Functional Programming

COMS W4115

Prof. Stephen A. Edwards
Fall 2007
Columbia University
Department of Computer Science

Original version by Prof. Simon Parsons

Functional vs. Imperative

Imperative programming concerned with "how."

Functional programming concerned with "what."

Based on the mathematics of the lambda calculus (Church as opposed to Turing).

"Programming without variables"

It is inherently concise, elegant, and difficult to create subtle bugs in.

It's a cult: once you catch the functional bug, you never escape.

Referential transparency

The main (good) property of functional programming is *referential transparency*.

Every expression denotes a single value.

The value cannot be changed by evaluating an expression or by sharing it between different parts of the program.

No references to global data; there *is* no global data.

There are no *side-effects*, unlike in *referentially opaque* languages.

The Joy of Pascal

```
program example(output)
var flag: boolean;

function f(n:int): int
begin
  if flag then f := n
          else f := 2*n;
  flag := not flag
end

begin
  flag := true;
  writeln(f(1) + f(2));
  writeln(f(2) + f(1));
end
```

What does this print?

Strange behavior

This prints 5 then 4.

Odd since you expect

$$f(1)+f(2)=f(2)+f(1)$$

Mathematical functions only depend on their inputs

They have no memory

Variables

At the heart of the "problem" is fact that the global data **`flag`** affects the value of **`f`**.

In particular

```
flag := not flag
```

gives the offending behavior

Eliminating assignments eliminates such problems.

In functional languages, variables not names for storage.

Instead, they're names that refer to particular values.

Think of them as *not very variables.*

Simple functional programming in ML

A function that squares numbers:

```
% sml
Standard ML of New Jersey, Version 110.0.7
- fun square x = x * x;
val square = fn : int -> int
- square 5;
val it = 25 : int
-
```

A more complex function

```
- fun max a b =
=    if a > b then a else b;
val max = fn : int -> int -> int
- max 10 5;
val it = 10 : int
- max 5 10;
val it = 10 : int
-
```

Notice the odd type:

```
int -> int -> int
```

This is a function that takes an integer and returns a function that takes an integer and returns an integer.

Currying

Functions are first-class objects that can be manipulated with abandon and treated just like numbers.

```
- fun max a b = if a > b then a else b;
val max = fn : int -> int -> int
- val max5 = max 5;
val max5 = fn : int -> int
- max5 4;
val it = 5 : int
- max5 6;
val it = 6 : int
-
```

Tuples and arguments

You can also pass and return *tuples* to/from functions:

```
- fun max(a,b) = if a > b then a else b;
val max = fn : int * int -> int
- max (10,5);
val it = 10 : int
- fun reverse(a,b) = (b,a);
val reverse = fn : 'a * 'b -> 'b * 'a
- reverse (10,5);
val it = (5,10) : int * int
- max (reverse (10,5));
val it = 10 : int
-
```

Polymorphism

Reverse has an interesting type:

```
- fun reverse(a,b) = (b,a);
val reverse = fn : 'a * 'b -> 'b * 'a
```

This means it can reverse a two-tuple of any type and any other type:

```
- reverse (10,5.2);
val it = (5.2,10) : real * int
- reverse ("foo", 3.14159);
val it = (3.14159,"foo") : real * string
```

Recursion

ML doesn't have variables in the traditional sense, so you can't write programs with loops.

So use recursion:

```
- fun sum n =
=   if n = 0 then 0 else sum(n-1) + n;
val sum = fn : int -> int
- sum 2;
val it = 3 : int
- sum 3;
val it = 6 : int
- sum 4;
val it = 10 : int
```

Power operator

You can also define functions as infix operators:

```
- fun x ^ y =
=   if y = 0 then 1
=   else x * (x ^ (y-1));
val ^ = fn : int * int -> int
- 2 ^ 2;
val it = 4 : int
- 2 ^ 3;
val it = 8 : int
-
```

Using the same thing twice

Without variables, duplication may appear necessary:

```
fun f x =
  g(square(max(x,4))) +
  (if x <= 1 then 1
   else g(square(max(x,4))));
```

One fix: use an extra function:

```
fun f1(a,b) = b + (if a <= 1 then 1 else b);
fun f x = f1(x, g(square(max(x,4))));
```

The let expression

The easiest way is to introduce a local name for the thing we need multiple times:

```
fun f x =
  let
    val gg = g(square(max(x,4)))
  in
    gg + (if x <=1 then 1 else gg)
  end;
```

let is *not* assignment

```
- let val a = 5 in
=   (let
=      val a = a + 2
=    in
=      a
=    end,
=   a)
= end;
val it = (7,5) : int * int
```

Data Types

Programs aren't very useful if they only manipulate scalars.

Functional languages are particularly good at manipulating more complex data types.

You've already seen tuples, which is a fixed-length list of specific types of things.

ML also has *lists*, arbitrary-length vectors of things of the same type.

Lists

Tuples have parenthesis, lists have brackets:

```
- (5,3);
val it = (5,3) : int * int
- [5,3];
val it = [5,3] : int list
```

Concatenate lists with @:

```
- [1,2] @ [3,4,5];
val it = [1,2,3,4,5] : int list
```

Cons

Add things to the front with `::` (pronnounced “cons”)

```
- [1,2,3];
val it = [1,2,3] : int list
- 5 :: it;
val it = [5,1,2,3] : int list
```

Concatenating is not the same as consing:

```
- [1,2] :: [3,4];
stdIn: Error: operator and operand don't agree [l
  operator domain: int list * int list list
  operand:         int list * int list
  in expression:
    (1 :: 2 :: nil) :: 3 :: 4 :: nil
```

Other list functions

```
- null [1,2];
val it = false : bool
- null nil;
val it = true : bool
- null [];
val it = true : bool
- val a = [1,2,3,4];
val a = [1,2,3,4] : int list
- hd a;
val it = 1 : int
- tl a;
val it = [2,3,4] : int list
```

Fun with recursion

```
- fun addto (l,v) =
=   if null l then nil
=   else hd l + v :: addto(tl l, v);
val addto = fn : int list * int -> int list

- addto([1,2,3],2);
val it = [3,4,5] : int list
```

More recursive fun

```
- fun map (f, l) =
=   if null l then nil
=   else f (hd l) :: map(f, tl l);
val map = fn : ('a -> 'b) * 'a list -> 'b list

- fun add5 x = x + 5;
val add5 = fn : int -> int

- map(add5, [10,11,12]);
val it = [15,16,17] : int list
```

But why always name functions?

```
- map( fn x => x + 5, [10,11,12]);
val it = [15,16,17] : int list
```

This is called a *lambda* expression: it's simply an unnamed function.

The **fun** operator is similar to a lambda expression:

```
- val add5 = fn x => x + 5;
val add5 = fn : int -> int
- add5 10;
val it = 15 : int
```

Recursion with Lambda Expressions

Q: How do you call something recursively if it doesn't have a name?

A: Give it one.

```
- let
=   val rec f =
=     fn x => if null x then nil
=             else hd x + 1 :: f (tl x)
= in f end
= [1,2,3];
val it = [2,3,4] : int list
```

Pattern Matching

Functions are often defined over ranges

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise.} \end{cases}$$

Functions in ML are no different. How to cleverly avoid writing if-then:

```
fun map (f,[]) = []
  | map (f,l) = f (hd l) :: map(f,tl l);
```

Pattern matching is order-sensitive. This gives an error.

```
fun map (f,l) = f (hd l) :: map(f,tl l)
  | map (f,[]) = [];
```

Pattern Matching

More fancy binding

```
fun map (_,[]) = []
  | map (f,h :: t) = f h :: map(f,t);
```

“_” matches anything

h :: t matchs a list, binding **h** to the head and **t** to the tail.

Call-by-need

Most imperative languages, as well as ML, use *call-by-value* or *call-by-reference*. All arguments are evaluated before being passed (copied) to the callee.

But some functional languages use *call-by-need*.

```
- fun infinite x = infinite x;
val infinite = fn : 'a -> 'b
- fun zero _ = 0;
val zero = fn : 'a -> int
- zero (infinite 2);
```

This doesn't terminate in ML, but it does with call-by-need.

Call-by-need deterministic in the absence of side effects.

Reduce

Another popular functional language construct:

```
fun reduce (f, z, nil) = z
  | reduce (f, z, h::t) = f(h, reduce(f, z, t));
```

If **f** is "$-$", **reduce(f,z,a::b::c)** is $a-(b-(c-z))$

```
- reduce( fn (x,y) => x - y, 0, [1,5]);
val it = ~4 : int
- reduce( fn (x,y) => x - y, 2, [10,2,1]);
val it = 7 : int
```

Another Example

Consider

```
- fun find1(a,b) =
=   if (a = 1) then true else b;
val find1 = fn : int * bool -> bool

- reduce(find1, false, [3,3,3]);
val it = false : bool

- reduce(find1, false, [5,1,2]);
val it = true : bool
```

Call-by-need

```
fun find1(a,b) = if (a = 1) then true else b;
```

Call-by-value:

```
reduce(find1, false, [1, 3, 5, 7]
find1(1, find1(3, find1(5, find1(7, false))))
find1(1, find1(3, find1(5, false)))
find1(1, find1(3, false))
find1(1, false)
true
```

Call-by-need:

```
reduce(find1, false, [1, 3, 5, 7]
find1(1, find1(3, find1(5, find1(7, false))))
true
```

The Lambda Calculus

Fancy name for rules about how to represent and evaluate expressions with unnamed functions.

Theoretical underpinning of functional languages.
Side-effect free.

Very different from the Turing model of a store with evolving state.

ML:

```
fn x => 2 * x;
```

The Lambda Calculus:

$\lambda x\,.\,*\,2\,x$

English:

"the function of x that returns the product of two and x"

Bound and Unbound Variables

In $\lambda x\,.\,*\,2\,x$, x is a *bound variable*. Think of it as a formal parameter to a function.

"$*\,2\,x$" is the *body*.

The body can be any valid lambda expression, including another unnnamed function.

$$\lambda x\,.\,\lambda y\,.\,*\,(+\,x\,y)\,2$$

"The function of x that returns the function of y that returns the product of the sum of x and y and 2."

Arguments

$\lambda x\,.\,\lambda y\,.\,*\,(+\,x\,y)\,2$

is equivalent to the ML

```
fn x => fn y => (x + y) * 2;
```

All lambda calculus functions have a single argument.

As in ML, multiple-argument functions can be built through such "currying."

In this context, currying has nothing to do with Indian cooking. It is due to Haskell Brooks Curry (1900–1982), who contributed to the theory of functional programming. The Haskell functional language is named after him.

Calling Lambda Functions

To invoke a Lambda function, we place it in parentheses before its argument.

Thus, calling $\lambda x\,.\,*\,2\,x$ with 4 is written

$(\lambda x\,.\,*\,2)\,4$

This means 8.

Curried functions need more parentheses:

$(\lambda x\,.\,(\lambda y\,.\,*\,(+\,x\,y)\,2)\,4)\,5$

This binds 4 to y, 5 to x, and means 18.

Grammar of Lambda Expressions

Utterly trivial:

expr	→	*constant*
	|	*variable*
	|	*expr expr*
	|	(*expr*)
	|	λ *variable* . *expr*

Somebody asked whether a language needs to have a large syntax to be powerful. Clearly, the answer is a resounding "no."

Evaluating Lambda Expressions

Pure lambda calculus has no built-in functions; we'll be impure.

To evaluate $(+\,(*\,5\,6)\,(*\,8\,3))$, we can't start with $+$ because it only operates on numbers.

There are two *reducible expressions*: $(*\,5\,6)$ and $(*\,8\,3)$. We can reduce either one first. For example:

$(+\,(*\,5\,6)\,(*\,8\,3))$
$(+\,30\,(*\,8\,3))$
$(+\,30\,24)$
54

Looks like deriving a sentence from a grammar.

Evaluating Lambda Expressions

We need a reduction rule to handle λs:

$(\lambda x\,.\,*\,2\,x)\;4$

$(*\;2\;4)$

8

This is called β-reduction.

The formal parameter may be used several times:

$(\lambda x\,.\,+\,x\,x)\;4$

$(+\;4\;4)$

8

Beta-reduction

May have to be repeated:

$((\lambda x\,.\,(\lambda y\,.\,-\,x\,y))\;5)\;4$

$(\lambda y\,.\,-\,5\,y)\;4$

$(-\;5\;4)$

1

Functions may be arguments:

$(\lambda f\,.\,f\;3)(\lambda x\,.\,+\,x\;1)$

$(\lambda x\,.\,+\,x\;1)3$

$(+\;3\;1)$

4

More Beta-reduction

Repeated names can be tricky:

$(\lambda x\,.\,(\lambda x\,.\,+\,(-\,x\;1))\;x\;3)\;9$

$(\lambda x\,.\,+\,(-\,x\;1))\;9\;3$

$+\;(-\;9\;1)\;3$

$+\;8\;3$

11

In the first line, the inner x belongs to the inner λ, the outer x belongs to the outer one.

Free and Bound Variables

In an expression, each appearance of a variable is either "free" (unconnected to a λ) or bound (an argument of a λ).

β-reduction of $(\lambda x\,.\,E)\;y$ replaces every x that *occurs free in* E with y.

Free or bound is a function of the position of each variable and its context.

Free variables

$(\lambda x\,.\,x\,y\;(\lambda y\,.\,+\,y\;))\;x$

Bound variables

Alpha conversion

One way to confuse yourself less is to do α-conversion.

This is renaming a λ argument and its bound variables.

Formal parameters are only names: they are correct if they are consistent.

$\lambda x\,.\,(\lambda x\,.\,x)\;(+\;1\;x) \leftrightarrow_\alpha \lambda x\,.\,(\lambda y\,.\,y)\;(+\;1\;x)$

Alpha Conversion

An easier way to attack the earlier example:

$(\lambda x\,.\,(\lambda x\,.\,+\,(-\,x\;1))\;x\;3)\;9$

$(\lambda x\,.\,(\lambda y\,.\,+\,(-\,y\;1))\;x\;3)\;9$

$(\lambda y\,.\,+\,(-\,y\;1))\;9\;3$

$+\;(-\;9\;1)\;3$

$+\;8\;3$

11

Reduction Order

The order in which you reduce things can matter.

$(\lambda x\,.\,\lambda y\,.\,y)\;(\;(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)\;)$

We could choose to reduce one of two things, either

$(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)$

or the whole thing

$(\lambda x\,.\,\lambda y\,.\,y)\;(\;(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)\;)$

Reduction Order

Reducing $(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)$ effectively does nothing because $(\lambda z\,.\,z\;z)$ is the function that calls its first argument on its first argument. The expression reduces to itself:

$(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)$

So always reducing it does not terminate.

However, reducing the outermost function does terminate because it ignores its (nasty) argument:

$(\lambda x\,.\,\lambda y\,.\,y)\;(\;(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)\;)$

$\lambda y\,.\,y$

Reduction Order

The *redex* is a sub-expression that can be reduced.

The *leftmost* redex is the one whose λ is to the left of all other redexes. You can guess which is the *rightmost*.

The *outermost* redex is not contained in any other.

The *innermost* redex does not contain any other.

For $(\lambda x\,.\,\lambda y\,.\,y)\;(\;(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)\;)$,

$(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)$ is the leftmost innermost and

$(\lambda x\,.\,\lambda y\,.\,y)\;(\;(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)\;)$ is the leftmost outermost.

Applicative vs. Normal Order

Applicative order reduction: Always reduce the leftmost innermost redex.

Normative order reduction: Always reduce the leftmost outermost redex.

For $(\lambda x\,.\,\lambda y\,.\,y)\ (\ (\lambda z\,.\,z\,z)\ (\lambda z\,.\,z\,z)\ )$, applicative order reduction never terminated but normative order did.

Applicative vs. Normal Order

Applicative: reduce leftmost innermost

"evaluate arguments before the function itself"

eager evaluation, call-by-value, usually more efficient

Normative: reduce leftmost outermost

"evaluate the function before its arguments"

lazy evaluation, call-by-name, more costly to implement, accepts a larger class of programs

Normal Form

A lambda expression that cannot be reduced further is in *normal form*.

Thus,

$\lambda y\,.\,y$

is the normal form of

$(\lambda x\,.\,\lambda y\,.\,y)\ (\ (\lambda z\,.\,z\,z)\ (\lambda z\,.\,z\,z)\ )$

Normal Form

Not everything has a normal form

$(\lambda z\,.\,z\,z)\ (\lambda z\,.\,z\,z)$

can only be reduced to itself, so it never produces an non-reducible expression.

"Infinite loop."

The Church-Rosser Theorems

If $E_1 \leftrightarrow E_2$ (are interconvertable), then there exists an E such that $E_1 \rightarrow E$ and $E_2 \rightarrow E$.

"Reduction in any way can eventually produce the same result."

If $E_1 \rightarrow E_2$, and E_2 is is normal form, then there is a normal-order reduction of E_1 to E_2.

"Normal-order reduction will always produce a normal form, if one exists."

Church-Rosser

Amazing result:

Any way you choose to evaluate a lambda expression will produce the same result.

Each program means exactly one thing: its normal form.

The lambda calculus is deterministic w.r.t. the final result.

Normal order reduction is the most general.

Turing Machines vs. Lambda Calculus

In 1936,

- Alan Turing invented the Turing machine
- Alonzo Church invented the lambda calculus

In 1937, Turing proved that the two models were equivalent, i.e., that they define the same class of computable functions.

Modern processors are just overblown Turing machines.

Functional languages are just the lambda calculus with a more palatable syntax.