Functional programming languages as a software engineering tool

Simon L Peyton Jones, University College London

29th August 1986

1. Introduction

In 1968, Dijkstra's famous letter [Dijks68] suggested that the GOTO statement was a harmful feature of programming languages. In due course, this gave rise to a new class of structured programming languages, which lacked the GOTO statement, but which in compensation provided a set of control structures.

In 1977, Backus' Turing award lecture [Back78] suggested that the assignment statement was in fact the cause of the trouble, and the GOTO statement was entirely innocent! This view (which has much earlier origins; see for example Landin's paper [Land66]) has given rise to the class of functional programming languages, which lack the notion of assignment (and hence side-effects), but which in compensation support functions as first class citizens (that is, functions may be passed as arguments to functions, returned as results, stored in data structures and so on).

Since then a considerable amount of work has been done on the implementation of functional programming languages, but rather less has been written about their suitability as a software engineering tool. A programming language is, of course, only one of the tools in a software engineer's toolkit, but the language chosen has a far reaching effect (for good or ill) on other aspects of the program development process. It is the purpose of this paper to suggest that functional languages are an appropriate tool for supporting the activity of programming in the large, and to present a justification of this claim.

We begin with a discussion of what we are looking for in a programming language, followed by a short introduction to functional programming (Section 3). This is followed by a discussion of the major features of functional programming which support software engineering. In particular we discuss polymorphic typing (Section 4), higher order functions and lazy evaluation (Section 5), abstract data types and modules (Section 6), formal methods applicable to functional programs (Section 7), and parallel functional programming (Section 8). Finally in Section 9 we discuss some of the major shortcomings of the functional style.

2. What do we look for in a programming language?

Before we can evaluate the effectiveness of functional languages we must identify the characteristics which we desire in a programming language. As Abelson and Sussman remark [Abels85],
"A powerful programming language is more than just a means for instructing a computer to perform tasks - the language also serves as a framework within which we organise our ideas about processes".

This challenge of unambiguously expressing our ideas about processes is one of the major objectives of computer science.

It is hard to improve on John Locke's remarkable insight, written in 1690:

The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three:

(i) Combining several simple ideas into one compound one, and thus all complex ideas are made.

(ii) The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations.

(iii) The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.

John Locke, An Essay Concerning Human Understanding (1690).

The first and third of his insights apply directly to programming languages. To be specific, we look for the following characteristics in a programming language:

(i) It should be secure against accidental inconsistencies.

(ii) It should have powerful means of combination, by which compound expressions are built from simpler ones (Locke's first point). Examples include the control structures of structured programming, data structures and so on. Powerful combining forms lead to concise programs, so that we can hold the whole program in mind at once and "think big thoughts all at once". They also lead to modular programs, which are built up systematically out of smaller pieces, rather than being a single monolithic structure.

(iii) It should have powerful means of abstraction, by which compound objects can be named and manipulated as units (Locke's third point). Examples include function and procedure definition, modules, classes and so on. Powerful means of abstraction also support modular programming.

(iv) It should be amenable to formal methods of program specification, reasoning, transformation and analysis.

(v) With the advent of parallel machines it is highly desirable that the language should be able to express parallel computations.

(vi) It should be possible to construct an efficient implementation, since we want to actually run our programs. Furthermore it should be possible to reason about the performance and resource requirements of a particular program.

This list of characteristics gives the structure of the rest of this paper; we will discuss them in the order they are given above. First, however, we offer an introduction to functional programming, which may safely be omitted by the cognoscenti.

3. A brief introduction to functional programming

In this section we will give a very brief introduction to functional programming. Other introductions to functional programming are given by Turner [Turn82] and Darlington [Darl84]. For fuller treatments see Burge [Burge75], Henderson [Hend80], and Claeys et al [Glas84]. The best book about using the functional idiom as a means of writing good programs is undoubtedly Abelson and Sussman's excellent book [Abels85].