Meaning and Process in Mathematics and Programming

PETER GROGONO

1. Introduction

The common theme of several papers published recently in *For the Learning of Mathematics* is that children experience difficulties of various kinds while learning to write programs and that teachers need new strategies to help them overcome these difficulties [3, 5, 6, 7, 9, 10]. Naturally enough, the authors of these papers implicitly assume that "learning to program" means learning to program in the languages currently used in schools: BASIC, LOGO, and Pascal. Since programming languages are my principal field of study, my perspective is different. When I read about learning problems related to programming, my response is "what is wrong with the language?"

In this paper, I describe some recent trends in programming language design. My purpose is to inform, not advertise. I do not claim to have answered the question "which kind of language should be used to teach mathematics?" To answer this I would have to know why mathematics is taught at all. Is mathematics a bag of tricks that may be useful in later life? Is mathematics taught because it is an important part of our culture or because it helps young people to think logically and abstractly? These are questions for mathematics teachers. In the long run, computer software can be adjusted to their requirements.

Both mathematics and computer science provide us with at least two views of the world. The first is an abstract, static view. Some statements are true and others are false, but truth values do not change. The value of a function depends on the values of its arguments and nothing else. A circle is the set of points (x, y) that satisfy the equation $x^2 + y^2 = R^2$ for some constant R.

The other view is concrete and dynamic. Things change. The state of the world and the variables of a program depend on the time at which we look at them. A circle is a curve of constant curvature traced by an object moving at uniform speed whose direction changes at a constant rate.

Mathematics and computer science provide tools for both these ways of thinking. While the industrial revolution saw important advances in both mathematics and its applications, the concern with the foundations of mathematics that began during the nineteenth century and has continued until the present day has put undue emphasis on the static nature of mathematics. This applies particularly to the mathematics that we teach. The old idea of "moving towards a limit" has been replaced by the frozen values of δ and ϵ.

Seymour Papert sees computer programming as a way of bringing motion back into mathematics [8]. Partly as a result of his work, many teachers assume that computers are capable of demonstrating only the dynamic component of mathematics. Papert's view of computing is based on what computer scientists call *procedural*, or *imperative*, programming. He considers a program to be a *sequence of instructions* telling a turtle and, by extension, a computer, what to do. The problem with this viewpoint is that it makes programming unnecessarily difficult. Although the idea of a sequence of instructions is intuitively appealing, it leads to difficulties in reading, debugging, and reasoning about programs. If programming is to be introduced into the mathematics curriculum, there should be a relationship between programs and simple mathematical concepts.

Papert considers *debugging* to be an important component of programming. The ability to correct errors systematically is useful in many disciplines including mathematics, and computer programming provides a useful context in which it can be learned. It is helpful, however, to distinguish between errors caused by failure to understand the problem and errors caused by deficiencies in the programming language. My hypothesis is that many of the difficulties children have with programming are due to the language, not the problem. My purpose in this article is to discuss the drawbacks of sequences of instructions as a programming paradigm and to introduce other paradigms that do not have these drawbacks.

2. Imperative programming

Most of the well-known programming languages, such as BASIC, LOGO, FORTRAN, and Pascal are *imperative* or *procedural* languages. A program is a *sequence of instructions* that are executed by the computer. The languages are called "imperative" because a program consists of a sequence of *commands* or *instructions*, or "procedural" because a program is a *procedure* for achieving a given effect. Writing a program in an imperative language is rather like telling a person how to do something. Papert captured this aspect of programming very successfully when he introduced the turtle into LOGO. Beginners can debug their programs by pretending to be turtles and obeying the instructions of the program.

The problem with imperative languages is that it is difficult to reason about programs that are written in an imperative style. Particularly in a mathematics curriculum, we would like to be able to prove that a program performs the task that it is supposed to perform, to

For the Learning of Mathematics 9, 1 (February, 1989)
FLM Publishing Association, Montreal, Quebec, Canada

manipulate programs like algebraic expressions, and to demonstrate that two different programs perform the same task. If we cannot do any of these things for even the simplest examples, we cannot incorporate programming into a mathematical curriculum in a reasonable way.

There is a mathematical theory of imperative languages in which reasoning about programs is possible, but it is beyond the reach of most undergraduate students, let alone school students. Consquently, programs are indigestible lumps. They may or may not work, and discussions about meaning and correctness are just so much hand waving.

Computer scientists have long been aware of these problems. For many years, they have looked for languages which have pleasant mathematical properties and which can be executed efficiently. Three approaches dominate.

Procedural languages have matured into *object-oriented* languages. The best known object-oriented language is Smalltalk, but there are already many others. When we write a program in an object-oriented language, we think of a collection of *objects*. Each object belongs to a *class*, which determines the properties of the object. Classes are arranged in a hierarchy which provides a taxonomy of properties.

Although object-oriented languages are in some ways an improvement over imperative languages, there is as yet no simple theory for them. The other two approaches, which we consider in more detail, are *functional programming* and *logical programming*. Both of these styles can be explained in terms of simple and familiar mathematics.

3. Functional programming

The concept of functional dependency is a cornerstone of mathematics. We can think about a function $f: A \rightarrow B$ in two ways. Extensionally, it is a subset of the Cartesian product $A \times B$ that satisfies certain properties. Intensionally, it is a rule that enables us to compute a value from any given value in A.

Any program can be viewed as a function. We simply write $Y = f(X)$, where X and Y are vectors representing the input data and the results, respectively, and f is a function representing the program. The input, X, uniquely determines the output, Y. If this is not the case, we have omitted something from X. For example, if the program contains a random number generator, X must include the source for the generator, whether it is the time of day or a lump of radium.

The analogy with functions does not help unless we can decompose programs into functional components. A functional program consists of expressions which satisfy two conditions. First, the value of an expression depends only on the values of its free variables. Second, an expression has no semantic properties other than its value.

These conditions are violated by the assignment, loop, input, and output statements of conventional imperative languages. People accustomed to using imperative languages are unwilling to believe that programming is possible without these statements. To show that we can indeed write useful programs in a purely functional style, I will give a brief description of the language Miranda.

David Turner, the designer of Miranda, has been interested in functional programming for many years. He designed his first functional language, St Andrews Static Language (SASL) in 1974 [12]. Miranda is his most recent functional language [13].

Miranda turns the computer into a kind of super calculator. If we enter an expression, Miranda replies with the value of the expression. I will avoid verbosity in the examples by using "→" to mean "evaluates to". For example, if you enter "3 + 5", Miranda replies "8". I will write this computation as:

$$3 + 5 \rightarrow 8.$$

Miranda also accepts function definitions in a style quite close to conventional mathematical usage. We could define a squaring function by writing

$$sq\ x = x * x$$

and then use it:

$$sq\ 3 \rightarrow 9.$$

For a function which has different definitions in different parts of its domain, Miranda provides definitions with multiple clauses. In the following definition of the absolute value function, the conditions are written after the values, as in mathematics.

$$\begin{aligned} abs\ x &= x,\ x \geqslant 0 \\ &= -x,\ x < 0 \end{aligned}$$

When a function has particular values at certain points in its domain, we can specify these. The following definition of the factorial function illustrates both this feature and recursion.

$$\begin{aligned} &fac\ 0 = 1 \\ &fac\ n = n * fac(n - 1),\ n > 0 \end{aligned}$$

Although Miranda has a complete set of data structures, I will describe sequences only. There are two notations for sequences. The expression [1, 3, 5] is a sequence of three numbers. Miranda provides abbreviations for arithmetic progressions. The sequence 0, 2, ..., 20 can be written in the form [0, 2, ..., 20]. The empty sequence is written [].

For programming purposes, we consider a non-empty sequence to consist of a *head* and a *tail*. The sequence [1, 3, 5] has as head the integer 1 and as tail the sequence [3, 5]. The expression $a:x$ denotes a sequence with head a and tail x. The function *len* returns the length of a squence and the function *append* returns the result of concatenating two sequences.

(1) *len* [] = 0
(2) *len* (a:x) = 1 + *len x*
(3) *append* [] *y* = *y*
(4) *append* (a:x) *y* = *a* : (*append x y*)

The importance of functional programming is that the definitions can be interpreted in two ways. Consider the function *len* defined by (1) and (2). We can understand the two parts of the definition declaratively as equations:

(1') The length of the empty sequence is zero.

(2′) The length of the non-empty sequence $a:x$ is one more than the length of x.

Alternatively, we can understand the definitions as computational rules.

(1″) To compute the length of the empty sequence, compute zero.

(2″) To compute the length of the non-empty sequence $a:x$, compute the length of x and add 1 to the result

The functions *sum* and *prod* return the sum and product of a sequence of integers, respectively.

```
sum [] = 0
sum (n:x) = n + sum x
prod [] = 1
prod (n:x) = n * prod x
```

It should be clear by now that Miranda makes extensive use of recursion. Since we cannot use loops, we have to use recursion for any kind of repeated computation. In Miranda, as in most functional languages, we can define higher-order functions. We can use these to conceal many instances of recursion. The only difference between the functions *sum* and *prod*, for example, is that *prod* has "*" and "1" where *sum* has "+" and "0". We can abstract the operator and unit out of the definition, obtaining a function which "folds to the right".

```
foldr f u [] = u
foldr f u (a:x) = f a (foldr f u x)
```

We can use *foldr* with any binary function f with a unit u which satisfies $f(x, u) = x$. We can redefine *sum* and *prod* using *foldr*:

```
sum = foldr (+) 0
prod = foldr (*) 0
```

and we can also define, for sequences of Boolean objects,

```
forall = foldr (and) True
exists = foldr (or) False
```

It might appear from what we have seen that functional programming is rather dull. We are limited to providing some data, running the program, and looking at the results. Interaction seems to be impossible. It turns out, however, that, we can use sequences to write interactive programs.

Miranda evaluates as many components of a sequence as are required and no more. We can define infinite sequences, such as

```
from n = n : from (n + 1)
```

If we enter *from* 1, Miranda will respond by typing 1, 2, 3, ..., continuing until we interrupt it. A function can accept an infinite sequence as its argument and return an infinite sequence as its value. For example, we could define

```
squares (n:x) = (sq n) : squares x
```

and then enter

```
squares keyboard
```

The object *keyboard* appears to the system as a stream of numbers. The effect is that each time we enter a number, Miranda responds by typing its square. This is a simple, interactive program.

A general functional program is a function which accepts one or more infinite sequences as its arguments and returns one or more infinite sequences as its value. Unlike imperative programs, functional programs have many of the properties that we associate with algebraic expressions. In particular, we can replace any expression by another expression with the same value. The following example is due to Darlington [1].

The function *len*, defined by equations (1) and (2), gives the length of a sequence. The problem is to define a function, *twolen*, which computes the combined length of two sequences. For example,

```
twolen [1, 3, 5] [2, 4] → 5
```

We can easily define *twolen* in terms of functions that we already know:

(5) `twolen x y = len (append x y)`

This definition, which follows directly from the specification, is clearly correct. Correctness is important, but it is not the whole story. Although it is simple and concise, *twolen* constructs a new sequence for the sole purpose of finding its length. Can we avoid this? Suppose that the first argument of *twolen* is the empty sequence, []. Treating the function definitions as equations, we have

```
twolen [] y = len (append [] y)      from (5)
            = len y                  from (3)
```

The other possibility is that the first argument of *twolen* is a non-empty sequence, $a:x$.

```
twolen (a:x) y = len (append (a:x) y)      from (5)
               = len (a : (append x y))    from (4)
               = 1 + len (append x y)      from (2)
               = 1 + (twolen x y)          from (5)
```

We have eliminated *append* from the definition (5) and obtained the following more efficient definition.

```
twolen [] y = len y
twolen (a:x) y = 1 + (twolen x y)
```

The derivation of *twolen* exploits the dual interpretation of a functional program. Imperative languages do not have dual interpretations and we cannot manipulate programs in this way.

The concept of dual interpretation is too sophisticated to be used at an introductory level. Fortunately, we do not need it. We can explain functional programming by saying that the computer can solve equations provided that the equations are expressed in a certain way. We are allowed to manipulate the equations using conventional algebra, provided that the equations we derive are expressed in the appropriate way.

Many teachers believe that assignments and loops are easy to understand but that recursion and higher-order functions are difficult to understand. If we learn to "think like a computer", simulating each step of a computation, it is indeed easier to understand an imperative program

than a functional program. But bringing our way of thinking down to the level of a computer is surely a step backwards, not forwards.

4. Logic programming

Logic programming, like functional programming, depends on a dual interpretation, but we write predicates instead of equations. suppose for instance, that we write

$$2x^2 - 8 = 0.$$

Although this equation implicitly defines two values of x, it is not a functional program because it does not conform to the rules for computation. We can, however, view it as a predicate in a logic program. It becomes a statement in predicate calculus when we quantify it

(6) $\exists x.2x^2 - 8 = 0.$

The logical meaning of (6) is "there are one or more values of x such that $2x^2 - 8 = 0$", a statement that may or may not be true. As a computation, (6) means "find values of x such that $2x^2 - 8 = 0$".

There is another way of looking at logic programming. Each program states a theorem about the existence of certain objects. The task of the computer is to find a constructive proof of the theorem and to present the objects that satisfy it.

Whereas the goal of an imperative or a functional program is a single solution, the goal of a logic program is to find all solutions that satisfy the predicate. Program (6), for instance, has solutions $x = 2$ and $x = -2$.

A logic programming language is usually sound but not complete. That is, any solution obtained by the program *P(x)* certainly satisfies *P*, but there may be values of x which satisfy P but which the program cannot find. It is not enough to understand predicate logic: we must also know something about the capabilities of the proof strategy.

The most popular and well-known logic language is Prolog. From the point of view of this paper, however, Prolog has a number of drawbacks. Its proof strategy is rather weak and, to compensate for this weakness, the designers of Prolog added "extralogical" features. Prolog programmers have to choose between rather restricted inference rules and going outside the logical semantics of the language. Prolog cannot solve programs with equational constraints, such as (6).

My discussion of logic languages is based on a relatively new language called Trilogy [2, 11]. Trilogy uses a variety of decision procedures to satisfy predicates, but it remains faithful to the underlying logic. Trilogy may fail to find a solution, but it will never find an incorrect solution.

Trilogy is an interactive language. If we write a query, which is a predicate with free variables, Trilogy will respond by displaying values of the variables which satisfy the predicate, if it can find any.

Trilogy is a typed language, like Pascal. The predicate $x::L$ says that the possible values of x are long integers, that is, integers with an absolute value of less than 2^{31}. The predicate $x::L[1..10]$ restricts x to the values 1, 2, ..., 10. This predicate is a valid query with solutions $x = 1$, $x = 2$, ..., $x = 10$. We can add further constraints. For example, the query

$$x::L[0..] \;\&\; 17 * x < 1000$$

finds multiples of 17 that are less than 1,000. The symbol "&" is a logical and, "*" denotes multiplication, and "<" has its conventional meaning. The query

$$x::L[1..10] \;\&\; y::L[1..10] \;\&\; z::L[1..10] \;\&\; x * x + y * y = z * z$$

finds all Pythagorean triangles with sides up to 10 units long. The symbol "=" denotes equality in the mathematical sense, not assignment. Both "&" and "=" are commutative operators.

A Trilogy program either *succeeds* or *fails*. If it succeeds, it yields values of the free variables that satisfy the predicate. If there are no such values, the program fails.

We can define predicates in Trilogy just as we can define functions in Miranda. The following predicate is satisfied when y is the absolute value of x. The symbol "|" denotes a logical or. Like "&", it is commutative.

$$\begin{aligned} &pred\ Abs\ (x::L, y::L)\ iff \\ &\qquad x \geqslant 0 \;\&\; y = x \\ &\quad | \; x < 0 \;\&\; y = -x \end{aligned}$$

To appreciate the various ways in which we can use *Abs* in a program, we must give it a context. Suppose that we write

(7) $P \;\&\; Abs(x, y) \;\&\; Q$

in which P and Q are predicates. The meaning of (7) is simple and unambiguous: it is a true statement if there are values of x and y such that P, $Abs(x, y)$, and Q are all true. The Trilogy processor, however, must process the expression from left to right. There are three cases to consider.

First we assume that when P has been processed, both x and y have received values. In this case, $Abs(x, y)$ acts as a test. If, for instance, P yields $x = -3$ and $y = 3$, *Abs* is satisfied and the processor moves on to Q with these values. If, on the other hand, P yields $x = 7$ and $y = 4$, *Abs* fails and the entire program fails.

The second possibility is that P binds one variable but not the other. In this case, *Abs* succeeds but imposes a constraint. For example, if P yields $x = 3$, then *Abs* will set y to 3. If P yields $y = 3$, *Abs* will constrain x to be either 3 or -3 during the evaluation of Q.

Finally, P might assign values to neither x nor y. In this case, *Abs* succeeds but passes on the constraint $y = |x|$ to Q. If P has already imposed constraints on x and y, this constraint will be added to the list of constraints passed to Q.

Constraint programming is a very powerful technique but, not surprisingly, it is expensive. Trilogy provides a number of ways of trading flexibility and efficiency. We can make our programs more efficient without losing soundness but they may not find as many solutions.

Suppose we know that the predicate P in (7) always provides a value for x. In this case, *Abs* can always find a unique value for y and the constraint mechanism is un-

necessary. We can inform Trilogy of this by writing *Abs* in the following form.

$$\begin{aligned} &pred\ Abs\ (x{:}{<}L,\ y{:}{>}L)\ iff \\ &\qquad x \geqslant 0\ \&\ y = x \\ &\quad \mid x < 0\ \&\ y = -x \end{aligned}$$

By writing $x{:}{<}L$ in the heading, we are stating that the value of x will be known when *Abs* is called. By writing $y{:}{>}L$, we are stating that, when *Abs* has finished, y will have a value. In other words, x is an *input parameter* and y is an *output parameter*. The Trilogy compiler will treat *Abs* as a function and will compile it as efficiently as if it were written in an imperative language such as Pascal.

Trilogy programs satisfy the laws of logic in the same sense that Miranda programs satisfy the laws of algebra. We can replace predicates and use the laws governing logical connectives. The manipulations may change the efficiency of a program, but they do not change its meaning or the results that it obtains. Consider, for example, the following Trilogy programs, in which P denotes a potentially long computation that is satisfied by either $x = 1$ or $x = 2$.

(8) $(x = 1 \mid x = 2)\ \&\ P\ \&\ x = 2$
(9) $x = 2\ \&\ P\ \&\ (x = 1 \mid x = 2)$

These programs are equivalent because "&" is commutative. The execution of (8) proceeds as follows: x receives the value 1, to satisfy the first term of the conjunction; P is evaluated but $x = 2$ fails. The processor tries again with $x = 2$, P succeeds again, and the program as a whole succeeds. Executing (9), on the other hand, evaluates P once only.

This program is trivial and improving it is straightforward. It is more difficult to improve complicated programs, but we quickly learn rules of thumb such as "put ORs as late as possible" The important point is that the program obeys simple and familiar mathematical laws. We can always distinguish between changing the meaning of a program and rearranging it for efficiency.

5. Pictures

Graphics provide a useful summary and illustration of the different paradigms of programming. Papert's most dramatic contribution was to teach programming concepts using graphics. Images on the screen provide an immediate indication that a program is correct or incorrect. The significance of LOGO, especially for beginners, is in large part due to turtle graphics. The turtle provides a direct, visual model for the execution of a LOGO program. A turtle program is a sequence of turtle instructions and, as the program runs, the turtle moves. Mistakes in the sequence are instantly visible. Adding graphic operations to techniques in these languages are quite different from those of LOGO. They consist of statements about pictures rather than recipes for drawing them.

For any value in a mathematical space, we require a representation. We represent numbers, for example, with strings of digits. A picture in a functional or logical program is a value whose representation is a screen image. We can define primitive pictures and functions or relations which are appropriate for pictures. The most primitive value is a point; a line or curve is a set of points; and a picture is a set of lines. Useful relations include *above, beside, on* (superimposition), and *inside* (superimposition with a scaling factor). The value of the expression *above (triangle, square),* for example, might be a drawing resembling a house.

The relationship between turtle graphics and functional graphics epitomizes the relationship between the programming paradigms. It is difficult to draw with a turtle for reasons that are irrelevant to the act of drawing. Hillel describes the difficulties that children encounter when drawing simple shapes [5, 6]. For example, when the turtle has drawn the base of the house, what angle must it turn through before starting the roof? Problems like this do not exist in functional graphics, because all we have to say is that the roof is *above* the base. Henderson illustrates the power of functional graphics by giving a short program that reproduces M C. Escher's drawing *Square Limit* [4].

By using turtle graphics, we are placing obstacles in the path of children who want to draw with the help of computers. A more primitive, and therefore more difficult, method does not necessarily provide a better learning experience.

6. Conclusion

How do functional and logical programming relate to the issues raised by *FLM* authors and others?

Samurçay notes that, after ten programming sessions, children have difficulty transforming the algebraic description of a program into a procedural description [9]. She attributes the difficulty to the differences between mathematical variables and equations on the one hand and program variables, assignments, and loops on the other. This is an example of what Papert sought to prevent: the computer is programming the children, making them express calculations as sequences of instructions.

Leron and Zazkis draw a parallel between recursion and induction [7]. They observe that children find it easier to write recursive programs than to understand inductive proofs. This suggests that functional programming might not be too abstract for children and that it might be fruitful to use recursive functions as a prelude to teaching inductive proof techniques.

Hancock emphasizes the need for a *mental model* of a program [3]. He explains that languages such as BASIC do not have an explicit model and beginners have difficulty constructing their own models. The difficulties arise partly because the programmer must imagine the state of the computation and the effect of each instruction on the state. Functional and logical languages have explicit models: equations and predicates, respectively. Programming in these languages has the potential for complementing and developing mathematical skills, rather than working against them, as BASIC seems to do.

Hillel discusses some of the difficulties that children have with turtle graphics [5] and illustrates them in an amusing and memorable way [6]. In particular, he identifies cognitive difficulties in moving from the intuitive way

of analysing and reconstructing drawings to the "procedural analysis" required by LOGO. He argues for LOGO and turtle geometry on the grounds that they contain enough content "to keep children busy throughout their elementary schooling." But should we keep children busy by teaching them unnecessarily primitive techniques?

The disadvantage of BASIC and LOGO is that they make programming more difficult than it needs to be. Children are restricted to solving trivial problems with the computer, not because they cannot understand the problems but because they cannot write programs of the required complexity. The most important reason for introducing high-level languages into the classroom is that they can be used to solve interesting problems. We can tell the computer *what* it has to do without explaining in laborious detail how to do it.

Although I have discussed both functional and logical languages in this article, my personal belief is that logical languages are easier to understand than functional languages. The principles of logic are more easily grasped than the principles of higher order functions. Prolog has already been used successfully in European schools. Just as the research language LISP inspired the educational language LOGO, advanced functional and logic languages will inspire new educational tools. The techniques are available to anyone who seeks to apply them.

Notes on the programming languages

Both Miranda and Trilogy are general purpose programming languages. A full description of either of them is beyond the scope of this paper. Both languages have a complete range of data structures. Miranda has a polymorphic type system which combines the freedom of LOGO with the security of Pascal. Trilogy has Pascal-like types, but the syntax is simpler than that of Pascal and local type declarations are not required because the compiler can infer the necessary information. The "Tri" in "Trilogy" indicates that Trilogy combines the features of third, fourth, and fifth generation programming languages. It is a logic language that also contains features for imperative, functional, and database programming.

Miranda is available for Unix machines but not, as far as I know, for personal computers. Trilogy runs on IBM PCs and compatibles and is available from Complete Logic Systems.

References

[1] Darlington, J. Program transformation. In *Functional Programming and its Applications: an Advanced Course*. Cambridge University Press, 1982. 193-215.

[2] Grogono, P. More versatility with Pascal-like Trilogy. *Computer Languages*, 5, 4 (1988). 83-89.

[3] Hancock, C. Context and creation in the learning of computer programming. *FLM*. 8, 1 (1988), 18-24.

[4] Henderson, P. Functional Geometry. *Symposium on LISP and Functional Programming*. ACM. 1982. 179-187.

[5] Hillel, J. On logo squares, triangles and houses. *FLM*. 5. 2 (1985). 38-45.

[6] Hillel, J. 'Un château en Espagne'. *FLM*. 6. 3 (1986). 24-26.

[7] Leron, U. and R. Zazkis. Computational recursion and mathematical induction. *FLM*. 6. 2 (1986), 25-28.

[8] Papert, S. *Mindstorms*. Basic Books, 1980.

[9] Samurçay, R. Learning programming: an analysis of looping strategies used by beginning students. *FLM*. 5, 1 (1985), 37-43.

[10] Senteni, A. Filling squares: variations on a theme. *FLM*. 6. 2 (1986), 13-17.

[11] *Trilogy User's Manual*. Complete Logic Systems, Inc., 741 Blueridge Avenue, North Vancouver, BC, Canada V7R 2J5.

[12] Turner, D. *SASL Language Manual*. St Andrews University Technical Report, 1976.

[13] Turner, D. Miranda: a non-strict functional language with polymorpyhic types. *Lecture Notes in Computer Science*. vol 201. Springer-Verlag, 1985.