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 Glaser 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].

Functional languages are members of the class of **declarative languages**, that is, languages in which programs have a **declarative reading** which asserts properties of the functions defined by the program, as well as an **algorithmic reading** which describes how the functions are to be computed. Another common term for functional languages is **applicative languages.** By way of contrast with the declarative nature of functional languages we will refer to conventional procedural languages (like Pascal or C) as **imperative.**

For the sake of definiteness, examples given in this paper will be written in the language Miranda [Turn85]. However Miranda is by no means the only functional languages, and we summarise the main ones at the end of this section.

3.1 Definitions and scripts

A Miranda program consists of a set of definitions (the **script**) together with an expression to be evaluated. Executing the program consists of evaluating the expression and printing the result. For example, the empty script together with the expression:

```
3+4
```

is a valid Miranda program. When executed it will print the result:

```
7
```

Here is a possible Miranda script:

```
a = 4
b = 5
c = a-b
```

Each line is a separate definition, and the script gives values to the identifiers "a", "b" and "c". These definitions are closely analogous to the Pascal **const** declaration, which gives a value to an identifier, which remains constant throughout its scope. They are quite different to assignment statements, which change the value of an identifier. The identifiers are names for values, not names for boxes containing values.

With this script, the expression:

```
b-c
```

would evaluate to:

```
6
```

Functions may be defined in the script. For example the script:

```
average x y = (x+y)/2
```

defines "average" to be a function which takes the mean of its two arguments. The expression:

```
average 4 6
```

evaluates to:

```
5
```

Notice that function application is denoted by juxtaposition, so that we write "average 4 6" not "average(4,6)". Parentheses are used solely for grouping, rather than having two purposes as they do in Pascal.

Identifiers can also be bound to values locally, using a "where" clause:

```
f x = y+1 where y = 2*x
```

Functions can be defined locally in a similar manner.

Miranda is a strongly typed language, and the compiler deduces a type for each identifier (see Section 4). The type of a factorial function, for example, would be

```
        factorial :: num -> num
```

That is, factorial is a function which takes a number and returns a number. The type of average is

```
        average :: num -> (num -> num)
```

That is, "average" is a function which takes a number and returns a function, which takes a number and returns a number. Thus, we may form a **partial application** of "average"; for example, the expression

```
        (average 4)
```

is the function which averages its argument with 4. This way of treating a function of several arguments as a function of a single argument which returns a functional result is called **currying.**

3.2 Guards and pattern matching

Two other constructs aid in function definition, **guards** and **pattern matching.** An example of a function using a guard is the greatest common divisor function:

```
        gcd a b = gcd a (b-a),     a>b
                = gcd (a-b) b,     a<b
                = a,               a=b
```

This definition is reminiscent of a mathematician's syntax, and gives three alternative right hand sides for "gcd". The appropriate one is chosen depending on the value of the guard, which is a boolean expression following a comma on the right hand side. The last guard may be omitted to give an "otherwise" case.

An example of a function using pattern matching is:

```
        factorial 0 = 1
        factorial n = n * (factorial (n-1))
```

The appropriate equation is chosen depending on the value of the argument, by matching it against the pattern on the left hand side. In this case we could use a guard instead, but pattern matching comes into its own when data structures are introduced.

3.3 Data constructors and types

The major built in data structure is the **list.** For example, the expression:

```
        [4,3,9]
```

denotes a list of three numbers. Lists may be built with an infix list construction operator ":". Thus the expression:

```
        4:[3,9]
```

denotes a list whose first element is 4 and the rest of which is [3,9]; it is the same list as [4,3,9]. The primary way in which lists are taken apart is by using pattern matching. For example, the function which takes the length of a list could be defined like this:

```
        length [] = 0
        length (x:xs) = 1 + (length xs)
```

(Note: the parentheses around "length xs" are not required.) The empty list is denoted [], so the first equation says that the length of the empty list is 0. The second equation says that the length of a list whose first element is "x" is 1 plus the length of the rest of the list.

Miranda allows the programmer to declare new types. For example, the script

```
        tree ::= EMPTY | NODE tree num tree
```

defines a new type "tree", and says that an object of type "tree" is

- **either** EMPTY,
- **or** a NODE with three fields, of type "tree", "num" and "tree" respectively.

Objects of type "tree" can be built using the constructors EMPTY and NODE; for example

```
        EMPTY

        NODE EMPTY 3 EMPTY

        NODE (NODE EMPTY 4 EMPTY) 5 EMPTY
```

are all trees. Trees can be taken apart using pattern matching. For example, the function "sum" adds up all the numbers in the tree:

```
        sumtree EMPTY         = 0
        sumtree (NODE t1 n t2) = sumtree t1 + n + sumtree t2
```

3.4 Other functional languages

The language ML, originally designed as part of the Edinburgh LCF project [Gord79], is now being significantly redesigned to take account of the experience gained with ML and other functional programming languages [Miln85]; the new language will be called Standard ML. A version of ML with lazy semantics has been implemented by Johnsson and Augustsson at the Chalmers University of Technology [Augus84].

Hope, designed by Burstall, MacQueen and Sannella [Burst81] at Edinburgh, is the language which Darlington's team at Imperial College are using for the ALICE project. It is a large language (for instance, it supports user-defined operators which may be prefix, infix or distfix).

Turner has designed a series of languages, Sasl [Turn76], KRC [Turn82] [Hamm84], and now Miranda.

Ponder is a language designed by Fairbairn with an interesting generalisation of the ML type system [Fairb82], while Cardelli's Amber language generalises ML's types in a different way [Card85].

SISAL, a functional language originating from the dataflow stable and targeted particularly at numerical computation, is in the late stages of development by a partnership between the University of Manchester, MIT, DEC and Lawrence Livermore National Laboratory [Univ85].

Lispkit, designed by Henderson [Hend80], is a functional language which shares the syntax and simplicity of Lisp, and is explicitly designed for easy portability.

Despite this plethora of languages, there are many issues where a consensus is emerging, and it is this consensus that we attempt to present here.

4. Polymorphic typing

Strong typing has been widely accepted in conventional programming practice as a technique which catches a large class of programming errors at the compilation stage. Unfortunately, strong typing sometimes leads to tiresome bureaucracy, and this problem becomes particularly serious in a functional programming style.

Consider, for example, the "length" function defined earlier. The function will work equally well on a **list of numbers** and on a **list of characters,** but a conventional strong typing system will force the programmer to write separate functions for these two data types.

The ability to write such generic functions is so important to functional programming that early functional programming languages were typeless. Fortunately, however, based on the work of Hindley [Hindl69], Milner developed the concept of **polymorphic typing** [Miln78] [Damas82], which generalises the notion of type to allow types such as **list of *,** meaning "a list of objects of any type". Now the length function has a well behaved type:

```
length :: [*] -> num
```

(Miranda uses "[*]" to mean "list of *".) Likewise, a function which reverses a list has type:

```
reverse :: [*] -> [*]
```

Milner also showed that the type of an expression can be **inferred from the source program by the compiler,** rather than explicitly declared by the programmer. This is the approach taken in ML, the first language to include polymorphic typing; Cardelli describes the method [Card85a].

Miranda also infers its types, but allows optional type declarations; if they are present, the compiler will check them [Turn85a].

Polymorphic typing thus gives us the best of both worlds - we can have complete type checking of fully generic programs, without having to write any explicit type declarations.

Type inference turns out to be quite a delicate balance between the expressiveness of the language and the computability of the type checking algorithm. For example, Hope also includes a polymorphic typing system, but it also allows **overloading** of operators, and this turns out to require explicit type declarations.

Even more general polymorphic type systems are exemplified by the languages Ponder [Fairb85] and Amber [Card85], both of which also require explicit type declarations.

Polymorphism is not restricted to functional languages, and it is applicable to both conventional imperative languages [Harl84] and logic languages [Mycr84]. However, functional languages seem to have provided a particularly hospitable framework for polymorphic typing systems.

The best overview of the world of types and polymorphism is given in a recent paper by Cardelli and Wegner [Card85b].

5. Means of combination

In common with imperative languages, functional languages provide many ways of "gluing together" simple expressions to form more complex ones. Powerful glue leads to

(i) **Concise programs.** It is well established that textually short programs can be written faster and debugged more quickly than long programs (see for example Chapter 8 of Brooks [Brook82]). Concise programs allow us to "think big thoughts in one go". The value placed on conciseness can be seen by the fact that "very high level programming language" means "a language in which we can write big programs with few symbols".

This approach is not without its dangers, however (vide APL).

(ii) **Modular programs.** For a software engineer modularity is one of the key properties of a language. A modular program allows the programmer to restrict his attention to one part of the program to the exclusion of the rest. The availability of powerful glue is one of the prerequisites to modular programming, since the modules need to be "stuck together".

Some of the combining forms of functional languages are very similar to those of any other language; for example

- Function application.

- Local definitions using "where".

- Data constructors (often user-definable).

In addition, functional languages provide two forms of glue which are almost unique, namely **higher order functions** and **lazy evaluation.** These two features are explored in detail by Hughes [Hugh84]. A more recent innovation is the possibility of incorporating unification in functional language, a concept which we explain below.

We will discuss these ideas in turn, but first we mention two notational devices which have proved to be highly successful glue.

5.1 Pattern matching

The use of pattern matching to perform case analysis implicitly rather than using explicit conditionals has gained wide acceptance. All recently designed functional languages incorporate pattern matching in roughly the form outlined above.

The use of pattern matching leads to a very clear program structure, in which a function taking an argument of type "thing" will normally contain an equation for each possible "thing". Thus the function "sumtree" defined above contained one case for each kind of tree.

5.2 List comprehensions

Consider this program, which computes the infinite list of prime numbers:

```
        primes = sieve [2..]
        sieve (p:x) = p : sieve [ n | n <- x; n rem p > 0 ]
```

where [2..] denotes the infinite list of integers beginning with 2, "<-" means "drawn from", and "rem" is the remainder operator. The expression after "p : sieve" is called a **list comprehension,** and can now be read as "the list of all n, drawn from (the list) x, such that the remainder when n is divided by p is greater than zero".

This notation for describing a list, inspired by Zermelo-Frankel set notation, was first introduced in a functional language by Burstall and Darlington in NPL [Burst77], and taken up by KRC, Hope and Miranda. It is a declarative construct, and fits very gracefully into a functional context. There seems to be no doubt that it makes functional programs significantly more concise, although its acceptance is not yet as widespread as pattern matching.

5.3 Higher order functions

In functional languages, a function is a "first class citizen". It may be treated as an object in the same way as any other data object, passed to another function as an argument, returned as a result, stored in a data structure, and so on. A higher order function is a function which takes a function as an argument and/or returns a function as a result.

5.3.1 An example

Borrowing an example from Hughes' paper, consider the function which adds up the elements of a list of numbers:

```
        sumlist []     = 0
        sumlist (x:xs) = x + sumlist xs
```

The only parts of this function definition which are specific to adding up the elements of a list (rather than, say, multiplying them together) are the "0" of the first equation and the "+" of the second. Suppose we parameterised the definition of the "sumlist" function on these two values, giving a new function "fold", where we intend that

```
        newsumlist = fold (+) 0
```

(Note: we have to enclose the "+" in brackets before passing it to "fold" to avoid the parser interpreting "fold + 0" as the sum of "fold" and 0! This is just another example of using brackets for grouping.) The definition says that "newsumlist" is a function obtained by applying the general function "fold" to "+" and 0. We can obtain the definition of "fold" by a simple generalisation of the definition of "sumlist":

```
fold binop unit []     = unit
fold binop unit (x:xs) = binop x (fold binop unit xs)
```

(Note: we have to use prefix notation for "binop" because only built in operators like "+" are parsed as infix.) Notice that while "fold" is a function expecting three arguments, we can **partially apply** it to only two arguments in the definition of "newsumlist" (see Section 3.1). Having defined "fold" in this way, we can now define numerous other useful functions:

```
productlist = fold (*) 1
alltrue = fold (&) true
anytrue = fold (|) false
```

"productlist" multiplies the elements of a list of numbers together, "alltrue" and's together a list of booleans, while "anytrue" or's them together.

Another way to understand what "fold (+) 0" does is to observe that it takes a list such as:

```
3 : 5 : 34 : []
```

(":" associates to the right), and returns the expression:

```
3 + 5 + 34 + 0
```

In other words, the ":" has been replaced by "+" and the "[]" has been replaced by 0. This can also be seen by referring to the definition of "fold". We can now see that:

```
fold (:) [] xs
```

will simply return a copy of the list "xs", since we replace ":" with ":" and "[]" with "[]". Similarly, if "ys" is a list, then

```
fold (:) ys xs
```

will return a copy of the list "xs", except that the "[]" at the end will be replaced with the list "ys". Thus

```
append xs ys = fold (:) ys xs
```

defines the function "append" which concatenates two lists. Finally, for the stout-hearted, we can even use "fold" to define "map", where "map f xs" applies the function "f" to each element of the list "xs":

```
map f = fold ((:) . f)
```

where the infix "." denotes function composition, so that "(f . g) x" evaluates to "f (g x)".

5.3.2 Conclusions

The function "fold" is a higher order function, since it takes a function as an argument. The ability to define higher order functions allowed us to generalise our original definition of "sumlist", so that instead of a monolithic definition of "sumlist" we were able to build it up out of the modules "+" and 0 using "fold" as glue. Of course this is a rather small example, but we were nevertheless able to derive quite a rich variety of functions by using "fold" as a means of combination of other simple functions.

Higher order functions are therefore rather like the control structures of conventional languages. They combine simpler functions together to form more complex ones. Unlike conventional languages, however, which have a fixed set of control structures, a functional language allows new higher order functions to be defined at will.

In summary, it is as though the language allowed new control structures to be defined whenever desired, to precisely fit the application.

5.4 Lazy evaluation

One mechanism for gluing programs together which the Unix* operating system provides is the **pipe,** which allows the output of one program to be fed directly into another. For example, the command:

```
        tbl paper | eqn | nroff
```

feeds the file "paper" into the "tbl" program, sending the output directly into the input of the program "eqn", whose output is fed directly to the program "nroff". Unless the file "paper" very small then, rather than completely process "paper" with "tbl" and writing the output to a temporary file before invoking "eqn", Unix will instead run "tbl", "eqn" and "nroff" all at once, with moderate sized buffers in between each. This avoids the need for large temporary files, and makes the first output appear as early as possible.

The popularity of Unix may partly be attributed to the ease with which programs can be glued together in this way. Some functional languages provide a rather similar sort of glue in a much more thorough-going way, through a technique called **lazy evaluation.** Suppose the result of a function "f" is a large list, which is passed to another function "g":

```
        g (f x)
```

Then we would prefer that the list produced by "f" is only generated as "g" uses it up, and this is just what lazy evaluation achieves. In effect "f" and "g" run as coroutines; "g" runs until it needs the next item of the list, whereupon "f" is awakened. It runs until it has produces the next item, and then goes back to sleep to allow "g" to continue. We can think of "g" **demanding** the next list item from "f", and this gives rise to the term **demand driven evaluation.**

In a lazy functional language all data structures are only evaluated on demand. Just as in the case of Unix, lazy evaluation provides powerful glue for combining programs together. Unlike Unix, however, which requires a special construct to express this combining form, simple function composition serves to combine lazy functions together.

Lazy evaluation has a far reaching effect on program style. Consider, for example, a function which computes square roots by iterative approximation. The formation of successive approximations and the termination condition are intimately interwoven in a imperative language. In a lazy functional language we can separate the square root function into two functions, one of which generates the (infinite) list of all approximations to the square root, and the second of which eats up this list until it finds a sufficiently good approximation. This is a more modular approach, since we can now replace the first function with a superior method of generating approximations without affecting the second (and vice versa).

Lazy evaluation has allowed us to modularise the square root function into two independent sub-functions, by taking the termination condition out of the loop body. This is a typical of the way in which lazy evaluation aids modularity; the program is factored into a generator function which constructs a number of possible solutions (the approximations in the square root case) and a selector function which examines them in turn and chooses the appropriate one. A more sophisticated numerical example is given by an arbitrary precision arithmetic package, where the precision of the answer can be arbitrarily increased in a demand driven manner [Peyt83].

As another example, consider a chess-playing program which works by exploring the tree of possible moves. We could write the function which evaluates the value of a position "p" like this:

```
        minimax (prune (generate_moves p))
```

The function "generate_moves" generates the (possibly infinite) tree of moves starting at position "p", while "prune" prunes the tree in some suitable way to make the tree small enough to apply "minimax" which performs the well known mini-max algorithm. Lazy evaluation ensures that the tree will only be generated as it is used. The program is modular, so that, for example, we could change to using the alpha-beta heuristic instead of mini-max without changing "prune" or "generate_moves". In a imperative language these three functions have to be inextricably tied up together in a single monolithic structure.

5.5 Adding unification to functional languages

Logic languages such as Prolog [Clock81] are also free from side effects and hence declarative [Kowal79] (so long as we eschew Prolog's dynamic ASSERT, which causes a side effect). However, the relationship between functional and logic languages is still extremely unclear.

Logic languages are based on the extremely powerful **unification** operation [Robin71] (a sort of two-way pattern match, by contrast with the one-way pattern matching of functional languages), which, among other things supports **backtracking** and, in principle, allows programs to be run **backwards** as well as forwards (though running the program backwards is normally too inefficient to be feasible). Logic languages are only **first order,** and lack the power of higher order programming.

Unification is, however, clearly an extremely powerful glue for building programs, and it is natural to ask whether it can be incorporated into the framework of functional languages. In 1983 Darlington suggested a method of extending functional languages to include unification [Darl83], by extending the **relative** list comprehension described above to **absolute** list comprehension. Then, instead of writing

```
[ x | x<-L; p x ]
```

meaning "the set of x drawn from L such that (p x)", we might write

```
[ x | p x ]
```

meaning "the set of x drawn from S such that (p x), where S is the set of all values to which p is applicable" (the applicability of p is determined by the type-checker). As an example, consider

```
split s = [ [s1, s2] | append [s1, s2] = s ]
```

where append is defined by

```
append [[], k2] = k2                        (1)
append [(x:k1), k2] = x : append [k1, k2]   (2)
```

Now suppose s = [1, 2] (so that s = 1:[2]). We have

```
split [1,2] = [ [s1, s2] | append[s1, s2] = 1:[2] ]
```

and now we must perform a unification with the definition of append, to get (as one possibility, by unifying with (2))

```
split [1,2] = [  [1:k1, k2] | append [k1, k2] = [2] ]
```

Taking it one step further, unifying with (1)

```
split [1,2] = [ [[1], [2]] | ]
```

which is one valid result. The others will be generated from choosing different equations with which to unify.

Darlington shows that the technique will work for higher order functions as well, but there is a limit here, since unification is known to be undecidable at second order and above [Huet73] [Huet75] [Goldf81] [Fages83].

There has been a recent flurry of papers on the subject (see for example [Robin84] [Reddy85] [Linds85] [Darl85]), and there seems no doubt that the idea is here to stay.

6. Means of abstraction

Suppose a manufacturer wishes to design a new car. First of all specifications will be established for the various components (brakes, engine, audio hifi system, and so on), and then the design of these components will be assigned to different individuals. It is taken for granted that the designer of the braking system does not need to know anything about the design of the hifi system, because these subsystems do not interact.

Modularity of this kind is perhaps our primary tool for managing complexity. Unfortunately, essential though it is, modularity cannot be taken for granted in program design. It is all too easy for unforeseen interactions between program modules to cause the system to fail. The programming language should therefore provide mechanisms to **control the interactions** between different parts. All languages do this to some extent, but functional languages are particular strong in this respect.

6.1 Abstraction through function definition

The simplest form of abstraction, which is supported by any programming language worthy of mention, is functional (or imperative) abstraction. The function:

```
append xs ys = fold (:) ys xs
```

abstracts the idea of concatenating two lists away from the method of concatenation. We are free to change the implementation of the function so long as we do not change the abstraction it implements. At the most trivial level we are free to change the name of the formal parameters, but we could make a more significant change, by writing an explicit definition of append:

```
append [] ys     = ys
append (x:xs) ys = x : append xs ys
```

Functional languages enforce a very clearly defined interface between a function and its caller. The result of a function is governed solely by its arguments, and if the same function is applied to the same arguments on a different occasion the same result will be produced. This can be guaranteed because the absence of assignment means that no side effects can take place between the two calls.

In a large Pascal program it generally proves impossible to communicate between functions entirely by parameter passing, so some communication is mediated by shared global variables which are modified as a side effect of calling the function (or procedure). This allows a potentially undisciplined interaction between functions, and it is often an unexpected side effect which causes an imperative program to fail. Modularity is lost, because the insides of one module interact in an intimate way with the insides of another module. The alternative of restricting imperative programs to eschew all side effects (except those local to a single function) proves too restrictive, because imperative languages have insufficiently powerful combining forms and storage allocation mechanisms.

Functions in a functional language implement the mathematical abstraction of a function, whereas functions in an imperative language implement something altogether more complicated and harder to understand.

6.2 Abstract data types and modules

Many modern imperative languages, such as Ada*, contain mechanisms to support the development of large systems. Examples of such mechanisms include modules, packages or classes. They constitute the next level of our attempt to manage complexity, by breaking the system into components of manageable size which can then be combined together in a secure way to form a complete system. Typically they support abstract data types, multi-tasking, and generic shareable program components. They all implement information-hiding in some form, to enforce limitations on the forms of interaction between components.

Recent work strongly suggests that functional languages form a hospitable environment for such efforts. ML successfully supported abstract data types from its inception [Gord79], using a mechanism supported almost entirely by the type checker, imposing no run time overhead. Miranda provides an even simpler mechanism with similar properties [Turn85a]. As an example of the power of these mechanisms, the LCF system written in ML has an abstract type "theorem". Objects of this type are guaranteed to be theorems of the logic concerned because such objects can only be produced by combining together other theorems and axioms using the laws of deduction provided by the abstract type.

Attention is now turning to more sophisticated constructs. In imperative languages the mechanisms for constructing generic packages, for example, tend to be rather complex and ad hoc. Recent work in typing systems for functional languages suggests that it will be possible to make modules "first class citizens", capable of being typed, named, parameterised and processed. Just as treating functions as "first class citizens" opened the way to higher order functions, and user-defined combining forms, so this treatment of modules will allow the programmer to instantiate and combine modules using combining forms defined for the purpose.

This work is at an experimental stage at present, but the signs look encouraging. The most accessible points at which to begin further reading are Burstall and Lampson [Burst84] [Burst84a], MacQueen [MacQ85], and Cardelli and Wegner [Card85b].

7. Formal methods

The use of formal methods to reason about programs is the very stuff of software engineering. Formal methods are a key tool in the transformation of programming from black art into an engineering discipline. Our aim is to generate efficient and correct programs from formal specifications, and to do so using machine-checkable techniques.

In order for such formal methods to be practicable it is essential that the programming language should have simple semantics, for otherwise the tasks of program transformation and verification become extremely burdensome. Lacking the assignment statement, functional languages do indeed have extremely simple semantics by comparison with almost any "real" imperative language. In this section we will discuss two areas where mechanised formal methods have been successfully brought to bear.

7.1 Program transformation

The primary goal of program transformation methods is to transform a comprehensible though inefficient **specification** into an efficient but possibly obscure **program** [Darl81]. Closely related goals are the design of efficient data representations [Darl80], and proving properties of programs [Turn83].

This is an expanding field, and a rich source of references may be found in [Parts83]. The most commonly encountered framework for program transformation is the Fold/Unfold system of Burstall and Darlington [Burst77a], and we give a short example here to give the flavour of the subject, taken from [Darl81]. Assume that the following functions are defined (we use curly brackets on the left to identify equations):

```
{lg.1}  length [] = 0
{lg.2}  length (x:s) = 1 + length s

{ap.1}  append [] s2 = s2
{ap.2}  append (x:s1) s2 = x : append s1 s2
```

and that we want to define a function to append two lists together and find the length of the resulting list. We can write it thus:

```
{lo2}   lengthof2 s1 s2 = length (append s1 s2)
```

but while this definition is adequate and clear, it is somewhat inefficient, and we will derive a more efficient version. Instantiating {lo2} with s1=[] gives

```
        lengthof2 [] s2
        = length (append [] s2)      instantiating by {lo2}
{lo2.1} = length s2                  unfolding by {ap.1}
```

Now instantiating {lo2} with s1=(x:s1) gives

```
        lengthof2 (x:s1) s2
        = length (append (x:s1) s2)  instantiating by {lo2}
        = length (x : append s1 s2)  unfolding by {ap.2}
        = 1 + length (append s1 s2)  unfolding by {lg.2}
{lo2.2} = 1 + lengthof2 s1 s2        folding by {lo2}
```

Finally, we see that {lo2.1} and {lo2.2} constitute a complete new definition of lengthof2, which is "more efficient" than the first version.

7.1.1 Transformation Systems

The transformation given above is completely formal, and amenable to machine **checking.** However, it is not very clear what "more efficient" means, so machine **generation** of proofs is problematic.

While in the example above the sequence of transformations applied is fairly simple, larger examples become considerably more complex, and there is no doubt that, even with machine assistance, it would become impossible to conduct large proofs one step at a time. There are two basic strategies for addressing this problem, which Partsch and Steinbruggen [Parts83] call the **generative set approach** and the **catalog approach.**

The generative set approach uses a basic set of elementary transformations (fold, unfold, instantiate and so on), together with some kind of **meta language** in which to express patterns of transformation, so that the proof can be conducted at a higher level of abstraction (cf [Gord79], where ML is used as a proof meta language, but in a different proof system). The proof then becomes a program in the meta language. Darlington is using Hope itself as a meta language for conducting proofs about Hope programs [Darl81], and this is also the approach taken by the large and impressive CIP project [Moll84].

The catalog approach works by proving, in advance, a sufficiently large set of powerful theorems about the (higher order) functions involved in the program. These theorems are, in effect, "high level transformations" and therefore much more tractable than the elementary transformations of the generative approach. The catalog approach works well when there is a very restricted set of higher order functions, such as in the language FP, where it leads to an "algebra of programs", as described by Backus [Back78]; further references are [Wadl81] [Islam81] [Kieb81].

7.1.2 Program transformation in compilers

So far we have discussed program transformation in the context of supporting the program development process. This is an ambitious and long term goal. Program transformation has also been applied with considerable success in the more limited context of program compilation, and large sections of almost any compiler for a functional language can be regarded as program transformations. They are thus amenable to formal proofs of correctness, and are readily communicable to other compiler writers.

Examples abound, including:

- The combinator abstraction algorithm of Turner [Turn79].
- The compilation of pattern matching [Augus85] and list comprehension constructs [Wadl86].
- The supercombinator (or lambda lifting) algorithm of Hughes [Hugh82] [Johns85].
- The optimisation of functional programs by partial evaluation [Hudak84] [Jones85].

7.2 Abstract interpretation

Abstract interpretation is a technique for deducing information about a program from its text, by executing an abstract version of the program. An appropriate abstraction is chosen according to what information is wanted.

As an example, suppose we wanted to know the sign of

```
34298 * (-3981) * 300993
```

The hard way to find the sign of this number is to perform the two multiplications in full and look at the sign of the result. A simpler method is to perform a more abstract calculation:

```
+ <*> - <*> +
```

We replace each number with an abstract representation (its sign), and replace the multiplication operator with an abstract version "<*>", which implements the well-worn "rule of signs".

```
+ <*> + = +
- <*> + = -
+ <*> - = -
- <*> - = +
```

Now it is easy to compute the answer "-", which tells us that the result of the original calculation would have been negative. The abstraction we chose suited the question we wanted to ask, and the execution in the abstract domain was much less tedious than the original.

Abstract interpretation is an area of much recent research, since the technique seems to have potentially wide applicability. It is in one area, however, that of **strictness analysis** that most progress has been made.

7.2.1 Strictness analysis

Functional languages are promoted as a natural vehicle for programming parallel machines, so it is legitimate to ask where such parallelism comes from.

It turns out that it all comes from evaluating the argument(s) of a function in parallel (with each other and with the function itself). When, then, can we evaluate the argument of function in parallel? It can certainly never render our program incorrect (since it can have no side effects), which is reassuring, but it may not be a good way to use the machine's resources.

One approach is to drop the problem in the programmer's lap, and require annotations in the program to indicate where parallelism is desired. However, we might prefer to find some automatic technique for discovering when it is **safe** to evaluate an argument in parallel. In this context, it is "safe" to evaluate an argument to a function in parallel if we know that the function will need the value of its argument, and hence must eventually evaluate it. For instance, consider the function

```
f x y = if (expensive y) then x+1 else x+2
```

where "expensive" is some expensive function returning a boolean. We know that x's value will be needed, and we could evaluate it in parallel. This kind of analysis is sometimes called **strictness analysis** (a function is **strict** if it always evaluates its argument).

Mycroft's thesis [Mycr81] describes a method for performing this strictness analysis using abstract interpretation. Mycroft suggests reducing the data domain to "undefined" and "defined". Then if, by executing the program in this abstract domain, we find that

```
f "undefined" = "defined"
```

then f does not evaluate its argument, so we must not evaluate the argument in parallel (because it may not terminate). If, on the other hand

```
f "undefined" = "undefined"
```

then f does evaluate its argument. In this case, it is safe to evaluate the argument in parallel. Matters are in fact more complicated than this, but only slightly, and Mycroft's work comes complete with a sound theoretical basis. However, his work only applies to flat domains (ie no data structures) and first order functions.

Recent work has, however, extended the method to higher order functions [Burn85] and data structures [Hugh85] [Wadl85].

8. Parallel computation

One of the most attractive features of functional programming languages is that they are not inherently sequential, as conventional imperative languages are. As Section 3 implied, a functional program is executed by evaluating an expression, and several sub-expressions may be evaluated concurrently. For example, in the expression

```
(3+4) * (sqrt 24)
```

the addition can take place concurrently with the square root. Thus the hope offered by functional languages is that

> parallel execution of functional programs, through concurrent evaluation of sub-expressions, may be possible without adding any new language constructs or detailed program tuning.

If taken without qualification this statement is rather misleading, since it seems to promise "parallelism without tears", but in fact co-operation never comes for free. We can, however, take the statement as highlighting an opportunity. If the challenge of parallel machines is a mountain then concurrent graph reduction offers us a promising looking path between the steep cliffs.

The idea of concurrent execution of programs without adding new lanaguage constructs is not new. The FORTRAN compiler for the Cray-1 is designed to spot vectorisable sections of programs written in (almost) ordinary FORTRAN. However, the effective use of the Cray relies on the programmer writing his program in such a way that the compiler can spot that it is vectorisable. Furthermore, high performance is an extremely delicate property of programs, and easily destroyed by seemingly innocuous modifications.

We hope that in the case of functional languages the parallelism is less delicate and more general, so that the programmer's task is made easier. First, therefore, we will discuss the task of writing parallel functional programs.

8.1 Writing parallel functional programs

It is tempting to believe that an arbitrary functional program would run much faster on a parallel graph reduction machine. This comforting belief is quite erroneous [Clack85]. Many functional programs are essentially sequential (that is, at any moment there are few redexes in the graph). For example, an insertion sort program cannot insert the next element into the result until the previous insertion has completed (or at least partly completed). It is simply unreasonable to expect any old functional program to run fast on a parallel machine.

In order to achieve good parallel performance the program must contain **algorithmic parallelism**. That is, the algorithm must contain gross inherent parallelism. The most obvious sort of algorithmic parallelism is given by **divide and conquer** algorithms, which divide the task at hand into two or more independent sub-tasks, solve these independently, and then combine the results to solve the original task. A standard example of such an algorithm is quicksort, which splits the set to be sorted into two subsets which can be sorted independently. Other examples include any kind of search algorithm (which covers many Artificial Intelligence applications) and large numerical computations. Experiments confirm that substantial parallelism is obtainable [Tighe85] [Clack85].

It is therefore still the program designer's responsibility to create an algorithm which will divide the task at hand into reasonably independent sub-tasks. It is unreasonable to expect the machine to do this automatically, since it may involve major algorithmic changes (such as changing insertion sort to quicksort).

8.2 Writing parallel programs is easier in a functional language

Why not program in a conventional language which supports multiple tasks, such as Ada? There are a number of ways in which writing a parallel program in a functional language is superior to this:

(i) In conventional languages the division of the problem into separate tasks is static and fixed. A task is conceived as a relatively large unit, and tasks generally cannot be created and deleted dynamically. There will be relatively few tasks, and the programmer must clearly identify all of them in his design.

In a functional language the parallelism can be dynamic, and there is no static division of the problem into tasks. Instead, the programmer designs an algorithm whose inherent parallelism will enable concurrent reduction to take place at different places in the graph. The "grain" of parallelism is therefore smaller and more dynamically adaptable as the computation proceeds.

(ii) In conventional languages the tasks communicate with each other by sending messages or making specially protected subroutine calls to each other. The programmer has to design synchronisation and communication protocols between tasks so that they cooperate correctly and achieve mutual exclusion where necessary. It is up to the programmer to ensure that these communication protocols are correct, and failure to do so can result in a transient malfunction of the program.

In a functional program the synchronisation between different reductions is mediated entirely by the shared graph. A reduction is made known to the graph by the indivisible operation of overwriting the root of the redex with the result of the reduction, and no other synchronisation is necessary (though see the next section for efficiency considerations).

(iii) The tasking structure of conventional languages adds a layer of considerable complexity to the programmer's model of what is going on. If it is hard to reason about a sequential program, it is even harder to reason about a multi-tasking program, because the programmer has to bear in mind all the possible time orderings in which execution might take place. The behaviour of the program should be independent of the scheduling of the tasks, but it is up to the programmer to ensure that this is the case.

There are no extra language constructs required to write parallel functional programs. The result of the program is guaranteed to be independent of the way in which reductions are scheduled, though this scheduling may have a strong impact on efficiency. Thus it is no harder to reason about a parallel functional program than a sequential one.

To summarise, the programmer does not have to design a static task structure, guarantee mutual exclusion and synchronisation, or establish communication protocols between tasks. This frees him for the creative activity of designing a parallel algorithm.

9. Shortcomings of functional programming

It would be misleading to present functional languages as the panacea for all ills, and this section summarises their principal shortcomings.

Until recently the raw efficiency of functional languages has been a problem, but recent implementations now approach the speed of compiled Pascal [Augus84a] [Fairb85a], so the compiler technology seems now available to alleviate this problem.

9.1 Reasoning about performance

The goal of a programmer is to write programs that are

(i) (absolutely) correct; that is, they should meet their specification.

(ii) (reasonably) efficient; that is, they should consume as few machine resources as possible.

In order to achieve these goals the programmer has to reason about

(i) the meaning of his program, to assure himself that it has the same meaning as the specification.

(ii) the resource consumption of his program, to assure himself that it will consume only reasonable resources.

We have argued that functional languages are much more tractable than imperative ones when reasoning about correctness. On the other hand, it is normally fairly straightforward to reason about the memory space and CPU cycles consumed by a imperative program, because the programmer has an accurate mental model of how execution takes place.

Unfortunately, **a major weakness of functional languages is the difficulty of reasoning about their space and time behaviour** , especially the former. In particular, a functional program will often have much worse space-time behaviour than the programmer might expect. The trouble is that it is difficult for the programmer to maintain an accurate mental model of how execution takes place, and whatever understanding he may develop is normally wrecked by the implicit coroutine behaviour of lazy evaluation. Furthermore, seemingly innocuous (and correctness preserving) changes to the program may have truly dramatic effects on its run-time behaviour. For example, the space required by a three line program to compute the set of all sublists of a list of length N can change from O(N) to O(N!) as a result of a minor program change (common subexpression elimination).

Meira takes the efficiency of functional programs as the main subject of his thesis [Meira85], and chapter 6 of Stoye's thesis [Stoye85] gives a good summary of the various guises in which this problem manifests itself.

The unpredictable performance of functional programs is possibly the most serious obstacle to their widespread adoption in production environments.

9.2 Input/output and nondeterminism

The absence of side effects in functional programs raises the question of how input and output is to be performed. The solution is to regard the program as a function mapping the input data (from the user's keyboard or a file) to the output data (which the operating system can send to the user's screen or a file).

```
                   ----------------------
Keyboard ----> | Functional program | ----> Screen
                   ----------------------
```

Lazy evaluation is a big help in this context, because it means that the (potentially infinite) data structure representing the input need only be constructed as the program consumes it, which allows interactive programs to be supported.

Nevertheless, it is somewhat less flexible input/output paradigm than the free-wheeling approach offered by imperative languages.

A more fundamental problem is the absence of nondeterminism in functional languages. This gives a problem whenever the time ordering of external events is important, as for example in a multi-user operating system [Hend82], or a program that has to service asynchronous interrupts. It has become clear that some fundamentally new primitive(s) must be introduced to support nondeterministic programming, and a number of suggestions have been made, including **nondeterministic merge** [Park82], **nondeterministic choice** (McCarthy's "amb" operator [McCar63]) [Abram82] [Jones83], **hiatons** [Park82], **timestamps** [Broy83], and **oracles.** No clear consensus has yet emerged on this issue.

9.3 Arrays

Most functional languages do not support arrays, which are vital for expressing numerical algorithms. The immediate reason for this is that the updating of arbitrary individual elements of an array is hard to implement efficiently because in principle this operation returns a whole new array (but see Hudak and Bloss [Hudak85] for ways to optimise this).

This problem is more technological than fundamental, and the SISAL development effort has the efficient handling of arrays as a major objective [Univ85].

10. Conclusions

We have presented a case for using functional programming languages as a tool to help meet the challenge of writing correct programs.

To this end we reviewed their type systems, means of combination and abstraction, amenability to formal methods, and support for parallel programming.

Language design is only one component of a software engineering methodology, but it has a far-reaching effect (for good or ill) on the rest of the programming environment. We conclude that functional languages merit serious consideration as a key component of advanced program development systems.

* Unix is a trade mark of Bell Labs.
* Ada is a trade mark of the US Department of Defence.

References

[Abels85] Abelson H and Sussman GJ, "Structure and interpretation of computer programs", MIT Press, 1985.

[Abram82] Abramsky S, "SECD-M - a virtual machine for applicative multiprogramming", Computer Systems Lab, Queen Mary College, Nov 1982.

[Augus84] Augustsson L, "A compiler for Lazy ML", Proc ACM Symposium on Lisp and Functional Programming, Austin, Aug 1984.

[Augus84a] Augustsson L, "A compiler for lazy ML", Proc ACM Lisp Conference, Austin, Aug 1984.

[Augus85] Augustsson L, "Compiling pattern matching", Functional Programming and Computer Architecture, ed Jounnaud, LNCS 201, Springer Verlag, Aug 1985.

[Back78] Backus J, "Can programming be liberated from the von Neumann style?", CACM 21(8), pp613-641, Aug 1978.

[Brook82] Brooks FP, "The mythical man month", Addison Wesley, 1982.

[Broy83] Broy M, "Applicative real-time programming", Proc 9th IFIP, Information Processing 1983, North Holland, pp259-264, 1983.

[Burge75] Burge WH, "Recursive programming techniques", Addison Wesley, 1975.

[Burn85] Burn G, Hankin CL and Abramsky S, "Strictness analysis of higher order functions", Science of Computer Programming (to appear); also DoC 85/6, Dept Comp Sci, Imperial College, London, Apr 1985.

[Burst77] Burstall, RM, "Design considerations for a functional programming language", Proc Infotech State of the Art Conference, Copenhagen, 1977.

[Burst77a] Burstall RM and Darlington J, "A transformational system for developing recursive programs", JACM 24(1), pp44-67, Jan 1977.

[Burst81] Burstall RM, MacQueen DB, Sannella DT, "Hope - an experimental applicative language", Edinburgh report CSR-62-80, 1981.

[Burst84] Burstall RM and Lampson B, "A kernel language for abstract data types and modules", Semantics of Data Types, Kahn et al, LNCS 173, Springer Verlag, 1984.

[Burst84a] Burstal RM, "Programming with modules as typed functional programming", Int'l Conf on 5th Generation Computing Systems, Tokyo, Nov 1984.

[Card85] Cardelli L, "Amber", Bell Labs, Murray Hill, April 1985.

[Card85a] Cardelli L, "Basic polymorphic typechecking", Polymorphism 2(1), Jan 1985.

[Card85b] Cardelli L and Wegner P, "On understanding types, data abstraction and polymorphism", Tech Rep CS-85-14, Dept Comp Sci, Brown Univ, Aug 1985.

[Clack85] Clack CD and Peyton Jones SL, "Generating parallelism from strictness analysis", Internal Note 1679, Dept Comp Sci, University College London, Feb 1985.

[Clock81] Clocksin and Mellish CS, "Programming in Prolog", Springer Verlag, 1981.

[Damas82] Damas L and Milner R, "Principal type schemes for functional programs", Proc ACM Symposium on Principal of Programming Languages, pp207-212, 1982.

[Darl80] Darlington J, "The design of efficient data representations", Imperial College, 1980.

[Darl81] Darlington J, "The structured description of algorithm derivation", in Algorithmic Languages, ed de Bakker and von Vliet, North Holland, 1981.

[Darl83] Darlington J, "Unifying logic and functional languages", Imperial College, 1983.

[Darl84] Darlington J, "Functional programming", in Distributed Computing, ed Chambers et al, Academic Press, 1984.

[Darl85] Darlington J, Field AJ, Pull H, "The unification of functional and logic programming", Dept of Computer Science, Imperial College, Feb 1985.

[Dijks68] Dijkstra EW, "The GOTO statement considered harmful", CACM 11(3), pp147-148, March 1968.

[Fages83] Fages F and Huet GP, "Complete sets of unifiers and matches in equational theories", Proc 8th Colloquium on Trees in Algebra and Programming, Springer Verlag LNCS 159, pp205-220, 1983.

[Fairb82] Fairbairn J, "Ponder and its type system", TR 31, Computer Lab, Cambridge, Nov 1982.

[Fairb85] Fairbairn J, "A new type checker for a functional language", Science of Computer Programming (to appear), 1985.

[Fairb85a] Fairbairn J, "Design and implementation of a simple typed language based on the lambda calculus", Tech Rep 75, Computer Lab, Cambridge, May 1985.

[Glas84] Glaser H, Hankin C and Till D, "Principles of functional programming", Prentice Hall, 1984.

[Goldf81] Goldfarb W, "The undecidability of the second order unification problem", Theoretical Computer Science 13, pp225-230, 1981.

[Gord79] Gordon MJC, Milner AJ, Wadsworth CP, "Edinburgh LCF", Springer Verlag LNCS 78, 1979.

[Hamm84] Hammond K, "The KRC manual", CSA/16/1984, DSAG-3, University of East Anglia, May 1984.

[Harl84] Harland DM, "User defined types in a polymorphic language", Computer Journal 27(1), pp47-56, Jan 1984.

[Hend80] Henderson P, "Functional programming", Prentice Hall, 1980.

[Hend82] Henderson P, "Purely functional operating systems", in Functional Programming and its Applications, ed Darlington, Henderson and Turner, CUP, pp177-192, 1982.

[Hindl69] Hindley R, "The principal type scheme of an object in combinatory logic", Trans American Mathematical Society 146, pp29-60, 1969.

[Hudak84] Hudak P and Kranz, "A combinator based compiler for a functional language", Proc 11th Symposium on Principles of Programming Languages, pp122-132, Jan 1984.

[Hudak85] Hudak P and Bloss A, "The aggregate update problem in functional programming systems", Proc 12th ACM Symposium on Principles of Programming Languages, 1985.

[Huet73] Huet GP, "The undecidability of unification in third order logic", Information and Control 22, pp257-267, 1973.

[Huet75] Huet GP, "Unification in the typed lambda calculus", Proc Symposium on the lambda calculus and computer science theory, Springer Verlag LNCS 37, pp192-212, 1975.

[Hugh82] Hughes RJM, "Graph reduction with supercombinators", PRG-28, Programming Research Group, Oxford, June 1982.

[Hugh84] Hughes RJM, "Why functional programming matters", PMG-40, Chalmers University of Technology, Goteborg, Sweden, 1984.

[Hugh85] Hughes RJM, "Strictness detection in non-flat domains", Programming Research Group, Oxford, Aug 1985.

[Islam81] Islam N, Myers TJ, Broome P, "A simple optimiser for FP-like languages", Proc ACM Conference on Functional Programming Languages and Computer Architecture, New Hampshire, pp33-40, Oct 1981.

[Johns85] Johnsson T, "Lambda lifting - transforming programs to recursive equations", Functional Programming and Computer Architecture, ed Jounnaud, LNCS 201, Springer Verlag, Aug 1985.

[Jones83] Jones S, "Abstract machine support for purely functional operating systems", PRG-34, Programming research group, Oxford, Aug 1983.

[Jones85] Jones ND et al, "An experiment in partial evaluation - the generation of a compiler generator", Proc Conf on Rewriting Techniques and Applications, LNCS, Springer Verlag, 1985.

[Kieb81] Kieburtz RB, Shultis J, "Transformations of FP program schemes", Proc ACM Conference on Functional Programming Languages and Computer Architecture, New Hampshire, pp41-48, Oct 1981.

[Kowal79] Kowalski R, "Logic for problem solving", North Holland, 1979.

[Land66] Landin PJ, "The next 700 programming languages", CACM 9(3), pp157-166, Mar 1966.

[Linds85] Lindstrom G, "Functional programming and the logical variable", ACM Conference on the Principles of Programming Languages, pp266-280, Jan 1985.

[MacQ85] MacQueen D, "Modules for Standard ML", Polymorphism 2(2), Oct 1985.

[McCar63] McCarthy J, "A basis for a mathematical theory of computation", in Computer Programming and Formal Systems, ed Braffort and Hirschberg, North Holland, pp33-70, 1963.

[Meira85] Meira SRL, "On the efficiency of applicative algorithms", PhD thesis, Computer Lab, University of Kent, March 1985.

[Miln78] Milner R, "A theory of type polymorphism in programming", Journal of Computer and System Sciences 17(3), pp348-375, Dec 1978.

[Miln85] Milner R, "The Standard ML core language", Polymorphism 2(2), Oct 1985.

[Moll84] Moller B (ed), "A survey of the project CIP", TUM-18406, Institut fur Informatik, Technische Universtitat Munchen, July 1984.

[Mycr81] Mycroft A, "Abstract interpretation and optimising transformations for applicative programs", Edinburgh CST-15-81, 1981.

[Mycr84] Mycroft A and O'Keefe R, "A polymorphic type system for Prolog", Artificial Intelligence 23, Elsevier, pp295-307, 1984.

[Park82] Park D, "The fairness problem and nondeterminism in computing networks", Proc 4th Advanced Course on Theoretical Computer Science, Mathematisch Centrum, 1982.

[Parts83] Partsch H and Steinbruggen R, "Program transformation systems", ACM Computing Surveys 15(3), pp199-236, Sept 1983.

[Peyt83] Peyton Jones SL, "Arbitrary precision arithmetic using continued fractions", Internal Note 1533, University College London, Dec 1983.

[Reddy85] Reddy US, "On the relationship between logic and functional languages", in Functional and logic programming, ed Degroot and Lindstrom, Prentice Hall, 1985.

[Robin71] Robinson JA, "The unification computation", in Machine Intelligence 6, Edinburgh University Press, pp63-72, 1971.

[Robin84] Robinson JA, "Syracuse University Parallel Expression Reduction - first annual report", Syracuse University, Dec 1984.

[Stoye85] Stoye W, "The implementation of functional languages using custom hardware", PhD thesis, Computer Lab, University of Cambridge, May 1985.

[Tighe85] Tighe S, "A study of the parallelism inherent in combinator reduction", Parallel processing program, MCC, Austin, Texas, Aug 1985.

[Turn76] Turner DA, "The Sasl manual", St Andrews, Dec 1976.

[Turn79] Turner DA, "A new implementation technique for applicative languages", Software Practice and Experience 9, pp31-49, 1979.

[Turn82] Turner DA, "Recursion equations as a programming language", in Functional Programming and its Applications, ed Darlington, Henderson and Turner, CUP, pp1-28, 1982.

[Turn83] Turner DA, "Functional programming and proofs of program correctness", in Tools and Notions for Program Construction, Neel (ed), CUP, 1983.

[Turn85] Turner DA, "The Miranda manual", University of Kent, Canterbury, 1985.

[Turn85a] Turner DA, "Miranda - a non-strict functional language with polymorphic types", Functional Programming and Computer Architecture, ed Jounnaud, LNCS 201, Springer Verlag, Aug 1985.

[Univ85] University of Manchester , "SISAL language reference manual", M-146, Computer Science Dept, University of Manchester, Jan 1985.

[Wadl81] Wadler P, "Applicative style programming, program transformation, and list operators", Proc ACM Conference on Functional Programming Languages and Computer Architecture, New Hampshire, pp25-32, Oct 1981.

[Wadl85] Wadler P, "Strictness analysis on non-flat domains", Programming Research Group, Oxford, Nov 1985.

[Wadl86] Wadler P, "Compiling pattern matching and list comprehensions", in Implementation of Functional Programming Languages, Peyton Jones, Prentice Hall, 1986.