 is il                    ad)
```

Other useful List functions:
hd= head
tl = tail
nth = list element selctor
rev =reverse a list
length = number of elements

Function Pattern Matching in ML

- Function definition as series of alternatives:

```
fun appends(l1, l2) =
  if l1 = nil then l2
  else hd(l1) :: append (tl(l1), l2);
```

- Becomes

```
fun append(nil, l2) = l2
  | append (h::t, l2) = h :: append(t, l2);
```

Function Pattern Matching in ML

- More complex example

```
fun split(nil) = (nil, nil)
  | split([a]) = ([a], nil)
  | split(a::b::cs) =
    let val (M,N) = split(cs)
  in
    (a::M,b::N)
  end;
```

Higher-Order Functions

- Higher-order functions are functions that take functions as arguments and/or return functions

```
fun map(F, nil) = nil
  | map(F, x::cs) = F(x)::map(F,xs);
```

To Add 5 to every integer we could...

```
fun add5(x) = x+5;
map add5, [3,24,7,9]; => [8,29,12,14]
```

```
map (fn x=> x+5) [3,24,7,9]; => [8,29,12,14]
```

“Currying” in ML

- **Currying** is a method in which a **multiple argument function** is replaced by a **single argument function** that **returns a function with the remaining arguments**.

```
fun add(x,y) = x + y : int;
>> val add = fn int * int -> int
```

```
fun add x = fn y=> x+y;
>> val add = fn int -> int ->int
```

```
fun add x y = x+y;
>> val add = fn int -> int ->int
```

Standard ML of New Jersey

- Download and Install from
  - http://www.smlnj.org/smlnk.html
- to run (after installation): Type "sml"
- Try some stuff from Stott's ML Class notes.
- If you want to exit type:
  - OS.Process.exit(OS.Process.success);
  - OR press Ctrl-D

Scope in ML is Lexical

- Top level environment has all **pre-defined bindings**
- Every **val** binding adds another row to the symbol table when compiling/interpreting
- Each **row hides earlier bindings** of the same name (does not destroy them)
- Local bindings can be made in functions definitions
- **Locals are removed from the symbol stack** when the function definition is complete

Binding of Referencing Environments

- Scope rules are used to determine the reference environment
    - Static and dynamic scoping
- Some languages allow **references to subroutines**
    - Are the scope rules applied when the reference is created or when the subroutine is called?
- In **shallow (late) binding**, the referencing environment is bound when the subroutine is called
- In **deep (early) binding**, the referencing environment is bound when the reference is created.

ML Example

```
- val x = 5;
val x = 5 : int
- fun wow z = z + x;
val wow = fn : int -> int
- wow 9;
val it = 14 : int
- val x = 10;
val x = 10 : int
- wow 9; (* wow uses x = 5 *)
val it = 14 : int
```

- Earlier binding of x is used due to static scope rules

ML Example

```
- val x = 5;
val x = 5 : int
```

What if we pass the function wow as an argument to another function declared after the new binding x = 10 has been created?

```
- wow 9; (* wow uses x = 5 *)
val it = 14 : int
```

- Earlier binding of x is used due to static scope rules

ML Example

```
- val x = 5;
val x = 5 : int
- fun wow z = z + x;
val wow = fn : int -> int
- val x = 10;
val x = 10 : int
- fun twice (a,b) = b (a * 2);
val twice = fn : int * (int -> 'a) -> 'a
- twice (3, wow); (* still uses x = 5 *)
val it = 11 : int
```

- Deep binding uses original reference environment

Deep and Shallow Binding

- **Deep Binding, Shallow Binding** are both concepts related to giving a function/subroutine a referencing environment in which to run.
- This is important when a subprogram is passed in our out as a parameter to another (i.e., a “funarg”).

Some questions:
When a funarg that is passed in is run, does it use the environment it has when run? or the when when defined?
Also, when a funarg is passed out and run the environment it created in is gone; how do we deal with that?

Closures

- Deep binding is implemented using **closures**
  - Remember them from chapter 3?
- A closure is the **combination of a reference to a subroutine and an explicit representation of its referencing environment**

Referential Transparency

- Bindings are immutable.
- Any time you see a name, you may substitute in the value bound to that name and NOT alter the semantics of the expression.
- **“no side effects.”**
- Functional programing languages **try** to enforce **referential transparency**.
  - ML is not pure functional: “Don’t get lulled into a false sense of referential transparency” (from *ML for the Working Programmer*)

Referential Transparency

- **“equals can be substituted for equals”**
  - If two expressions are defined to have equal values, then one can be substituted for the other in any expression without affecting the result of the computation.
  - For example, in

```
s = sqrt(2); z = f(s,s); we can write
z = f(sqrt(2), sqrt(2));
```

Referential Transparency

- A function is called **referentially transparent** if given the same parameter(s), it always **returns the same result**.
- In **mathematics all functions** are referentially transparent
- In **programming this is not always the case**, with use of imperative features in languages.
  - The subroutine/function called could affect some global variable that will cause a second invocation to return a different value.
  - Input/Output
    - In ML, what if we replace `print “abc”` by its return value `()`

Why is referential transparency important?

- Because it allows the programmer to **reason about program behavior**, which can help in proving correctness, finding bugs that couldn't be found by testing, simplifying the algorithm, assist in modify the code without breaking it, or even finding ways of optimizing it.

```
s = sqrt(9);
x = s*s + 17 *k / (s-1);
// can replace x with:
// sqrt(9)*sqrt(9) + 17 *k/(sqrt(9)-1) = 9+17*k/2;
```

Advantages of functional programming (Scott)

- 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 for functional programming (Scott)

- 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 changeone element
  - Very heavy use of pointers (space and time and locality problem)
  - Frequent procedure calls
  - Heavy space use for recursion
  - Requires garbage collection

Problems for functional programming (Scott)

- Requires a different mode of thinking by the programmer
- Difficult to integrate I/O into purely functional model
  - Leading approach is the monads of Haskell -- sort of an imperative wrapper around a purely functional program; allows functions to be used not only to calculate values, but also to decide on the order in which imperative actions should be performed.