Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 1

Introduction and Overview

Topics covered:

- Imperative programming
- Functional programming
- The merits of functional programming
- Overview of the course
- Lambda notation and its benefits

Imperative programming

Imperative (or procedural) programs rely on modifying a *state* by using a sequence of *commands*.

The state is mainly modified by the *assignment* command, written `v = E` or `v := E`.

We can execute one command before another by writing them in sequence, perhaps separated by a semicolon: $C_1 ; C_2$.

Commands can be executed conditionally using `if`, and repeatedly using `while`.

Programs are a series of instructions on how to modify the state.

Imperative languages, e.g. FORTRAN, Algol, C, Modula-3 support this style of programming.

An abstract view

We ignore input-output operations, and assume that a program runs for a limited time, producing a result.

We can consider the execution in an abstract way as:

$$\sigma_0 \to \sigma_1 \to \sigma_2 \to \cdots \to \sigma_n$$

The program is started with the computer in an initial state σ_0, including the inputs to the program.

The program finishes with the computer in a final state σ_n, containing the output(s) of the program.

The state passes through a finite sequence of changes to get from σ_0 to σ_n; in general, each command may modify the state.

Functional programming

A functional program is simply an *expression*, and executing the program means *evaluating* the expression.

- There is no state, i.e. there are no variables.
- Therefore there is no assignment, since there's nothing to assign to.
- And there is no sequencing and no repetition, since one expression does not affect another.

But on the positive side:

- We can have recursive functions, giving something comparable to repetition.
- Functions can be used much more flexibly, e.g. we can have higher order functions.

Functional languages support this style of programming.

Example: the factorial

The factorial function can be written imperatively in C as follows:

```
int fact(int n)
{ int x = 1;
  while (n > 0)
   { x = x * n;
     n = n - 1;
   }
  return x;
}
```

whereas it would be expressed in ML as a recursive function:

```
let rec fact n =
  if n = 0 then 1
  else n * fact(n - 1);;
```

Why?

At first sight a language without variables, assignment and sequencing looks very impractical.

We will show in this course how a lot of interesting programming can be done in the functional style.

Imperative programming languages have arisen as an abstraction of the hardware, from machine code, through assemblers and macro assemblers, to FORTRAN and beyond.

Perhaps this is the wrong approach and we should approach the task from the human side. Maybe functional languages are better suited to people.

But what concrete reasons are there for preferring functional languages?

Merits of functional programming

By avoiding variables and assignments, we gain the following advantages:

- Clearer semantics. Programs correspond more directly to abstract mathematical objects.
- More freedom in implementation, e.g. parallelizability.

By the more flexible use of functions, we gain:

- Conciseness and elegance.
- Better parametrization and modularity of programs.
- Convenient ways of representing infinite data.

Denotational semantics

We can identify our ML factorial function with an abstract mathematical (partial) function $\mathbb{Z} \to \mathbb{Z}$:

$$[\![\mathrm{fact}]\!](n) = \begin{cases} n! & \text{if } n \geq 0 \\ \bot & \text{otherwise} \end{cases}$$

where $\bot$ denotes undefinedness, since for negative arguments, the program fails to terminate.

Once we have a state, this simple interpretation no longer works. Here is a C 'function' that doesn't correspond to any mathematical function:

```
int rand(void)
{ static int n = 0;
  return n = 2147001325 * n + 715136305;
}
```

This gives different results on successive calls!

Semantics of imperative programs

In order to give a corresponding semantics to imperative programs, we need to make the state explicit. For example we can model commands as:

- Partial functions $\Sigma \to \Sigma$ (Strachey)
- Relations on $\Sigma \times \Sigma$ (Hoare)
- Predicate transformers, i.e. total functions $(\Sigma \to \mathit{bool}) \to (\Sigma \to \mathit{bool})$ (Dijkstra)

If we allow the `goto` statement, even these are not enough, and we need a semantics based on *continuations* (Wadsworth, Morris).

All these methods are quite complicated.

With functional programs, we have a real chance of proving their correctness, or the correctness of certain transformations or optimizations.

Problems with functional programs

Functional programming is not without its deficiencies. Some things are harder to fit into a purely functional model, e.g.

- Input-output
- Interactive or continuously running programs (e.g. editors, process controllers).

However, in many ways, infinite data structures can be used to accommodate these things.

Functional languages also correspond less closely to current hardware, so they can be less efficient, and it can be hard to reason about their time and space usage.

ML is not a pure functional language, so you can use variables and assignments if required. However most of our work is in the pure functional subset.

Overview of the course (1)

We start with the theoretical underpinning, λ-calculus, and move up to the practical business of programming in ML.

Roughly $\frac{1}{3}$ of the course is devoted to the theoretical parts, and $\frac{2}{3}$ to the practical parts.

The theoretical parts are:

1. Introduction and overview (this lecture)
2. λ-calculus as a formal system
3. λ-calculus as a programming language
4. Types

Overview of the course (2)

The practical or 'applied' parts of the course are:

1. Introduction to ML
2. More about polymorphism
3. Recursive types
4. Proofs about programs
5. Further ML
6. Other styles of functional programming
7. ML examples (1)
8. ML examples (2)

Lambda notation

Lambda notation is a way of denoting functions, invented by Alonzo Church in the 1930s. We write:

$$\lambda x.\ E[x]$$

to denote 'the function of x that yields $E[x]$'. Here $E[x]$ is any expression which may or may not contain x.

For example $\lambda x.\ x$ is the identity function, which simply returns its argument, while $\lambda x.\ x^2$ is the squaring function.

The letter λ is arbitrary, a historical accident. Originally Church used $\hat{x}.\ E[x]$, and a series of typesetting errors transformed it into the present form. One also sees $x \mapsto E[x]$ and $[x]\ E[x]$.

Why?

In informal mathematics, when talking about a function, one normally gives it a name: 'define $f(x) = E[x]$ ... then $\cdots f \cdots$'.

Similarly, most programming languages will only let you define functions if you are prepared to give them a name.

This approach is rather inconsistent. It means we are not treating functions on a par with other mathematical objects. We don't say:

> Define x and y by $x = 2$ and $y = 4$ respectively. Then $xx = y$

Lambda notation helps to put functions on an equal footing. This is important in functional programming.

Benefits of λ-notation

The λ-notation can be useful too in order to make mathematics clearer.

When we discuss $x + xy$, we are often vague about whether we mean the value for a particular x, or a function of x (or y ...). Using λ-notation this can be made explicit.

In fact, using just:

- Variables, e.g. x, y.
- Constants, e.g. 3, $true$, $+$.
- Applications of functions to arguments, e.g. $f\ x$.
- Lambda abstractions of expressions over variables, e.g. $\lambda x.\ x + y$

we reach a general 'abstract syntax' for mathematics.

Currying

We can have functions of more than one argument by using, for example $\lambda x.\ \lambda y.\ x + y$.

We can think of this as a function $\mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$. When given one argument, say 2, it returns a function $\mathbb{Z} \to \mathbb{Z}$ that adds 2 to its argument, and this takes the second argument. This device is known as *currying*, after Haskell Curry.

We adopt the conventions that $\lambda x\ y.\ E[x, y]$ means $\lambda x.\ \lambda y.\ E[x, y]$, and that function application associates to the left, e.g. that $f\ x\ y$ means $(f(x))(y)$. This supports the use of currying, e.g.

$$(\lambda x\ y.\ x + y)\ 1\ 2 = (\lambda y.\ 1 + y)\ 2 = 1 + 2$$

Note that the brackets round function arguments are optional.

Variable binding

Many constructs in mathematics *bind* variables. The variable x in

$$\int_0^a 3x^2 \; dx$$

is bound, whereas a is free; likewise n is bound and k free in

$$\Sigma_{n=0}^{k} n^2$$

Free and bound variables can be quite complicated, as we shall see. Using λ-notation, all these variable binding constructs can be broken down so that the only binding operation is λ-abstraction.

Example of differentiation

For example, the everyday construct $\frac{d}{dx}x^2$ can be analyzed as follows:

$$D\ (\lambda x.\, \mathrm{EXP}\ x\ 2)\ x$$

where D is the (curried) differentiation operator $(\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R}$ giving the derivative of its first argument at the point indicated by its second argument, and EXP the exponentiation function.

So there are really two instances of x here, one bound, one free. The same occurs in

$$\int_0^x 2x\ dx$$

but here the everyday notation separates the two properly.

Russell's paradox

Originally, Church hoped to use λ-notation to give a foundation for mathematics, by introducing constants for logical operations, e.g. $\neg$ to mean 'not'. Unfortunately this turned out to be inconsistent, because defining $R = \lambda x.\ \neg\ (x\ x)$ we have:

$$R\ R = (\lambda x.\ \neg\ (x\ x))\ R = \neg\ (R\ R)$$

We know there is a correspondence between sets and their characteristic function, so if we think of $s\ x$ as meaning $x \in s$, this is simply the well known Russell paradox of the set of all sets that do not contain themselves:

$$R = \{x \mid x \not\in x\}$$