PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Functional Programs as Executable Specifications [and Discussion]

D. A. Turner, J. Fairbairn, D. Park, P. Wadler, B. A. Wichmann and M. H. Rogers

Phil. Trans. R. Soc. Lond. A 1984 **312**, 363-388
doi: 10.1098/rsta.1984.0065

Phil. Trans. R. Soc. Lond. A **312**, 363–388 (1984) [363]
Printed in Great Britain

Functional programs as executable specifications

By D. A. Turner
Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, U.K.

To write specifications we need to be able to define the data domains in which we are interested, such as numbers, lists, trees and graphs. We also need to be able to define functions over these domains. It is desirable that the notation should be higher order, so that function spaces can themselves be treated as data domains. Finally, given the potential for confusion in specifications involving a large number of data types, it is a practical necessity that there should be a simple syntactic discipline that ensures that only well typed applications of functions can occur.

A functional programming language with these properties is presented and its use as a specification tool is demonstrated on a series of examples. Although such a notation lacks the power of some imaginable specification languages (for example, in not allowing existential quantifiers), it has the advantage that specifications written in it are always executable. The strengths and weaknesses of this approach are discussed, and also the prospects for the use of purely functional languages in production programming.

1. Introduction

Computers are a species of the genus 'symbol-processing machine'. By a symbol-processing machine we mean one that receives symbols at one or more input channels, and in response, possibly after an elapse of time during which the machine may undergo many changes of internal state, the machine transmits symbols along one or more output channels. We ignore the physical representation of the symbols and the nature of the processes whereby they are delivered to the machine or received from it, since by definition a symbol-processing machine is one built to transmit or transmute *information*, not energy or power or some other physical quantity.

From this point of view a symbol-processing machine may be regarded as the physical realization of a mathematical function. The idea of a function will be familiar: a function is a relation between two previously given sets (or 'types') called the *domain* and the *codomain*. To each element of the domain the function assigns a unique element of the codomain. Here the domain is the set of allowed input symbols (or vectors of input symbols if the machine has several input channels) and the codomain is a set that includes all possible output symbols (or vectors of symbols). The relation between the two is the function that the machine implements.

A convenient way to define a function is by giving a series of equations. For example, if the domain of possible input values is {A, B, C} and the codomain is {X, Y, Z}, a function f might be defined by the equations

$$\mathrm{f\,A} = \mathrm{X}, \quad \mathrm{f\,B} = \mathrm{Y}, \quad \mathrm{f\,C} = \mathrm{Z}.$$

Equivalently, we can give a set of input–output pairs describing the behaviour of the function, in this case {(A, X), (B, Y), (C, Z)}; this set is called the *graph* of the function. It is clear that if the function has an infinite graph it will in general require an infinite number of equations

to describe it. Sometimes, however, it will still be possible to describe the function in a finite number of equations, for example by introducing a variable. Thus if the domain and codomain are integers, we might define a function g, by

$$g\ n = n + n,$$

where n stands for an arbitrary integer; such an equation is really a schema, standing for an infinite number of equations of the earlier sort.

An important property of functions is *extensionality*: this is the principle that if two functions have the same graph they must be regarded as the same function. So if we were to define the function h (also on integers) by

$$h\ n = 2 \times n,$$

then g and h would be the *same* function, and we shall write g = h. So the mathematical idea of a function corresponds rather closely to the engineering notion of a *black box*: what matters is the input–output behaviour, not the internal mechanism by which it is achieved.

Returning now to the computer as a species of symbol-processing machine, if we leave aside purely physical differences (such as the fact that computers are built out of micro-electronic circuits, which enables them to operate much more rapidly than earlier devices), what are the differences between the computer and all the other kinds of symbol-processing machine, some of which have existed since much earlier periods? There is one essential difference: the computer is a *universal* symbol-processing machine, which can simulate the logical behaviour of any symbol-processing machine that it is possible to construct. Thus arises the problem of programming. Programming the computer consists of instructing it which, out of the infinity of possible symbol-processing machines, is the one we wish it to simulate on any particular occasion.

Note then that all computers are equivalent in this sense, that each is a universal machine that can simulate the behaviour of any other machine, and therefore among the machines that they can simulate are each other. Computers differ from one another (apart from physical differences such as speed, or number of input–output channels) solely in the language in which they require to be instructed as to the machine they are to simulate.

If we accept that a symbol-processing machine is an implementation of a mathematical function, then the problem of programming a computer can be restated as that of instructing it as to which function we require it to implement, with the function being given as a series of equations or something similar. This is the point of view of functional programming.

Before proceeding further with the ideas of functional programming, there are two issues that warrant further discussion. The first is whether we have glossed over certain difficulties in treating symbol-processing machines as implementations of functions; the second is that the notion of a universal machine is itself rather remarkable and merits some further comment.

There are three apparent objections to the claim that the behaviour of an arbitrary symbol-processing machine can be captured by the mathematical notion of function. The first is that the machine might behave unreliably, so that, for example, if there are two legal outputs X and Y, represented say by 1 V (volt) and 2 V respectively, the machine might sometimes get into a funny state that gave a reading 1.5 V, or break altogether and give a reading of 0 V. We can dismiss this problem as not our concern: machines that are unreliable in this sense are of no interest to computing science (except as objects that we return to the manufacturer with a note asking that they be replaced).

There is, however, one way in which the machine can fail to produce a meaningful answer that we cannot dismiss as being outside the concern of computing science. Anyone who has had even the most casual exposure to programming will have observed that it is possible to write programs corresponding to machines that fail to produce any output by getting into an endless cycle of state changes. This is an inescapable problem because it can be shown that any programming language that is universal (in the sense that every computable function can be defined in it) will contain programs with endless loops. It is not theoretically possible to have a universal language from which all such programs have been filtered out (for example, by a test in the compiler). The class of functions implemented by symbol-processing machines must therefore be understood to include *partial functions*, which fail to assign a value to some of the points in their domain. Partial functions are a nuisance mathematically, because with them functional composition is not always defined, so we adopt the convention, due to Scott (1970), that computations that fail to terminate are considered to yield a special value, $\perp$ (pronounced 'bottom'). Throughout what follows we assume that a bottom element has been adjoined to each type and that the term 'function' therefore includes partial function.

A second objection to the 'machines-as-functions' hypothesis is that the machine may be *non-deterministic*, in the sense that for a given input the machine may be capable of returning one of several outputs, all of which we regard as legitimate, and that we either cannot, or do not wish to, predict which one we will obtain on any particular occasion. A number of computing scientists, most notably E. W. Dijkstra (1976), have argued that non-deterministic machines should be considered the normal case. We can remark here that a non-deterministic machine may be represented by a function that returns a set, and thus does not fall outside the functional point of view. Nevertheless it is only fair to say that the programming of non-deterministic systems within a purely functional style poses certain logical problems that at the time of writing do not have any generally agreed solution; we shall return to this point in the closing section of the paper. Most work on functional programming has been concerned with the description of deterministic systems and this is the point of view we shall adopt in the bulk of the remainder of this paper.

The third possible objection to the characterization of machines as functions is that this appears not to cover machines whose behaviour is history sensitive. Consider, for example, a machine whose input symbols are drawn from the set {0, 1} and whose output is the same as its input if the number of '1' symbols previously received is even, but the complement of its input if the number of previous '1' symbols is odd. (Such a machine would be said to have two internal states and to flip state every time it received a '1'.) Although the behaviour of this machine cannot be characterized by a function from input events to output events, it is still the case that the output history, taken as a whole, is a function of the input history, taken as a whole. To write down some equations describing this function we need some notation for infinite sequences. We shall adopt the convention (common to several programming languages, including the one used for all the examples in this paper) that a:x denotes a sequence whose first member is 'a' and whose remainder is the sequence 'x'. With this convention the function f, which characterizes the behaviour of our parity flipping machine, can be described as

$$\begin{aligned} \mathrm{f}(0{:}\mathrm{x}) &= 0{:}\mathrm{f}\,\mathrm{x}, \\ \mathrm{f}(1{:}\mathrm{x}) &= 1{:}\mathrm{g}\,\mathrm{x}, \\ \mathrm{g}(0{:}\mathrm{x}) &= 1{:}\mathrm{g}\,\mathrm{x}, \\ \mathrm{g}(1{:}\mathrm{x}) &= 0{:}\mathrm{f}\,\mathrm{x}. \end{aligned}$$

Here f is a function whose input and output are both infinite sequences of 0s and 1s, and g is an auxiliary function of the same type. Note that f describes the behaviour of a machine that starts life in the 'even' state, while g describes a machine that starts life in the 'odd' state. In fact these equations do not give an explicit definition of f and g, but only set up a relation of mutual recurrence between them; it is easy to see, however, that the equations are sufficient to determine both functions uniquely.

The class of machines whose behaviour can be described by functions from input history to output history is quite general and comprises all (deterministic) machines with internal state. Functional programming in its modern form may be said to date from the development of programming systems that made it easy to write down definitions of functions whose domain and codomain consist of infinite sequences (Friedman & Wise 1976; Henderson & Morris 1976; Turner 1976).

The remaining task of this introduction is to elucidate the claim that the digital computer is a universal machine. The claim is that a computer can (given a suitable program) simulate the behaviour of any symbol-processing machine that it is possible to construct; or equivalently, compute any function that it is possible to compute by mechanical means. It might be though that there is no limit to the ingenuity that can be employed in calculating the value of a function mechanically, and that therefore the class of functions computable by mechanical means is not a definite class at all.

It turns out, however, that there is a definite class of functions whose value at any point in their domain can be computed mechanically (see, for example, Davis 1958). A fairly small amount of apparatus is sufficient to compute all of these functions, and adding more apparatus (so long as it is mechanically realizable) makes no difference at all to the class of functions that can be computed. This class of functions is called the *recursive functions* (because the functions that can be computed by mechanical means are the same as the functions that can be characterized by recursion equations).

There are many interesting functions that are not recursive. Consider, for example, the function that when applied to a formula of first-order logic, returns 1 if it is a theorem and 0 if it is not. It can be shown that if we suppose a machine can be constructed to compute this function, paradoxes ensue, and that therefore such a machine cannot be built (Turing 1936). This is not because there is something ill-defined about the notion of truth in first-order logic; on the contrary, we can build a machine, M, which will, one after another, generate all the theorems of first-order logic (for example, by constructing in order of increasing length, all valid proof sequences). Moreover, it is known that first-order logic is *complete*, in the sense that every formula that logically follows from the axioms can be proved as a theorem and will therefore eventually show up on the output list of machine M. A formula that does not eventually appear on the output list is not a theorem and does not follow from the axioms.

The function that maps formulae of first-order logic onto {0, 1} as they are or are not theorems is therefore perfectly well specified mathematically, it just happens not to be computable. Note, however, that a partial function that maps formulae onto {⊥, 1}, returning 1 for theorems and failing to terminate on non-theorems *is* computable, it can obviously be computed by a minor variant of machine M. Another well known example of a function known not to be computable is one that takes a computer program (complete with any data it may require) and returns 0 or 1 as the program terminates or fails to terminate. Fortunately almost all of the functions in which we are interested for practical reasons do turn out to be computable.

The distinction between functions that are in principle computable by mechanical means and those that are not so computable, even in principle, is a fundamental discovery of mathematical logic. A consequence of this distinction is that when we consider possible notations for defining functions, these notations fall into two classes: first, those notations that have the property that any function that can be defined by using them is known to be mechanically executable; second, those more powerful notations that also enable us to write down definitions of functions, such as the decision function for first-order logic or the halting function for computer programs, which are not even in principle computable. Examples of notations of the first kind include recursion equations, lambda-calculus and FORTRAN. An example of a notation of the latter kind would be standard (i.e. Z.F.) set theory (see, for example, Drake 1974). Only notations of the first kind can be considered for use as programming languages; but for the purposes of general mathematical discourse, notations of the second kind are definitely required. (At least, this seems to be the view of the overwhelming majority of mathematicians at the time of writing.) Moreover in the process of reasoning about programs, which are written in notations of the first kind, we shall sometimes need to write down ideas that can only be expressed in notations of the second kind.

In what follows we present (by means of a series of examples) a style of functional programming that makes use of recursion equations together with some notation from set theory. Such a language may be regarded as lying somewhere on the boundary between programming languages and specification languages. Compared with a conventional programming language it is often rather concise, and shares with ordinary mathematics the property, which programming languages do not usually possess, of being *static* (i.e. equations do not change their truth value over time). Considered as a specification language it suffers from the restrictions inherent in being recursive: only computable functions can be denoted, so there are some useful and interesting specifications that cannot be expressed within it. It does, however, have the compensating advantage that specifications written in it are always executable. It can also be regarded, whenever we wish to do so, as being embedded in a richer mathematical language including notations for non-computable functions (we shall give one example of this).

In the closing sections of the paper we discuss briefly the current state of knowledge as regards the efficiency of functional programs (both in space and in time) when compared with conventional ones, the relation between functional programming and logic programming in the style of PROLOG, and the prospects for the adoption of purely functional languages in production use.

2. A FUNCTIONAL PROGRAMMING LANGUAGE

The functional programming language used in this paper is called MIRANDA and is based on the author's earlier languages SASL (Turner 1976) and KRC (Turner 1982*b*) with the addition of a type discipline essentially the same as that of ML (Gordon *et al.* 1979). There are a number of quite similar languages in use, of which perhaps the best known is HOPE (Burstall *et al.* 1980). What is presented here may therefore be regarded as representative of the modern style of functional programming, and should be contrasted with the more traditional style of functional programming based on LISP (McCarthy *et al.* 1962). We shall begin with a simple example, which brings out the basic idea of using recursion equations both to define data types and to

define functions over those data types. We shall then elaborate on the basic theme by introducing other necessary features of the notation before proceeding to some more substantial examples.

An example

Suppose that we are interested in binary trees, of the following rather simple kind. There are two atomic trees, called ALPHA and BETA, of which we suppose no properties other than that they are equal to themselves and unequal to each other. If a tree is not atomic, then it is an ordered pair, with two components, both of which are trees. Suppose also that we are interested in making on trees the operation of *reflection*, by which we mean recursively reversing the order of the components at every node in the tree. Figure 1 illustrates this with a diagram of a typical tree together with the result of reflecting it.

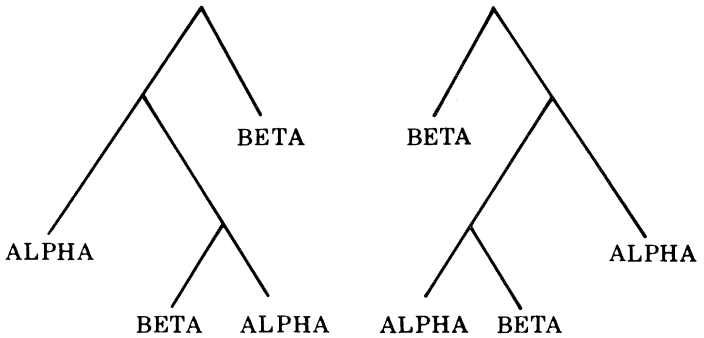

FIGURE 1. A typical tree with its reflection.

If we are to express this problem in a programming language, the first thing that we require is that it should allow us to introduce the data type *tree* in a suitably abstract way. No doubt we could find some scheme for encoding trees in terms of an already existing data type, say lists of integers, but it is clearly very undesirable that we should be forced to do so. There is now general agreement that any reasonable programming language should permit the user to introduce new data types with just the properties he required, and no irrelevant ones. Secondly we want to define the function *reflect*, which is a mapping from trees onto trees. Recursion equations can be used to accomplish both tasks, as follows (this is our first complete example of a functional program):

```
tree :: = ALPHA|BETA|pair tree tree
reflect ALPHA = ALPHA                                   (1)
reflect BETA = BETA                                     (2)
reflect (pair x y) = pair (reflect y) (reflect x).      (3)
```

The first line of the program defines the data type *tree* (the ' :: = ' sign should be pronounced 'comprises'). Note that it introduces four new identifiers: 'tree', which is the name of a type; 'ALPHA' and 'BETA', which are the names of atomic trees; and 'pair', which is a function that generates a new tree from two existing trees. It says that the data type 'tree' is the free algebra generated by the two 0-ary operators 'ALPHA' and 'BETA' and the 2-ary operator 'pair'. The remaining three lines of the program define the behaviour of the reflect function

on the three kinds of tree (the numbering of the equations is for our convenience here and is not part of the program). Note that the use of pattern matching in equation (3) avoids the need to invent names (such as 'left' and 'right') for the selector functions on a non-atomic tree.

It should be clear that the three equations given are sufficient to deduce the result of reflecting any finite tree. For example, the result of reflecting the tree in figure 1 may be inferred as

reflect (pair (pair A (pair B A) B)
= pair (reflect B) (reflect (pair A (pair B A)) (by (3))
= pair B (pair (reflect (pair B A)) (reflect A)) (by (2), (3))
= pair B (pair (pair (reflect A) (reflect B)) A) (by (3), (1))
= pair B (pair (pair A B) A) (by (1), (2))

where for brevity we have written A and B for ALPHA and BETA. This demonstrates that for computational purposes the equations of the program can be used as left-to-right replacement rules. Note that it is in general possible to make two or more replacements simultaneously in different parts of the expression under evaluation.

From the three equations of the program other properties of reflect can be inferred by using standard mathematical proof techniques. For example, the fact that reflect is its own inverse is shown by the following little proof (which works by induction over the structure of trees)

THEOREM reflect (reflect x) = x for all x::tree

Proof by induction on x

Case ALPHA

reflect (reflect ALPHA) = reflect ALPHA (by (1))
= ALPHA (by (1))

Case BETA

similarly... (by (2), (2))

Case pair x y

reflect (reflect (pair x y))
= reflect (pair (reflect y) (reflect x)) (by (3))
= pair (reflect (reflect x)) (reflect (reflect y)) (by (3))
= pair x y (ex hyp, ex hyp)

■

The sign :: means 'is of type'. This style of proof, based on the substitutivity of equality and structural induction (see Burstall 1969) is really very straightforward. Nevertheless it can get one quite a long way: for example, using essentially this method, the author has proved the correctness of a compiler for an applicative language (Turner 1981).

This example of a program to reflect binary trees has brought out several central points about the type of programming language here under discussion:

(i) that recursion equations are a convenient notation to define both data types and functions over those data types;

(ii) that the equations of the program can be read mathematically as premises from which to deduce other properties of the functions involved;

(iii) that they can be used to compute values of the functions by treating them as left-to-right replacement rules (so that computation is here a special case of deduction.)

In connection with (iii), it should be noted that in general more than one replacement can be made at a time, so there is significant opportunity to exploit concurrency in the implementation of a functional programming language.

A basic result of mathematical logic, Kleene's (1952) first recursion theorem tells us that whenever a system of recursion equations has a unique function for its solution, the function computed by treating the equations as left-to-right replacement rules is the *same* one. This result should be considered as the logical foundation of functional programming.

This basic theme will now be elaborated by the introduction of a number of necessary additional notions and notations.

Infinite and partial data structures

Continuing for a moment with the tree example, there is no reason why we should be forced to confine our attention to finite trees. We might be interested, for example, in the infinite tree shown in figure 2. This tree can be defined uniquely by the equation

inftree = pair ALPHA inftree.

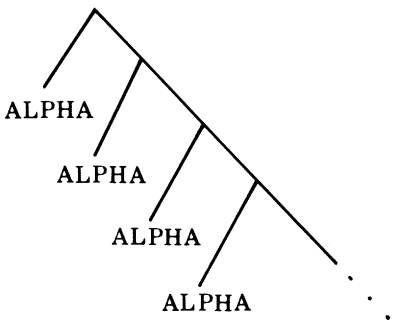

FIGURE 2. An infinite tree.

Moreover its reflection can be defined as

inftree2 = reflect inftree,

from which we may deduce

inftree2 = reflect (pair ALPHA inftree)
= pair (reflect inftree) (reflect ALPHA)
= pair (reflect (pair ALPHA inftree) ALPHA
= pair (pair (reflect inftree) ALPHA) ALPHA
and so on....

It is clear that we can compute with infinite trees in the sense that as much of their structure as may be required in any particular computation can be inferred in a finite number of steps. This will work on condition that the constructor function 'pair' is *lazy*, i.e. does not force premature evaluation of its arguments. The use of the term 'lazy evaluation' in this context was initiated by Henderson & Morris (1976). Of course the programmer must take care not to force complete evaluation of an infinite object; for example by trying to count the number of ALPHAs in inftree.

Having a lazy constructor function will also permit us to build *partial* trees. If we let $\perp$ denote the completely undefined tree (the result of a non-terminating computation of type tree), then the partial trees

pair ALPHA $\perp$
pair ALPHA (pair ALPHA $\perp$)
pair ALPHA (pair ALPHA (pair ALPHA $\perp$))...

exist and are all different from each other and from $\perp$, and it is clear that their *limit*, in a sense that can be made precise, is the infinite tree 'inftree'. Every infinite tree that can be defined by recursion equations is the limit of such a series of partial trees. The richer data type including partial and infinite trees can be derived from the finite type by including $\perp$ as an extra generator and closing the resulting data type under limits. The extension of recursion theory to deal with recursion over infinite objects is essentially due to Scott (1976).

Obviously an important question when we define a new data type is whether we intend it to be an ordinary finite algebra or whether we wish to include partial and infinite objects. We will assume that our programming language gives us some simple syntactic means (not discussed here) for discriminating between the various different possibilities that arise along these lines.

It is possible to make inductive proofs of the properties of programs that operate on infinite data objects by using a modified version of structural induction (Turner 1982*a*). Alternatively, one can use the fixpoint induction rule of Scott (1970).

Generic types

It is a highly desirable feature of a programming language that there should be some simple syntactic discipline that ensures that only well typed applications of functions to arguments can occur. For example, our function reflect is a mapping from trees to trees, for which we write

reflect :: tree → tree

and we expect to get an error message (at compile time) if reflect is applied to an object of another type, say an integer. That is to say, our programming language is *strongly typed*. We do not require the user to declare the type of reflect in the program, however, It is deduced from the equations of the program by using the algorithm of Milner (1978). The Milner type-discipline also permits generic (or 'polymorphic') functions; for example, if we define the identity function by

I x = x

then the type attributed to I is

I :: * → *,

where '*' is a type variable, standing for an arbitrary or generic type. The identifier I should be thought of as ambiguously denoting any one of a whole family of functions, one for each data type.

We can extend this idea to user-defined type constructors in the following way. Suppose we wish to modify the type 'tree' so that an atomic tree, instead of being just ALPHA or BETA, is a leaf of some arbitrary type; say, integer or boolean or string; the only constraint being

that in any one given tree, all the leaves must be of the same type. This new type, or rather *family* of types, is defined by

tree * :: = leaf * | pair (tree *) (tree *).

The name 'tree' is now that of a *type-constructor*, '*' stands for an arbitrary type, and 'tree int', 'tree bool' are examples of types that can be constructed by using the type constructor. So we have

pair (leaf true) (leaf false) :: tree bool
pair (leaf 1) (leaf 3) :: tree int
leaf (pair (leaf 1) (leaf 2)) :: tree (tree int),

whereas, say, pair (leaf 1) (leaf true) is *not* a legal tree and will evoke a compile-time error message. In connection with the new generalized trees the reflect function may be defined

reflect (leaf x) = leaf x
reflect (pair x y) = pair (reflect y) (reflect x).

The type of this new reflect function is (tree * → tree *), meaning that it may be applied to any object in the tree family, returning in each case a tree of the same type as its argument.

The Milner type-system embodies a major step forward in programming language design by making it possible to have polymorphic functions without compromising the ability to detect type errors at compile time. Previously it was possible to support polymorphism only by delaying type checking until run time. The Milner type scheme has been successfully used in ML, the metalanguage of the LCF proof system (Gordon *et al.* 1979), (Milner, this symposium). For a fuller discussion of the type structure of MIRANDA, which differs in some respects from that of ML, the reader is referred elsewhere (Turner 1984).

Built-in types

It should be clear that the type-definition apparatus is sufficiently powerful that the language does not need to have any built-in type notions at all, apart from the notion of *function*. All the standard types can be defined from first principles, by using the type definition mechanism. For example, the boolean type, the list type-family (for each type T there is the type of lists whose elements are in T), and the type of natural numbers, may each be introduced as

bool :: = true | false
list * :: = nil | cons * (list *)
nat :: = zero | suc nat.

By using the notation introduced by the last equation above, the number seven would be written

suc (suc (suc (suc (suc (suc (suc zero)))))).

Being forced to represent numbers in this way would actually be rather inconvenient, as well as inefficient in both time and space. This seems sufficient reason to allow numbers, and certain other data types, to be provided built-in to the language, even though it is not logically necessary to do so. The built-in type notions of MIRANDA are Booleans, numbers (both integer and floating point), characters (the usual ASCII character set), lists, tuples and, of course, functions. This choice is obviously to some extent arbitrary, and either a greater or a smaller selection of built-in data types could be provided.

To help the reader follow the examples in the next section we here introduce the notation used with lists. Lists are written by using square brackets and commas; so, for example, if we write

x = [2, 3, 4, 5]

then x is of type [int], that is 'list of int'. The main operations on lists are '#' (length), '!' subscripting, ':' (prefix), '++' (concatenation) and '−−' (list or set difference). So, for example, #x is 4, x!2 is 3, (note that subscripting starts at 1) 1:x is the list [1, 2, 3, 4, 5], x++x is the list [2, 3, 4, 5, 2, 3, 4, 5] and x−−[2, 5] is the list [3, 4]. On lists containing repeated elements the −− operator behaves as "bag difference", see (Turner 1982*b*) for its definition.

All the elements of a list must of course be of the same type. For sequences of mixed type we use tuples, which are denoted by round brackets and commas; for example (1, true, "red") is a tuple, of type (int, bool, [char]). Tuples and lists are quite different: so for example, tuples cannot be subscripted. Note that the string constant "red" is just shorthand for the list of characters ['r', 'e', 'd'].

Some shorthand notation is also provided for lists whose elements form an arithmetic series. For example [1..10] denotes the list of the numbers 1 through 10, and [0..] is the list of all the natural numbers in ascending order (lists can be infinite). Finally, note that the ':' operator can be used on the left, inside a formal parameter, in the equations defining a function. So, for example, the function for reversing a list is defined

```
reverse [ ] = [ ]
reverse (a:x) = reverse x++[a].
```

Higher order functions and currying

MIRANDA is fully higher order. One can define functions that are second-order, third-order, and so on without limitation. In fact any function of two or more arguments is really a higher order function, in the following way. Suppose we define the function 'plus' by

```
plus a b = a+b,
```

then plus is of type **int**→(**int**→**int**) and (plus 3) say, has a meaning in its own right: it is the function that adds 3 to integers. This device is known as 'currying', after the logician H. B. Curry (Curry & Feys 1956). As is normal when working with curried functions, we adopt the convention that functional application is left-associative.

An interesting example of a higher order function is the famous fixpoint operator of Curry, which we can define by

```
fix h = f
        where f = h f.
```

Note that the type of fix is given by

```
fix :: (*→*)→* .
```

Another useful higher order function, which is defined in the library of standard functions, is assoc, which has the type

```
assoc :: [(*, **)]→(*→**) .
```

This is an example of a generic type that requires more than one type variable to write it down; following ML, we use '*', '**', '***', etc. for successive type variables. The action of assoc is to take a list of pairs that represent the graph of a function, and to return the function itself. For instance

assoc [(1, red), (2, blue), (3, green)]

would be a function that maps integers to colours; we assume a type colour has been appropriately declared with constituents red, blue, etc.

Set abstraction

The last example just given introduces the idea that a list without repetitions may be used to represent a set. A powerful feature of ordinary mathematical notation is the ability to define a set by some property that all its members hold in common. This is a feature also of our functional programming language, by using a notation very similar to that of Z.F. set theory. We can write

{f x | x ← A},

which we pronounce 'set of all f x such that x in A'. The sign '←' denotes set membership, and the variable introduced on its left is a local variable of the set expression, whose scope is delimited by the curly brackets. The general form of these set expressions is

{EXP | CONDITION; . . . ; CONDITION},

where each CONDITION is either a GENERATOR or a FILTER. A GENERATOR is of the form

VARS ← EXP

and states that one or more variables (local to the set expression) are drawn from some previously existing set. A FILTER is just a boolean expression, further restricting the range of the variables introduced by the generators.

As already implied, the set is not a separate data type in this programming language; a set here is just a list from which repetitions have been removed. In addition to set abstraction we also permit list abstraction, which has the same syntax except that we use square brackets instead of braces around the expression, thus [. . . | . . .]. With list abstraction, repetitions are not removed from the result. We give three examples of the use of set abstraction:

squares = {n × n | n ← [0 . .]}

is the set of all the squares of natural numbers. The Cartesian product of two lists x and y (set of all pairs with one member from each) can be defined

cp x y = {(a, b) | a ← x; b ← y}.

Finally, the list of all Pythagorean triangles, that is right-angled triangles with integer sides, such as [3, 4, 5], can be written

pyths = {[a, b, c]) | a, b, c ← [1 . .]; a × a + b × b = c × c}.

We do also permit set abstraction in the form

{GENERATOR | FILTER},

which is a case that occurs commonly enough to be worth having a special form for it. For instance, if we have a set of pairs, R, representing the graph of a relation, and we wanted to derive from R a relation made irreflexive by removing all pairs of the form (a, a) we could write

{(a, b) ← R | a ≠ b}.

In one sense set abstraction adds no power to the language, because no data structure can be built with its aid that could not also be constructed by some kind of explicit recursion over the lists involved. In practice, however, the presence of set expressions can lead to a considerable improvement in the conciseness and clarity of programs and may be said to permit the expressions of ideas at a higher level of abstraction, by removing the necessity to deal with a layer of housekeeping details.

As an example of this we give a functional presentation of quicksort, due to Silvio Meira, a graduate student at the University of Kent (Meira 1983), Quicksort is the method of sorting, originally devised by Hoare (1962), in which the elements to be sorted are first partitioned into all those less than some arbitrary element (say the first), and all those greater than the chosen element. The two parts are then each sorted, recursively, by using the same method. Meira's definition of quicksort is

sort [ ] = [ ]

sort (a:x) = sort [b ← x | b ⩽ a] ++ [a] ++ sort [b ← x | b > a].

To appreciate the conciseness of this definition the reader should compare it with the definition of quicksort in a conventional programming language; see, for example, Graham (1983).

3. Some examples of program development

As illustration of the thesis embodied in the title of this paper, 'functional programs as executable specifications', we here take a small number of problems and for each one show how, working within the functional language framework sketched above, we can arrive at an executable solution expressed at a high level of abstraction. In each case except the first we will exhibit a series of solutions of increasing efficiency. In fact example (*c*) is a counter-example to the thesis in that the first solution we arrive at is not executable, but we show how an executable solution may be derived from it.

(*a*) *Permutations*

We seek to define a function that will take a finite list of distinct objects, and return a set (list) of all possible permutations of the original list. So we wish to define a function 'perms' with the type

perms :: [*] → [[*]].

Clearly the set of all permutations of the empty list is a singleton, containing just the empty list. For a non-empty list we can generate a permutation by choosing any member of the list

for the first member of the permutation, and then follow this by any permutation of the remaining elements; if we do this in each possible way we shall generate all the permutations. This reasoning leads directly to the recursive definition

```
perms [ ] = [[ ]]
perms x = {a:p | a←x; p←perms (x − − [a])}.
```

(*b*) *Topological sort*

Given a *partial ordering* relation, we have to derive any total ordering that is compatible with the given information. An example of a partial ordering is shown in figure 3.

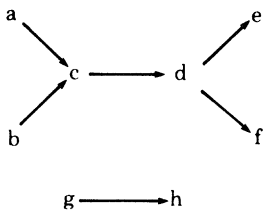

FIGURE 3. A partial ordering.

In this case there are many total orderings acceptable as output, two of which are

```
a b c d e f g h,
b a c g h d f e.
```

Given the information about the initial partial ordering, we have to produce any one of the compatible total orderings as output. In various guises, this is a problem that repeatedly turns up in computing science. For example, the type-checking phase of the compiler for MIRANDA involves an application of topological sort.

The first thing we have to decide is how the information about the partial ordering relation will be presented. We shall suppose that we are given (i) the set of all the elements involved in the ordering (this is called the 'carrier' of the relation) and (ii) a list of pairs such that if (p, q) is in the list, then p immediately precedes q in the ordering. For the partial ordering shown, this second item of information could be given as the list

```
[(a, c), (b, c), (c, d), (d, e), (d, f), (g, h)].
```

At first sight one might think that the carrier could be deduced from the list of pairs. The partial ordering might contain isolated elements (elements that do not precede nor are preceded by, anything else), however, and these would not be present in the list of pairs, since our notion of 'immediately precedes' is irreflexive.

We give first a very naïve definition of topological sort, which arises from a more or less literal formalization of the natural language definition of topological sort:

```
tsort :: [*] → ([(*, *)] → [*])
tsort carrier R = any {x←perms carrier | respects R x}
respects R x = and {posn a x < posn b x | (a, b)←R}
posn a x = hd [i←[1 . . #x] | x!i = a].
```

This uses four library functions: 'perms', of which we have seen a definition earlier; 'any', which picks a member of a list (and we do not care which); 'and', which takes a list of truth values and returns their conjunction; and 'hd', which takes the first member of a list. The method used by the above definition of tsort is to look at all permutations of the carrier, and pick any one that respects the given precedence relation R. A list x is said to 'respect' a relation R if whenever (**a, b**) is in R, **a** comes before **b** in the list x. The function 'posn' finds the index number of the first occurrence of an item in a list.

It is easy to convince oneself that the above solution is correct, and it is clearly executable, but at a cost in time that is completely unacceptable for any but very small carriers. This is a 'British Museum' algorithm. It proceeds by searching the space of all permutations of the carrier, completely blindly, until it finds one that is compatible with the partial ordering. Given a partial ordering involving, say, a few hundred elements, which would be typical of a practical application, the number of permutations of the elements is extremely large. To search them all, or even a significant fraction of them, is out of the question.

To find a solution in a reasonable time it is obvious that we must construct it piecewise, and using some intelligence. We begin by observing that for a first member of the output list we may choose any member of the carrier that is not preceded by anything in the relation. We can then remove this element from the carrier, and all pairs involving it from the relation, and topologically sort the remainder. This gives us the makings of an efficient solution.

```
tsort [ ] [ ] = [ ]
tsort carrier R = a : tsort (carrier − − [a]) {(m, n) ← R | m ≠ a}
                  where
                  a = any (carrier − − range R)
range R = {b | (a, b) ← R}.
```

Note that we generate the set of elements with no predecessor by taking the set difference between the carrier and the range of the relation. The program runs in polynomial time and is usable in practice. The most efficient known solutions may be derived from the above by building ancillary data structures that speed up the detection of elements with no predecessor (see, for example, Tarjan 1972).

(*c*) *The Hamming numbers*

The following problem is found in Dijkstra (1976) and is attributed by him to R. W. Hamming. We have to print in ascending order the series

1, 2, 3, 4, 5, 6, 8, 9, 10, 12 . . .

of those numbers whose prime factors are 2, 3, 5 only.

The solution can be written down directly as

```
ham = SORT {2↑a × 3↑b × 5↑c | a, b, c ← [0 . .]},
```

where SORT is a sorting function on infinite lists. This is not an executable program, even in principle, however, because there cannot exist a recursive definition of a function SORT that works on infinite lists. (We leave it as an exercise for the reader to prove that SORT is not computable (hint: assume SORT is computable and derive a decision procedure for a

problem known to be undecidable). For example by applying SORT to the output of a program that generates all the theorems of first-order logic we would obtain a decision procedure for first-order logic.) We can *derive* a recursive definition of ham from the above, however, as follows:

$$
\begin{aligned}
\text{ham} &= \text{SORT}\ \{2\uparrow a \times 3\uparrow b \times 5\uparrow c \mid a, b, c \leftarrow [0\,..]\} \\
&= \text{SORT}\ (\{1\} \cup \{2 \times h \mid h \leftarrow H\} \cup \{3 \times h \mid h \leftarrow H\} \cup \{5 \times h \mid h \leftarrow H\}) \\
&\quad \textbf{where}\ H = \{2\uparrow a \times 3\uparrow b \times 5\uparrow c \mid a, b, c \leftarrow [0\,..]\} \\
&= 1 : \text{SORT}\ (\{2 \times h \mid h \leftarrow H\} \cup \{3 \times h \mid h \leftarrow H\} \cup \{5 \times h \mid h \leftarrow H\}) \\
&= 1 : \text{MERGE}[\text{SORT}\ \{2 \times h \mid h \leftarrow H\}, \text{SORT}\ \{3 \times h \mid h \leftarrow H\}, \text{SORT}\ \{5 \times h \mid h \leftarrow H\}) \\
&\quad \textbf{where}\ \text{MERGE}[\text{SORT X1}, \ldots, \text{SORT Xn}] = \text{SORT}\ \{\text{X1} \cup \ldots \cup \text{Xn}\} \\
&= 1 : \text{MERGE}[[2 \times h \mid h \leftarrow \text{SORT H}], [3 \times h \mid h \leftarrow \text{SORT H}], [5 \times h \mid h \leftarrow \text{SORT H}]].
\end{aligned}
$$

But SORT H is ham, so we have

$$\text{ham} = 1 : \text{MERGE}[[2 \times h \mid h \leftarrow \text{ham}], [3 \times h \mid h \leftarrow \text{ham}], [5 \times h \mid h \leftarrow \text{ham}]].$$

This is a recursive definition of ham, the list of all hamming numbers, in terms of MERGE, a function that takes a list of sorted lists, and interleaves them to produce a sorted result. Unlike SORT, the function MERGE is computable, and from the last line, after a little rearranging, we arrive at the program

$$
\begin{aligned}
&\text{ham} = 1 : \text{MERGE}[\text{mult 2 ham}, \text{mult 3 ham}, \text{mult 5 ham}] \\
&\text{mult n x} = [n \times a \mid a \leftarrow x] \\
&\text{MERGE}[X, Y, Z] = \text{merge X}(\text{merge Y Z}) \\
&\text{merge}\ (a : x)\ (b : y) = a : \text{merge x}\ (b : y),\ a < b \\
&\qquad\qquad\qquad\qquad = b : \text{merge}\ (a : x)\ y,\ a > b \\
&\qquad\qquad\qquad\qquad = a : \text{merge x y},\ a = b :
\end{aligned}
$$

Here we have defined the three-way infinite merging function 'MERGE' in terms of two calls to a two-way infinite merging function 'merge'. In the definition of 'merge' we have written three right-hand sides, distinguished by *guards* (these are boolean expressions written to the right of a comma). This program is both time and space efficient, and the reader may recognize it as the 'stream-processing' solution of Kahn & McQueen (1977). If we think of a function from infinite lists to infinite lists as being a *process* then the above program can be thought of as describing a system of five communicating processes, namely 'mult 2', 'mult 3', 'mult 5' and two instances of a 'merge' process (see figure 4).

It is instructive to compare the functional solution given here with the corresponding solution written in a language based on communicating processes, such as OCCAM (Inmos 1982). In fact the program is very much harder to write in OCCAM, partly because of the need to make explicit the existence of buffers in the input channels to the merge processes.

This example is interesting for two reasons. First, because it shows how systems of communicating processes can be described in a purely functional language by recursion over infinite lists; this is important because it open up the way for problems involving input–output and communication to be dealt with cleanly within the functional style, as in Henderson (1982). The second point of interest brought out by this example is that an equational language of

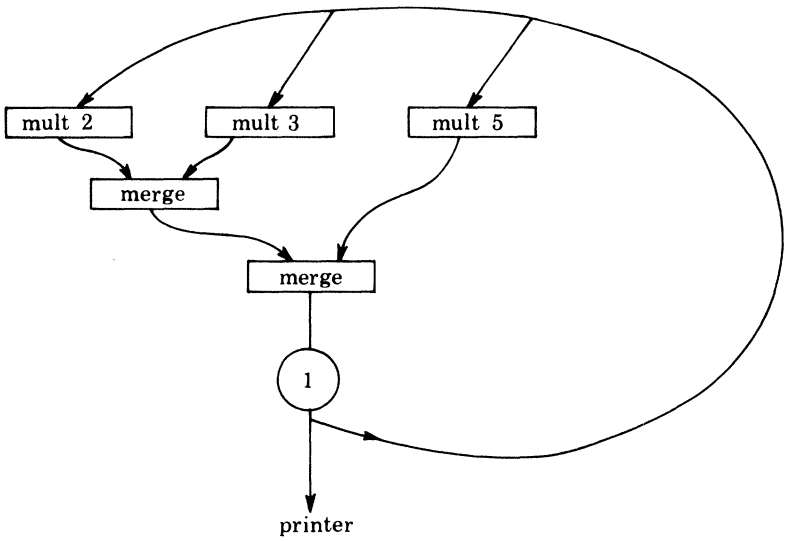

FIGURE 4. The process struture of the Hamming problem.

the kind being used here can be regarded when convenient as being embedded in a richer mathematical language including notations for non-recursive functions, and that it is often possible to derive an executable program, by a process of successive transformations, from an initial specification written in the richer, non-executable language. The problem of formal program derivation by the method of successive transformation and the related problem of formally transforming an inefficient program into a more efficient one have been widely studied within the functional language framework (see Darlington 1982; Bird 1983).

(*d*) *The knight's tour*

As our final example we consider the well known problem of the knight's tour. We have to find a sequence of positions on a chess board such that each position after the first is a knight's move from the previous one, and every square on the board is visited exactly once. We here exhibit four solutions of progressively increasing efficiency. Our first program is a more or less literal transcription of the problem statement:

```
tour = any {t ← perms squares | istour t}
istour t = and [knights_move (t!i) (t!(i+1)) | i ← [1 . . # t − 1]]
knights_move (a, b) (c, d) = ((a − c) ↑ 2 + (b − d) ↑ 2 = 5)
squares = {(a, b) | a, b ← [1 . . 8]}.
```

This is a 'British Museum' algorithm, which proceeds by searching blindly among all factorial (64) candidate tours, looking for ones that consist only of knight's moves. Our first refinement is to replace this program by a solution that builds the tours one move at a time, considering *only* knight's moves. We introduce a function 'gentours', such that if t is a partial tour (i.e. a list of zero or more squares separated by knight's moves, and with no square represented more than once) then 'gentours t' is the set of all maximal tours that can be obtained by extending t. A tour is maximal if there is no unvisited square within a knight's move of the last square added to the tour. We introduce also a function 'nextmoves', which when applied to a tour

returns the set of all unvisited squares within a knight's move of the last square added. By convention, if 'nextmoves' is applied to the empty tour it returns the set of all squares on the board:

```
tour = any {t←gentour [ ] | # t = 64}
gentours t = [t], nextmoves t = [ ]
           = union {gentours (x:t) | x←nextmoves t}
nextmoves [ ] = squares
nextmoves (x:t) = {x1←moves x | legal x1 & ~ member t x1}
moves (a, b) = {(a+i, b+j) | i, j←[−2..2]; i↑2+j↑2 = 5}
legal (a, b) = 1 ⩽ a ⩽ 8 & 1 ⩽ b ⩽ 8.
```

The definition of 'squares' is the same as in the earlier solution and 'member' is a library function of type $[*]\rightarrow(*\rightarrow\text{bool})$. This second solution has brought about a huge reduction in the size of the space to be searched, and is much closer to being practical. It still proceeds by trial and error, however, in that it considers at each stage all possible ways of extending the given tour by a knight's move in any direction. In our next refinement we replace this by a deterministic procedure in which at each stage we choose, out of the set of possible next squares, one particular square, by using a heuristic rule.

The heuristic we use can be explained as follows. Let us define the 'freedom' of an unvisited square in a partly toured board, as the number of unvisited squares a knight's move away from it. A square with a low freedom is in danger of being 'boxed-in' (i.e. becoming surrounded by visited squares, so that it can no longer be reached, making a successful completion of the tour impossible) and such a square should therefore be visited early, in preference to a square with a higher freedom. The heuristic rule then is that at each stage, out of the set of possible next squares for the knight to visit, we choose one for which the 'freedom' takes a minimum value.

So in our next solution the function 'gentours t', which returns the set of all maximal extensions of t, is replaced by a function 'gentour t', which returns just one extension of t, namely that obtained by applying the heuristic repeatedly until the tour can be extended no further. We are now relying on a specialised piece of knowledge about the problem, which is that the heuristic rule just given always succeeds in generating a complete tour, regardless of the starting square. This is by no means an obvious fact, but it is a fairly well-known piece of folklore about this problem, and it therefore seem legitimate to use it. Thus we obtain for our next refinement of the program:

```
tour = gentour [ ]
gentour t = t, nextmoves t = [ ]
          = gentour (x:t)
            where
            x = minimise freedom (nextmoves t)
            freedom x = # nextmoves (x:t).
```

We have omitted the definition of the function 'nextmoves', since it is exactly as in the previous version of the program. The function 'minimise' takes a function and a set (list) and returns a member of the list for which the function assumes a minimum value; it is in fact a MIRANDA library function and we omit its definition. Notice that in the above we have allowed the

heuristic to determine our starting square also. This will have the effect of causing us to start in one of the corners of the board, because in a completely unvisited board the freedom takes its lowest value at the corner squares.

The program given has the makings of an efficient solution. It is, however, suboptimally coded, because for each partial tour t on the path to the complete tour, 'nextmoves t' is calculated three times. Our last step is to eliminate this repeated calculation of 'nextmoves'. We modify 'gentour' so that it takes a pair, consisting of a tour together with the set of next moves available from it. Making this last refinement we arrive at our final and efficient program:

```
tour = gentour (squares, [ ])
gentour (m, t) = t, m = [ ]
               = gentour (m1, t1)
                 where
                 (m1, t1) = minimise freedom {(nextmoves(x:t), x:t) | x←m}
                 freedom (m, t) = #m.
```

In this program the definition of 'squares' and 'nextmoves' are as in previous refinements.

4. Observations and concluding remarks

We have outlined a style of functional programming based on recursion equations and demonstrated its use in constructing executable solutions to several small but non-trivial programming problems. Compared with the more traditional style of functional programming, as represented by LISP, we can see the strengths of our notation as being that it permits the introduction of user-defined types by what are in effect recursion equations over data domains; that it is higher order and supports functions as data objects in a fully extensional way; that it has a polymorphic type-discipline whereby type errors can be detected at compile time while still permitting generic functions and generic data types; that it permits the definition of infinite data objects of many different kinds; and that it has a facility for set-abstraction.

We have seen that by using the set notation it is often possible to write down, in a more or less direct transcription of the problem statement, a solution that is executable in principle although perhaps extremely inefficient. From this we can then develop more efficient solutions, either by a process of formal transformation, or by a series of refinements based on informal reasoning. The solutions we arrive at in this way are usually very much more concise than would be the expression of the corresponding solution in a conventional imperative programming language.

If we draw a distinction between the basic idea of an algorithm, and the housekeeping details needed for its efficient implementation on a Von-Neumann computer, we can say that a language of the kind used in this paper gives us a way to capture the former without getting entangled in the latter. The other important advantage that we have over an imperative language is that the static, equational, style in which our programs are written makes very much more straightforward the task of inferring program properties from the program text.

The efficiency of functional programs compared with imperative ones is still to some extent an open question (on which we shall comment briefly), but even if we make pessimistic assumptions about this, a functional programming language can still be a useful tool in software

production. By using such a language it is often possible to construct a working piece of software, and indeed to experiment with a series of different designs for it, at a small fraction of the cost in programmer time that would be required in a language like PASCAL. That is to say, even without an efficient implementation, a functional programming system could earn its keep in an industrial environment as a rather high-level tool for software prototyping.

In the remaining sections of this conclusion we discuss briefly some of the outstanding research topics in functional programming.

(*a*) *Non-determinism*

There is of course a gap between the tidy and rather mathematically motivated problems of the kind for which we have exhibited solutions in § 3 of this paper, and the larger and somewhat messier problems that characterize production programming. It is legitimate to wonder whether it continues to be possible to program in a purely functional style when the size of the problem is scaled up. (In this regard we must discount previous experience with LISP, because it is in no sense a purely functional language.) In recent years a number of groups have shown that it is possible to write larger programs, such as editors, compilers and so on, within a purely functional language, and the author is currently engaged in a S.E.R.C. funded project at the University of Kent, to construct a complete operating system in a functional language essentially the same as the one used in this paper.

There is, however, one extension to the language that seems to be necessary to write an operating system (in particular, to program the scheduling of concurrent tasks), and this is a certain kind of non-determinism. In some of the programming examples given, we made use of a selection function 'any' to express indifference as to which element of a set was chosen. For the purposes for which we have used it up to now, 'any' does not need to be non-deterministic other than in a very weak sense; it is in fact just the name of a particular selection function, which in the definition of the language we do not choose to specify (but which in the current implementation happens to be 'hd'). To solve certain kinds of systems programming problems, however, we need a version of 'any' with the stronger property that when applied to a list not all of whose element are $\perp$, it will return one of the non-bottom elements.

It is fairly easy to see that a version of 'any' with this bottom avoiding behaviour, if it is to be computable, cannot be a function at all, but must instead be a non-deterministic operator. Although there is no particular difficulty in implementing an operator with the required properties, its presence means that we have a language in which expressions are no longer single-valued, a circumstance that greatly complicates both the semantics and the proof theory (see, for example, Broy 1981). At the time of writing there is no consensus as to what is the best way to handle these problems. It is to be hoped that further research will bring about some simplifications in this area.

(*b*) *Efficiency*

The other issue that arises if we consider the prospects for functional programming 'in the large' is that of efficiency. There are some quite encouraging results in this issue. A recent study of sorting algorithms by Meira (1983) shows that for each known imperative sorting algorithm, there exists a functional sorting program of the same fundamental (time) complexity. By the same fundamental complexity we mean that if the original algorithm was say, quadratic in the length of the input, it remains quadratic; it it was $n \log n$ it remains $n \log n$, and so on. For

example, the average case behaviour of the functional version of quicksort, which we gave at the end of §2, is, like its imperative counterpart, of $n \log n$ complexity.

On the basis of this and related work, the author is strongly tempted to conjecture that for time complexity, there is no *fundamental* difference between imperative and applicative programming, i.e. that for each imperative algorithm of a given complexity there exists an applicative algorithm that has the same complexity (to within a multiplicative constant). We can here give only the briefest sketch of the argument in favour of this conjecture.

So far as the author is aware, all of the alleged counterexamples to this proposition depend on the presence, in imperative programming languages, of constant-time access data structures, such as the PASCAL *array* type. The absence of a constant-time access data structure from functional languages is, however, a matter of tradition and not of logical necessity. Furthermore there are ways to achieve, within the functional framework, the same effect as the imperative notion of 'update-in-place' on such data structures (see Meira 1984). This would appear to give us the basis for a general method of transcribing an imperative algorithm into a functional one while preserving its complexity (within a multiplicative constant).

The above remarks concern time complexity. For the space complexity of functional programs the situation is much less clear. In a language with lazy evaluation it is in general rather difficult either to predict or to control the space behaviour of programs. In fact Hughes (1984) has shown that there are some very simple problems for which it is surprisingly difficult to construct functional solutions with a reasonable space behaviour. This is an area where further research is needed.

Notwithstanding these difficulties there is much promising work now in progress on the design of machine architectures and instruction sets for the efficient support of functional languages and it seems probable that in the future we shall have implementations of purely functional languages efficient enough to allow their use in production programming.

(*c*) *Relation to logic programming*

Functional programming, as presented here, and logic programming in the style of PROLOG, as presented by Professor Kowalski and others (Kowalski this symposium), are obviously closely related, and both may be regarded as species of declarative programming. In both cases the program consists of *assertions*, and the computation proceeds by a process of deduction from those assertions. For functional programming the assertions consist of *equations*, for logic programming they consist of *material implications*. For the former the subject of discourse is *functions*, for the latter it is *relations*.

Each of these two styles of programming has advantages not possessed by the other. An important advantage of logic programming is that because it is based on relations, it can accommodate non-determinism in a straightforward way. Functional programming, however, has the important advantage of being *higher order*, i.e. it permits the manipulation of functions as data objects, in a way that respects the principle of extensionality, whereas nothing quite equivalent to this exists in logic programming.

At the present stage of our knowledge of logic programming and functional programming, it seems that neither can be regarded as a subject of the other, and that further research is required into both, and into the relation between them. It would be very desirable if we could find some more general system of assertional programming, of which both functional and logic programming in their present forms could be exhibited as special cases.

References

Bird 1983 Some notational suggestions for transformational programming. *Tech. Rep.* no. 153. University of Reading.

Broy, M. 1981 A fixed point approach to applicative multiprogramming. In *Proceedings of Nato summer school on theoretical foundations of programming methodology, Munich*, pp. 565–624. Dordrecht: Reidel.

Burstall, R. M. 1969 Proving properties of programs by structural induction. *Computer J.* **12**, 41–48.

Burstall, R. M., MacQueen, D. B. & Sannella D. T. 1980 HOPE, an experimental applicative language. *Univ. Edinburgh tech. rep.* no. CSR-62-80.

Curry, H. B. & Feys, R. 1958 *Combinatory logic*, vol. 1. Amsterdam: North-Holland.

Darlington, J. 1982 Program transformation. In *Functional programming and its applications* (ed. J. Darlington, P. Henderson & D. A. Turner), pp. 193–215. Cambridge University Press.

Davis, M. 1958 *Computability and unsolvability*. New York: McGraw-Hill.

Dijkstra, E. W. 1976 *A discipline of programming*. Englewood Cliffs, N.J.: Prentice-Hall.

Drake, F. R. 1974 *Set theory*. Amsterdam: North-Holland.

Friedman, D. P. & Wise, D. S. 1976 CONS should not evaulate its arguments. In *Proceedings 3rd International Colloquium on Automata Languages and Programming*, pp. 257–284. Edinburgh University Press.

Gordon, M. J., Milner, R. & Wadsworth, C. P. 1979 Edinburgh LCF. *Lecture notes in computer science*, vol. 78, Berlin: Springer-Verlag.

Graham, N. 1983 *Introduction to PASCAL*, p. 202. Minnesota: West.

Henderson, P. 1982 Purely functional operating systems. In *Functional programming and its applications* (ed. J. Darlington, P. Henderson & D. A. Turner), pp. 177–192. Cambridge University Press.

Henderson, P. & Morris, J. M. 1976 A lazy evaluator. In *Proceedings 3rd Symposium on principles of programming languages, Atlanta Georgia*, pp. 95–103.

Hoare, C. A. R. 1962 Quicksort. *Computer J.* **5**, 10–15.

Hughes, J. M. 1984 The design and implementation of programming languages. D.Phil. thesis. Oxford University.

Inmos Ltd 1982 *OCCAM programming manual*. Bristol: Inmos Ltd.

Kahn, G. & McQueen, D. 1977 Coroutines and networks of parallel processes. In *Proceedings IFIP 77*, pp. 993–998. Amsterdam: North-Holland.

Kleene, S. C. 1952 *Introduction to metamathematics*. Amsterdam: North-Holland.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. & Levin, M. I. 1962 *The LISP 1.5 programmer's manual*. M.I.T. Press.

Meira, S. R. L. 1983 Sorting algorithms in KRC: implementation, proof and performance. *Computing Laboratory rep.* no. 14. University of Kent at Canterbury.

Meira, S. R. L. 1984 A linear applicative solution to the set union problem. *Computing Laboratory tech. rep.* no. 23. University of Kent at Canterbury.

Milner, R. 1978 A theory of type polymorphism in programming. *J. computer system Sci.* **17**, 348–375.

Scott, D. S. 1970 Outline of mathematical theory of computation. Oxford University Programming Research Group, *tech. monogr.* no. 2.

Scott, D. S. 1976 Data types as lattices. *SIAM Jl Computing* **5**, 522–587.

Tarjan, R. E. 1972 Depth first search and linear graph algorithms. *SIAM Jl Computing* **1**, 146–160.

Turing, A. M. 1936 On computable numbers, with an application to the Entscheidungs problem. *Proc. Lond. math. Soc.* **42** (ser. 2), 230–265.

Turner, D. A. 1976 SASL language manual. *St Andrew's University tech. rep.* no. CS/75/1.

Turner, D. A. 1981 Aspects of the implementation of programming languages. D. Phil thesis, Oxford University.

Turner, D. A. 1982*a* Functional programming and proofs of program correctness. In *Tools and notions for program construction* (ed. D. Néel) pp. 187–209. Cambridge University Press.

Turner, D. A. 1982*b* Recursion equations as a programming language. In *Functional programming and its applications* (eds. J. Darlington, P. Henderson & D. A. Turner), pp. 1–28. Cambridge University Press.

Turner, D. A. 1984 The type structure of MIRANDA. (In the press.)

Discussion

J. Fairbairn (*Cambridge University Computer Laboratory, U.K.*). I am unhappy with the way that in miranda the existence of an equality function is assumed. We all know that equality is not computable for several types of object; infinite trees and arbitrary functions being examples. Hence the function 'assoc' of type $(* \times * * \text{list}) \rightarrow * \rightarrow * *$ cannot work unless the type substituted for '$*$' is a finite list or product of zero-order (non-function) types. This restriction is rather subversive, since the type of 'assoc' suggests that it should work for any kind of argument.

Instead, 'assoc' could be parameterized on the equality function, and would have type $(*\to *\to \text{bool})\to(*\times *\,*\,\text{list})\to *\to *\,*$. Similarly, functions for sorting may be parameterized on the order relation chosen, and so the same function could be used to sort into ascending or descending order. Surely this constitutes a more general approach, and does not subvert the type structure?

D. A. Turner. MIRANDA, like the earlier languages SASL and KRC, provides a built in equality test. This is a three valued function, yielding *true*, *false* or $\perp$. On discrete data objects it is always defined, and has an obvious and natural meaning. When applied to, say, a pair of functions it must yield $\perp$, because, as Dr Fairbairn points out, there is no reliable way of testing for equality on objects like that. I do not feel that there is anything illogical or subversive about this. A built-in equality test is a very useful thing to have, and the fact that it cannot be total (i.e. everywhere defined) does not seem to me to be a reason for not having it at all.

One might have wanted to have a type discipline that prevented people from applying the equality test to, say, functions. This is a perfectly reasonable thing to want. It is not possible within the Milner type-discipline, however, because there is only one kind of type variable and it ranges over all types. One cannot have a kind of type variable that ranges over, say, only discrete types. I decided to use the Milner type-system for MIRANDA because it is relatively simple and well understood, and does more or less exactly what I want. I have no objection, however, to other people experimenting with more complex and subtle type systems, however, such as I believe Dr Fairbairn is doing with his language PONDER.

With regard to Dr Fairbairn's proposal that the user should pass in the comparison function as a parameter, this is of course equally possible in MIRANDA and I do not see any conflict here. One could have a version of 'assoc' that works in the way Dr Fairbairn suggests. For example there is a built-in ordering relation (tested by '$>$' '$<$', etc.) on all finite data objects; it does the obvious thing on numbers and strings, and imposes some arbitrary but reproducible order on other types. The library function 'sort' sorts with respect to this built-in ordering. There is also a function 'sortwith', which takes a comparison function provided by the user as a parameter and sorts with respect to that. Having built-in comparison operators in no way prevents one from using the more general approach when it is appropriate.

D. Park (*Computer Science Department, Warwick University, U.K.*). I would like to draw attention to what appears to be an essential difficulty in the choice of primitives for programming with infinite sets in languages like MIRANDA.

(1) If the empty set $\varnothing$ is uniquely representable in MIRANDA, then the Scott approximation

$$\varnothing \sqsubseteq X$$

should hold between it and all other 'sets' X. Otherwise (assuming the usual relations between computability and lattice structure) the question

$$\varnothing = Y$$

would be decidable for MIRANDA expressions Y, which include arbitrary effective enumerations of sets. This would contradict well known computability results. (A typical expression that seems to demand such a 'non-terminating' empty set might be as intersect $(X,\{x\})$, with X an infinite set not containing x.)

(2) Using the monotonicity of (1) in the usual way, we should have

$$0 = \text{any}\,(\{0\}) = \text{any}\,(\text{union}\,(\{0\}, \varnothing) \sqsubseteq \text{any}\,(\text{union}\,(\{0\}, \{1\})$$

and similarly

$$1 \sqsubseteq \text{any}\,(\text{union}\,(\{0\}, \{1\}).$$

The only interpretation of these two inequalities appears to be that the expression any (union ({0}, {1}) is in some sense essentially non-deterministic. Whether 0 or 1 is produced depends on unspecified detail as to the timing of the processes that supply them.

Apparently, therefore, a system of primitives that allows both the 'choice' function any (X) and the conventional union (X, Y) must include a 'non-deterministic' notion, which violates the constraints needed for the usual reasoning with equations.

The reasoning involved appears to be very general. By taking 'sets' to be sequences of their elements, the non-deterministic notion is union (X, Y), which merges sequences 'angelically'. By factoring out sequencing, the non-determinism moves to the choice function any (X).

D. A. Turner. The logic of Professor Park's argument seems inescapable: any programming language with a proper implementation of infinite sets in it and a choice function must involve non-determinism. I was in fact aware of this, but I am very grateful to Professor Park for providing us with an argument that exposes the problem in a very clear way.

In the design of miranda I tried to postpone dealing with this problem by a decision *not* to introduce sets as a separate data type. Sets are just lists, from which repetitions have been removed. This enables the choice operator 'any' to be implemented in a deterministic way (in fact 'any' is just 'hd'). By the logic of Professor Park's argument, the problem must therefore shift to 'union'. Let us look at union then to see what difficulty arises.

The miranda library function 'union' is defined (for simplicity I show it as the union of two sets, rather than of a list of sets, which is now it is actually defined) as

```
union x y = mkset (interleave x y)
interleave x [ ] = x
interleave [ ] y = y
interleave (a:x) (b:y) = a:b:interleave x y.
```

The function 'mkset' removes repetitions from a list; there is no problem about this (provided the list elements are finite objects) and its definition need not detain us here. The problem is with 'interleave'. There are three kinds of list that could occur as arguments to interleave: infinite lists like [1 . .]; ordinary finite lists like [1, 2]; and finite *partial* lists like $1:2:\perp$. (Note then that in a lazy functional language, finite sets have two distinct kinds of representations over the last few years I have fallen into the habit of calling these 'strong' and 'weak' sets respectively.) Unfortunately one cannot always arrange that a finite set will be represented by a list of the second kind, representations in terms of lists of the third kind are sometimes inevitable.

The definition of union given works correctly both on infinite sets and on 'strong' finite sets, but does not in general give correct answers on 'weak' sets. This is less than fully satisfactory, but it seems to be the best one can do without introducing non-determinism. To get a definition of union that also works correctly on weak sets we would need a version of interleave that is

bottom-avoiding (symmetrically in its two arguments). This would obviously involve non-determinism (of precisely the kind that we probably need anyway to write operating systems).

So MIRANDA in its present form does provide a representation of infinite sets with a deterministic choice function, but at the cost of having an incomplete version of set union. Curiously, however, the difficulty with set union does not concern infinite sets but finite ones (of a certain kind). To overcome this problem would require the introduction of non-determinism.

The reason that non-determinism seems undesirable, and the reason that I am trying to avoid introducing it until I am forced to do so (by some application that cannot be done without it), is that, as Professor Park says, it violates 'the constraints needed for the usual reasoning with equations'.

P. WADLER (*Programming Research Group, Oxford, U.K.*). First, an important language that Professor Turner ignores is LISP. I have written a moderate-sized program in both LISP and KRC (the predecessor to the language used in the talk). The program was about 40 pages in LISP and about 4 pages in KRC, which speaks well for KRC. The main reasons for brevity were the syntax of KRC and the use of lazy evaluation. Set expressions did not contribute much to brevity; I tended to avoid them in favour of the standard MAP function.

Second, Professor Turner mentioned 'serious' uses of functional languages. Some very efficient implementations of LISP have been developed. One version, for the S-1 computer used at the Lawrence Livermore Laboratories, may be more efficient than the FORTRAN provided on the same machine. This makes it quite possible that LISP will be used for designing nuclear weapons, which is a very serious use indeed.

D. A. TURNER. I am certainly aware that there has been a great deal of experience built up, over the last twenty years or so, of programming in LISP and that there exist by now some quite efficient implementations of it. Unfortunately, however, this experience has very little bearing on the issues of functional programming. It needs to be said very firmly that LISP, at least as represented by the dialects in common use, is not a functional language at all. LISP does have a functional subset, but that is a rather inconvenient programming language and there exists no significant body of programs written in it. Almost all serious programming in LISP makes heavy use of side effects and other referentially opaque language features.

I think that the historical importance of LISP is that it was the first language to provide 'garbage-collected' heap storage. This was a very important step forward. For the development of functional programming, however, I feel that the contribution of LISP has been a negative one. My suspicion is that the success of LISP set back the development of a properly functional style of programming by at least ten years.

B. A. WICHMANN (*National Physical Laboratory, Teddington, U.K.*). Has Professor Turner any results on the space efficiency of his techniques?

D. A. TURNER. There has been rather little work done in this area, so far as I am aware. We have some quite encouraging results on the time complexity of functional programs, as I mentioned, but for space complexity the picture is much murkier. It is in general rather hard to analyse the space behaviour of 'lazy' languages, but one can give some simple examples

where the space behaviour of a lazy functional program is much worse than that of the corresponding imperative program. There are several possible lines of attack on this problem, but much more work needs to be done. It may be that we shall discover that there are classes of problem for which functional languages have fundamentally worse space behaviour than imperative ones. I very much hope that this is not so, but in our present state of knowledge we cannot discount it. Of course memory is getting cheaper very rapidly, so such a discovery would not necessarily be disastrous, although it would be bound to limit the applicability of functional techniques to applications that are not space-critical.

M. H. Rogers (*School of Mathematics, University of Bristol, U.K.*). How do functional programming languages compare with procedural and logical languages in (i) ease of programming, (ii) proving correctness of some specification expressed in ordinary mathematical language?

D. A. Turner. What one means by 'ease of programming' is rather hard to define, so it is difficult to make precise statements about this. However, I observe from my own behaviour and that of others, that the time required to produce an executable solution to a given problem (assuming for the moment that the level of performance of the solution is not an issue) is in general very much less when working in a functional language than in a conventional imperative one. I observe from the behaviour of others that logic programs have the same property. This presumably is one measure of 'ease of programming'.

On the other hand one can ask about the prerequisites on the programmer, in terms of length of training, level of previous knowledge required and so on. It seems to me that to learn to program well in a functional language requires a rather longer period of training and a more mathematical knowledge than is commonly expected of programmers today. The advantage is that programmers become more productive.

With regard to a comparison between functional programming and logic programming I am probably the wrong person to ask, since I have much more experience of the former than the latter and any answer I could give might only reflect this bias.

Of the three types of programming Professor Rogers mentioned, functional programming is much the closest to ordinary mathematics, because it is based on equations. This allows a rather nice style of proof, by use of the substitutivity of equality, which has been explored by myself and others, and which I have written about elsewhere.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES