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
- Historical remarks
- Overview of the course

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. We can relate this to the imperative view by writing $\sigma_n = E[\sigma_0]$.

- 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:

```
fun 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 $(\Sigma \to bool) \to (\Sigma \to bool)$ (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.

Historical remarks

Some of the ideas behind functional programming go back a long way, e.g. to 'lambda calculus', a logical formalism due to Alonzo Church, invented in the 1930s before electronic computers.

The earliest real functional programming language was LISP, invented by McCarthy in the 50s. However this had a number of defects, which we will discuss later.

The modern trend really begins with ISWIM, invented by Peter Landin in the 1960s.

The ML family started with Robin Milner's theorem prover 'Edinburgh LCF' in the late 70s. The language we shall study is essentially (core) Standard ML, but there are other important dialects, notably CAML and Objective CAML.

Overview of the course (1)

- Practicalities of interacting with ML.
- Key functional concepts, e.g. evaluation strategy, higher order functions.
- Polymorphic types.
- Recursive functions and recursive structures.
- Hints for effective programming.
- Exceptions, references and other imperative features.
- Proving programs correct.

Overview of the course (2)

We want to show the power of ML, so we'll finish with more substantial examples that illustrate some of the possibilities:

- Symbolic differentiation
- Recursive descent parsing
- A Prolog interpreter
- A theorem prover

The code for these examples will be made available on Thor.