The Fall and Rise of FP*

Rafael Dueire Lins
Departamento de Informática,
Universidade Federal de Pernambuco,
Recife, PE, Brazil
e-mail: rdl@di.ufpe.br

Abstract

Functional programming is a neat and very high-level programming paradigm, in which a program is a formed by a series of function definitions.

This paper analyses the evolution of functional programming within almost four decades of its existence.

1 Introduction

The first programming systems were systems of subroutines. The Mark I, a large electromechanical computer built in the early forties as a joint effort by Havard University and IBM, used subroutines already [1]. The Univac I appeared in 1951 as the first large scale electronic computer commercially available. It was designed so as to be relatively easy to program in its own code. The Univac I was a decimal, alphanumeric machine, with mnemonic instructions that were considered easy to remember and use. Programmers could not always see the advantage of assembly systems and compilers that were slow and clumsy on a machine with only 12,000 characters of high speed storage (200 microseconds average access time per 12 character word) [124].

Against the stablishment of her time, Dr. Grace Hopper firmly believed that programming ought and would be done in problem-oriented languages. Her group at Eckert-Mauchly Computer Corp., which later became the Univac division of Sperry Rand, was responsible for the development of a whole series of languages. The Univac programming group produced the first sort generators in 1951 and what was probably the first large scale symbol manipulation program, a program that performed symbolic differentiation [78].

Independent work also at Univac developed by Anatol Holt and William Turanski [66] gave birth to the concept and the compiler called GP (Generalised Programming), in which all programs would be written as if they were to be library programs. The program as written by the programmer would provide parameters and specifications according to which the very general library programs would be made specific to a particular problem. Subroutines were organised hierarchically in such a way to allow subroutines in one level to call on others at the next level. Specifications and

*Title inspired by the British Broadcast Corporation classical comic series "The Fall and Rise of Reginald Perrin", aired on the BBC from 1976 to 1979.

parameters could be passed from one level to the next. Emphasis was placed in the programming system that performed all kinds of services and library maintenance functions in addition to the translation and running of programs, rather than on the programming language itself.

The seed of functional programming was planted by merging the fundamental ideas of those two groups at Univac. Grace Hopper's group contributed with the genes of symbolic computations and making programming languages closer to applications. Holt's group contributed with the genes of having a higher-level of abstraction and machine independence. At this time, lists were pointed out as providing "a kind of modelling of objects inside the computer which expressly shows their relationships, in a way which makes finding their addresses easy" [49]. Those three elements together gave the fertile ground for the birth of LISP, which was designed to facilitate experiments with a system called the Advice Taker, "whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions [102].

In [56], David Gries reports that in the early 1960s very few people were involved deeply in understanding programming *per se* and that people seemed to be satisfied with programming as it was being performed. A few isolated papers and discussions did give some early indications that much remained to be done in the field of programming. One of the first references to the idea of proving programs correct was presented by John McCarthy in [103], where he stated that "instead of trying out computer programs on test cases until they are debugged, one should prove that they have the desired properties". As the 1960s progressed, it was slowly realised that there really were immense problems in the software field. The size and complexity of programs increased tremendously, without commensurate increases in the tools and abilities of the programmers; the result was missed deadlines, cost overruns and unreliable software. A consensus emerged that there really was a software crisis, that programming was not very well understood. The progress in hardware design over the last forty years was paramount. On the other hand, software design face an even more embarrassing situation than in the late 1960s, because larger software tools were built and very little advance was made in terms of reliability.

Imperative languages closely reflect the architecture and operations of von Neumann machines, requiring programmers to pay attention to machine-level details. This influence can be seen in the style of programming promoted by these languages, which is based on naming of elementary cells, assignments to these cells, and repetition of elementary actions. The state of computation is determined by the contents of each and every memory cell imposing a great difficulty in reasoning about the correctness of programs.

The functional programming style does not depend on these three actions. The simple and uniform data objects allow the design of data structures without concern for memory cells; values are produced by function application and passed on to other functions. Functional languages are, in essence, a nicer syntax to the λ-Calculus, a theory of functions [32, 12]. This mathematical background brings a promise of a much higher degree of abstraction into programming and hopes for better code provability and readability.

The history of the functional programming paradigm, now almost four decade old, comprises four phases. The first one spans from 1960 to 1978, starts with LISP, which can be pointed out as the first functional language. During that period was the most widely known and used functional language. The second period is inaugurated with John Backus' Turing Award Lecture [11], in which he drove the attention of the computer science community to functional programming as offering an alternative to the problems of programming known as "the software crisis" and as a way to break the von-Neumann bottleneck between processors and memory, which he perceived to be a limiting factor in the design of new computer architectures. This was the era of the boom of formal methods and functional programming. In the late 1980s, the end of the cold war, also meant a general loss of interest in formal methods – the Berlin wall and functional programming fell down together. The last phase of the FP history shows the rise in importance of the functional programming paradigm is happening slowly and steadily. Like a phenix, functional programming is ressurecting from its own ashes. This paper analyses the milestones of these four periods and casts some light on the rôle FP will have to play in the next millenium.

2 From 1960 to 1978: the LISP era

The concept of what is a functional language has evolved with time, but its main feature is that a program is a set of definitions of higher-order functions – functions are not only passed as parameters to other functions, but can also be the result of the evaluation of a given function. According to this definition, LISP (LISt Processor) [102] was the first functional programming language.

But LISP was not the only functional language in the early days of functional programming. APL (Applicative Programming Language), designed by Kenneth Iverson at Havard University during the late 1950s and early 1960s [74] can be viewed as an applicative language because of its heavy realiance on expressions. According to Ghezzi and Jazayeri [51] both APL and LISP were widely used and popular with their users. Landin's ISWIM (**I**f you **S**ee **W**hat **I** **M**ean), [88], Evans PAL (Programming Applicative Language)[47], Burge's McG [22] and Reynold's Gedanken [120] are examples of other functional languages of this period.

In this section, we analyse the important contributions of LISP to the functional programming paradigm and to the implementation of functional languages.

2.1 The LISP programming System

The LISP programming system was developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T., led by John McCarthy. The original aims of the system were [102]:

- Writing a compiler to compile LISP programs into machine language.
- Writing a program to check proofs in a class of formal logical system.
- Writing programs for formal differentiation and integration.
- Writing programs to realize various algorithms for generating proofs in predicate calculus.

- Making certain engineering calculations whose results are formulas rather than numbers.
- Programming the Advice Taker system.

The first version of LISP in 1960, also known as pure LISP, was completely functional. Nonapplicative features were introduced into the language later, in order to improve execution efficiency.

2.1.1 Garbage Collection

One of the most important features of a list processing system is the method by which it allocates the computer storage to the list structures. The stack discipline of Algol, in which the space allocated by a procedure call is needed while the procedure is "alive" is no longer valid. In list processing systems, a list generated by a procedure may outlive its generator. At the same time, list processing systems have very high demands for space. Information (cells) may be shared and tend to be short-lived. Those cells that store information no longer needed for computation are called *garbage*.

Reclaiming garbage cells non-automatically tend to be an error-prone task, due to the complex generation and modification pattern followed by the graph during computation. To free users of this burden, make programming higher-level and more space efficient, McCarthy devised an automatic dynamic memory management strategy for LISP, which he called *a garbage collector*. His algorithm, called mark-scan, works in two phases. If a machine runs out of space, the computation process is suspended and the garbage collection algorithm is called. First, the algorithm traverses all the data structures (or cells) in use, putting a mark in each cell visited. Then, the scan process takes place collecting all unmarked cells in a *free-list*. When the mark-scan process has finished, computation is resumed. The amount of time taken for garbage collection by the mark-scan algorithm is proportional to the size of the heap (the work space where cells are allocated).

Also in 1960, G.E.Collins [34] devised a new garbage collection strategy to avoid the pauses to LISP programs imposed by the mark-scan algorithm called *reference counting*. In reference counting, each data structure or cell has an additional field which counts the number of references to it, i.e. the number of pointers to it. During computation, alterations to the data structure imply changes to the connectivity of the graph and, consequently, re-adjustment of the value of the count of the cells involved. The disadvantage of the trivial algorithm for reference counting is the inability to reclaim cyclic structures. The first general solution to this problem was given by the author of this paper in [90].

In 1969, Fenichel and Yolchelson presented the *copying* algorithm. It is a modified version of the mark-scan algorithm suitable for machines with virtual memory operating systems. In this algorithm the heap is divided in two halves. The algorithm copies cells from one half to the other during collection. Its time complexity is proportional to the size of the graph in use. Mark-scan and copying algorithms generally traverse all the reachable data structures during garbage collection, which makes them unsuitable for real-time or large-virtual-memory applications. We refer to [77] for a detailed analysis of these algorithms.

2.2 Implementation milestone: The SECD Machine

The very first abstract machine to evaluate λ-expressions was the SECD machine, developed in 1964 by Peter Landin [87].

Landin's SECD machine is a generalisation of the abstract machine underlying implementations of imperative programming languages like Pascal. It was used to give the operational semantics of Algol. It is named after its four components: **S**tack, **E**nvironment, **C**ode, and **D**ump.

As all functions were curryied the granularity of computation was very fine. Variable capture was a problem and its avoidance implied in scanning the body of every λ-expression before replacing each formal parameter by the corresponding real parameter.

2.3 Strachey's Seven Questions

Following the presentation of Peter Landin's paper entitled "The next 700 Programming Languages" at the ACM Programming Languages and Pragmatics Conference in August 1965 there was a discussion printed together with the paper [88], in which Christopher Strachey raised seven questions. His views summarised on those points were most influential to the functional programming paradigm. Thus we quote them here:

> "I should like to intervene now and try to initiate a slightly more general discussion on declarative or descriptive languages and to try to clear up some points about which there is considerable confusion. I have called the objects I am trying to discuss DLs because I don't quite know what they are. Here are some questions concerning DLs: (1) What are DLs? (2) What is their relationship to imperative languages? (3) Why do we need DLs? (4) How can we use them to program? (5) How can we implement them? (6) How can we do this efficiently? (7) Should we mix DLs with imperative languages?
>
> It seems to me that what mean by DLs is not exactly what other people mean. I mean, roughly, languages which do not contain assignment statements or jumps. This is, as a matter of fact, not a very clear distinction because you can always disguise the assignments and the jumps, for that matter, inside other statement forms which make them look different.
>
> The important characteristic of DLs is that it is possible to produce equivalence relations, particularly the rule for substitution which Peter Landin describes as (β) in his paper. That equivalence relation, which appears to be essential in almost every proof, does not hold if you allow assignment statements. The great advantage then of DLs is that they give you some hope of proving the equivalence of program transformations and to begin to have a calculus for combining and manipulating them, which at the moment we haven't got.
>
> I suggest that an answer to the second question is that DLs form a subset of all languages. They are an interesting subset, but one which is inconvenient to use unless you are used to it. We need them because at

the moment we don't know how to construct proofs with languages which include imperatives and jumps.

How should we use them to program? I think this is a matter of learning a new programming technique. I am not convinced that all problems are amenable to programming in DLs but I am not convinced that there are any which are not either; I preserve an open mind on this point. It is perfectly true that in the process of rewriting programs to avoid labels and jumps, you've gone half the way towards going into DLs. When you have also avoided assignment statements, you've gone the rest of the way. With many problems you can, in fact, go the whole way. LISP has no assignment statements and it is remarkable what you can do with pure LISP if you try. If you think of it in terms of the implementations that we know about, the result is generally intolerably inefficient – but then that is where we come to the later questions.

How do we implement them? There have not been many attempts to implement DLs efficiently, I think. Obviously, it can be done fairly straightforwardly by an interpretive method, but this is very slow. Methods which compile a runable program run into a lot of very interesting problems. It can be done, because DLs are a subset of ordinary programming languages; any programming language which has sufficient capabilities can cope with them. There are problems, however: we need entities whose value is a function – not the application of a function but a function – and these involve some problems.

How to implement efficiently is another very interesting and difficult problem. It means, I think, recognising certain subsets and transforming them from, say recursion into loops. This can certainly be done even if they have been written in terms of recursions and not, as Peter Landin suggested, in terms of already transformed functions like **iterate** or **while**.

I think the last question, "Should DLs be mixed with imperative languages?", clearly has the answer that they should, because at the moment we don't know how to do everything in pure DLs. If you mix declarative and imperative features like this, you may get an apparently large programming language, but the important thing is that it should be simple and easy to define a function. Any language which by mere chance of the way it is written makes it extremely difficult to write compositions of functions and very easy to write sequences of commands will, of course, in an obvious psycological way, hinder people from using descriptive rather than imperative features. In the long run, I think the effect will delay our understanding of basic similarities, which underlie different sorts of programs and different ways of solving problems."

The seven questions formulated by Strachey more than thirty years ago are still valid. His answers or feelings guided a whole legion of researchers ever since.

2.4 SASL: the sign of new times to come

In order to make LISP more acceptable to users in general, a number of imperative features were introduced to pure LISP. This trend was followed by languages such as ISWIM [88], PAL [47], McG [22] and Gedanken [120]. In his Ph.D. thesis David Turner[1] reports:

> "after teaching LISP for one term (at St.Andrews) the author resolved to replace it by a more straightforward syntactic sugaring of the lambda-calculus (or equivalently of higher order equations). Since the language was intended purely for teaching purposes, the author decided to break with the tradition of embedding an applicative language in a larger language containing imperatives. Instead, SASL (St. Andrews Static Language) was designed as a purely applicative language and has remained so ever since."

SASL was in its origins simply the applicative subset of PAL (Programming Applicative Language) [47]. It throw out all imperatives and kept the style of defining higher-order functions without explicit λ's[2], the pattern matching and so on. If a language has no destructive assignment, the value of any sub-expression is static, it is said to be "pure" or to enjoy the property of *referential transparency*[3]. SASL was the first pure functional language.

A SASL program is an expression - the result of running the program is to print the value of the expression. To give a taste of the evolution in syntax SASL brought into functional programming one can compare the factorial function in LISP

```
DEFINE((( FAC (LAMBDA(N)
         (COND ((ZEROP N) 1)
          (T (TIMES (FAC(SUB1 N)) N)
       ))))))
```

and in SASL:

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

SASL had numbers, truthvalues, characters, lists and functions as primitive types. Strings were just lists of characters. All data objects have the same "civil rights"[4], allowing a function to return another function as its result.

If the arguments to a function are passed by value, i.e. all the arguments are evaluated before the function itself, it is called "strict". Strict evaluation correspond to applicative-order reduction in the λ-calculus. Non-strict languages may be implemented using either a *data-driven* (also known as dataflow) or a *demand-driven*

[1] Turner worked for his Ph.D. under the supervision of Strachey from 1969 to 1972. His work was interrupted by Strachey's death.

[2] ISWIM was possibly the first language to abolish λ's. In footnote, Landin says: "a not inappropiate title (to his paper [88]) would have been "Church without λ""

[3] This term was first used by Whitehead and Russell [152]. They said that a statement is not referentially transparent if questions arise not merely about its content but also about the circumstances of its assertion, i.e. it is context dependent.

[4] Strachey advocated this idea and coined the expression supporting the "equal civil rights campaign" in the early 1960's in the USA.

approach. Demand driven languages evaluate arguments as required by the function. If the result of the argument is recomputed each time it is needed, this evaluation mechanism is called *call-by-name*, which corresponds to *normal-order* reduction in the λ-calculus. Conversely, if the result is shared the evaluation mechanism is called *call-by-need*, and the language is said to be *lazy* or *procrastinating*. This corresponds to normal-order reduction in Wadsworth's shared λ-calculus [144]. Non-strict languages have the advantage that only expressions which must be evaluated to give the program result actually are evaluated. They allow the use of infinite data structures, such as infinite lists, which are described by a formation law. These structures can be seen as intensionally defined and have been used to permit I/O to be handled in a more natural manner by allowing input and output to be treated as potentially infinite lists of characters and infinite precision arithmetics, for instance. The first versions of SASL (1972/1976) had a strict semantics and were implemented using the SECD machine. In 1976, Turner decided to have the semantics of SASL closer to the λ-calculus and adopted a lazy semantics, giving signs of a new era to be inaugurated in functional programming.

3 From 1978 to 1987: The FP Boom

In the early seventies, as David Turner reports in his thesis [139], functional languages had the status of academic curiosities used, if at all, only for teaching purposes. There was an growing feeling that the collection of ideas that came to maturity in the late sixties and which were loosely called "structured programming" were failing to deliver the reduction in software production costs which was hoped from them. During the late 1970s, the cold war made governments of countries such as the U.S.A., U.K., France, Germany, and Italy make massive investments in formal methods. Software reliability was fundamental to enterprises such as the Star War programme.

In this scenario, John Backus in his Turing Award lecture [11], drew the attention of the computer science community to functional programming as offering a solution to the software crisis[5].

The simplicity and beauty of SASL attracted many people and the failure of languages such as LISP was attributed to the imperative features added to it in order to become more efficient. Purity of programming style became the name of the game. No one was prepared to compromise in any point neither to make programs more efficient nor to get closer to the needs of applications. The functional programming community used to claim that functional programs were:

- of a higher level of abstraction and semantic elegance due to the use of higher-order functions and lazy evaluation.
- easier to write than their imperative counterparts, because programs were the algorithms themselves.
- easier to read due to its very compact notation, reaching in some cases less than 10% of the size of their imperative equivalent.

[5]Turner remarks [139] that "Backus' detailed proposals for what he calls functional programming are rather outside the main strean of applicative languages, but to discuss that here would take us too far afield and is in any case a separate question from the relevance of his criticisms of existing programming culture and the architecture on which it is based".

- easier to prove correct as functional languages were based on the λ-Calculus.
- easier to go parallel due to referential transparency.
- executable specifications, providing simulation of programs for free.

Several other purely applicative languages appeared at this time, including HOPE [27], ID [7], LISPKIT [63] (the applicative set of LISP with the scope rules cleaned up), and LUCID [10].

3.1 Turner's Graph Reducer

Around 1920, Moses Schönfinkel draw the attention of the logic community to the convenience of eliminating variables from first-order logic, by transforming it into an applicative combination of constant functions or **combinators**. A few years later, Haskell B. Curry rediscovered Combinatory Logic and built a deductive theory. Very early the similarities between the λ-Calculus and Combinatory Logic were spotted.

In his seminal paper describing LISP [102] J. McCarthy addresses the problem of variable capture and using combinators and says:

> Difficulties arise in combining functions described by λ-expressions, or by any other notation involving variables, because different bound variables may be represented by the same symbol. This is called collision of bound variables. There is a notation involving operators that are called combinators for combining functions without the use of variables. Unfortunately, the combinatory expressions for interesting combinations tend to be lengthy and unreadable.

Turner [138] used a larger set of combinators than the one in Curry's Combinatory Logic, which avoided the generation of the K combinator. This new bracket abstraction algorithm, yielded quadratic code expansion, while Curry's generated an exponential size growth in relation to the corresponding λ-expression. Turner rewrites the leftmost-outermost combinator at each point in the code. When no further rewriting can take place the expression is said to be in *normal form*.

Turner was also the first to propose a graph reduction machine to interpret combinatory expressions. A stack was used to find the leftmost-outermost combinator, the next to be reduced. A local *unwind* was performed after each combinator reduction, avoiding pointer following from the root of the graph. An explicit fixpoint combinator was introduced to implement recursion. Instead of following its rewriting law:

$$Y\ x \Rightarrow x\ (Y\ x)$$

Turner used a strategy of performing the graph rewriting to an application of x to itself.

The simplicity of Turner's abstract machine made it a striking success. It inspired a legion of researchers all over the world who tried to get better sets of combinators. In [38], Curien presented a set of combinators based on the equivalence between the theories of the typed λ-calculus and cartesian closed categories as shown by Lambek [85] and Scott [127]. This system, called Categorical Combinators, served as the basis for the Categorical Abstract Machine [35], an abstract machine for the implementation of strict functional languages. Categorical Combinators were largely optimised in the rewriting system of Categorical Multi-Combinators [89], which are used to implement lazy functional languages [98, 93, 135, 96, 95].

3.2 ML

Since the very beginning, functional languages were used for theorem proving. The functional language "ML", which stands for Meta Language, was developed as a command language for the LCF (Logic for Computer Foundation) system, a system for proving the correctness of programs. ML is a strict functional language which contains assignments and references, it is thus impure.

The most important contribution of ML to programming is its type discipline. Based on Milner's polymorphically typed λ-calculus, ML was the first strongly typed functional language. The use of Hindley-Milner's algorithm allows a mechanism for type inference, which detects a large number of programming errors.

The language SML [105], Standard ML, is the result of a standardization effort and a marriage of the languages ML and HOPE [27].

3.3 Miranda

Miranda[6] [141] is a polymorphically typed pure lazy functional language developed by David Turner. Its syntactic elegance made it a standard in functional programming. Although, a detailed introduction to Miranda is out of the scope of this paper, we will emphasise some of its important contributions to the functional programming style.

The primitive or basic types in Miranda are: *num, char*, and *bool*. The type *num* comprises both integers and floating point numbers. As elements of the type *bool* we have the truth values *True* and *False*. The type *char* comprises the characters of the ASCII character set. In-built in Miranda there are also more complex data type constructors such as 'list' and 'tuple'.

A new algebraic type in Miranda is introduced by an equation using the symbol '::='. For instance, the type 'tree' can be introduced by the declaration of the type constructors as

```
tree ::=  Leaf num | Node num tree tree
```

Leaf and *Node* have type $num \rightarrow tree$ and $((num \rightarrow tree) \rightarrow tree) \rightarrow tree$, respectively.
(The $\rightarrow$ operator denotes curryfication[7] and is left-associative). In Miranda the definition mechanism uses pattern matching on constructors. For instance, we can define a function which gives the 'weight' of a tree as

```
wt (Leaf n) = n
wt (Node n t1 t2) = (wt t1)+(wt t2)+n
```

The type system of Miranda subsumes Pascal enumerated types, records, variant records, and some uses of pointers which are involved in the construction of recursive dynamic data structures.

[6] Miranda is a trademark of Research Software Ltd.

[7] After the logician H.B.Curry. Curryfication means the transformation in which a n-ary function becomes n functions to one argument.

ZF-expressions are a convenient and elegant syntactic 'sugaring' of a class of expressions which was introduced by Turner in KRC[140] and kept in Miranda with the same syntax:

$$\{exp\ ;\ qualifiers\}$$

where there can be any positive number of qualifiers. Each qualifier is either a generator or a filter. A generator is a list of values which will be attributed to a specific variable in the expression *exp*. A filter is an arbitrary boolean expression used to restrict further the range of a generator.

For example, the list of all the squares of the integer numbers between 1 and 10 can be expressed in Miranda by the following ZF-expression:

```
{ x*x; x <- (1..10)}
```

3.4 FP and Parallelism

The idea of parallel functional programming dates back to 1975 when Burge [23] suggested the technique of evaluating function arguments in parallel, with the possibility of functions absorbing unevaluated arguments and perhaps also exploiting speculative evaluation. In that same year, Berkling[17] suggested the application of functional languages to parallel processing.

The first attempt to exploit the parallelism of functional programs targeted either the evaluation of actual parameters before replacing them by the formal parameters or was done at combinator argument level. These strategies aimed at very fine grained parallelism and worked at a very high level abstract machines, such as graph reducers.

During this period novel architectures were seen as the way out of the inefficiency of functional languages, if compared to imperative ones such as C or FORTRAN. This led to a spate of designs for special-purpose machines, many of which were parallel. ALICE (Applicative Language Idealised Computing Engine) [40], designed at Imperial College in 1981, by Darlington and Reeve was the first and most famous reduction machine. The prototype ALICE comprised 40 Transputer-based processing agents and packet pools, connected by a multi-stage switching network. The eventual aim was to build ALICE in VLSI, but the only ALICEs built used stock hardware. The performance of this machine, as all the others from this period, was ultimately disappointing [62]. The granularity of computations was too fine and the use of many small packets also degraded performance to some extent. ALICE served as the basis for other more recent architectures such as the ICL Flagship [149, 147, 46] and EDS/Goldrush designs [146].

Parallel combinator reducers such as COBWEB [61], Burroughs' NORMA [122], and SKIM [133] are examples of other early machines which were built, but were overtaken by the widespread adoption of compilation techniques inaugurated by the G-machine. Machines such as Magó's FFP [100], Rediflow [80] and COBWEB [4] have failed to come to fruition.

One interesting proposal of parallel architecture was the graph reducer GRIP (Graph Reduction in Parallel) [115, 114], which is built from a network of distributed conventional processors. It has a two-level bus structure and incorporates fast packet-switching hardware for message routing. Intelligent memory units would operate on globally shared graph and spark task pools. It uses microcoded CPUs for the intelligent memory units and PALs for the packet-switching hardware. This gave a degree of design flexibility which allowed to undergo a transformation from a parallel abstract machine interpreter to a machine running compiled Haskell directly. Even with this new configuration the performance of the parallel GRIP machine is slower than the single node (sequential) version of the Haskell compiler.

On the language side a number of ideas were tried. Serial Combinators [68] were intended to be inserted by an automatically partitioning compiler. The Alfafa project implemented serial combinators for the Intel iPSC [52], but the performance figures were disappointing. Buckwheat re-implemented this model for the Encore Multimax, as shared-memory multiprocessor but got only a relative speedup [53].

Annotations for parallelism have been proposed by many authors. Burton's $@p$ to indicate parallel function application [29] was one of the first and simplest ideas. Burn's work [25, 24] on evaluation transformers and Hope$^+$ [82] on Flagship used complex annotations to control the precise degree of evaluation through strictness. Hudak's work [67, 71] on *para-functional programming* for functional languages with annotations that preserve the functional semantics, uses the idea of *schedules* of events, which includes explicit demands and process creation/termination, plus sequential and parallel compositions as well as task mappings to particular processors. Caliban [81] provided a separate functional language for task placement, in which a program comprises a process part and a wiring part. An implementation of Caliban for the Meiko transputer surface [36] was made but no performance figures were published [59].

The idea of trying to capture patterns of parallel computation, such as divide-and-conquer or pipelining, was named as *algorithmic skeletons* by Cole [33]. Ideally, the same skeleton can be used for different architectures: it is necessary only to change the implementation of the skeleton in order to get good performance portability. The user's program itself is unchanged, both textually and semantically. ZAPP [28] can be seen as an early example of skeletons. The use of skeletons seems to be promising and is being continued at several sites.

3.4.1 Dataflow

Single-assignment languages or *dataflow* may have either strict or non-strict semantics. The evaluation of all the arguments to functions is performed before evaluating the application.

The TTDA (Tagged Token Dataflow Architecture) designed at MIT [8] was never realised in hardware. The dataflow machine built at Manchester [57, 58] demonstrated good relative speedup for several small benchmark programs, and provided much basic data on task scheduling. The prototype machine with 12 function units ran dataflow programs at about 25% of the speed of a VAX 11/780 running C [57]. The Monsoon [113], the SIGMA-1 [153], the P-RISC [9] and *T [110] are examples of dataflow architectures. The last two of them are derived from conventional RISC machines.

Dataflow programs typically produce much fine-grained parallelism, which must be managed carefully to avoid memory exhaustion [37]. Despite the attempts to increase granularity by combining instructions into larger blocks, the results are still relatively fine-grained [125].

3.4.2 Parallel Garbage Collection

Donald Knuth credits Marvin Minsky for first suggesting parallelism as a way to avoid suspension of operations (Exercise 2.3.5–12, pp.422 in [83]). Parallelism need not imply concurrency. Garbage collection could occur, for example, during keyboard input, as long as it could be suspended on short notice to continue processing on the input and later be resumed without losing all the previously expended effort.

The model followed by architectures that tried to exploit parallelism of garbage collection consists of two processors, called the *mutator* and the *collector*, working in parallel and sharing the same the workspace (which is organised as a heap of cells). The mutator does all the 'useful' work, modifying the connectivity of the data structure and the values of data fields within that structure. The collector is solely responsible for identifying and re-cycling garbage cells. Its algorithm is based on Mark-Scan but runs continuously in two phases. In the first phase active cells are identified and marked; in the second phase the heap is scanned and cells that are known to be certain garbage are returned to the free-list. The collector makes no change to shape of the heap other than to link garbage cells into the free-list. It does, however, use and make changes to a mark field within each cell. These mark fields are also used by the mutator; indeed, they provide all the necessary communication between the two processes. Note that this architectural model is can easily adapt to work with Turner's graph reducer.

Guy Steele's Multiprocessing Compactifying Garbage Collection algorithm was the first published parallel and concurrent architecture for garbage collection [130, 131]. In addition to freeing unused storage, Steele's algorithm compacted remaining list structures to give better performance in a virtual memory environment. The mass of detail presented by Steele contributed to make understanding his ideas difficult.

Independently, Dijkstra proposed a similar scheme in some unpublished notes [43], later published in [44]. Dijkstra and his colleagues tackled this problem 'as one of the more challenging — and hopefully instructive — problems' in parallel programming. Their architecture attracted considerable interest in the computer science community. Woodger's scenario showed that if the granularity of the coarse grain algorithm was made finer, a bug would appear; in describing his proof of the algorithm Gries reported that he had 'seen five purported solutions to this problem, either in print or ready to be submitted for publication' each of which contained errors [55]. A correct version of the algorithm appeared in [45]. Wadler showed that, for time-sharing rather than multiprocessor systems, such algorithms require a greater percentage of processor time than classical sequential collection does [142].

Kung and Song developed a version of Dijkstra's algorithm which used four colours but did not need to trace the free-list [84], and Ben-Ari, who considered it 'one of the most difficult concurrent programs ever studied', presented several parallel mark-scan algorithms based on it but with much simpler proofs of correctness than those presented by Kung and Song, Gries and Dijkstra *et al.* [14, 15]. Ben-Ari's algorithms used only two colours. Gries accredits Stenning for an unpublished version of the on-the-fly algorithm which also used only two colours. Lamport generalised the architecture for using multiple processes [86]. The on-the-fly algorithm was also implemented in hardware/software in the Intel iAPX-432 microprocessor and iMAX operating system [119].

4 1987 to 1989: The Fall of FP

The end of the cold war meant that reliability was not the most important feature one could expect from software anymore. Research teams had to be redirected towards products which ought to be ready for sale within a 6 to 12 month horizon. The promises of the past were not delivered on time and general pessimism was cast on previous hopes:

- higher-order functions and lazy evaluation perform symbolic computation imposing that large data-structures are kept unevaluated, resulting in inefficient sequential code.
- functional programs were unable to cope with Input/Output easily.
- function composition can make programs extremely intricate to read in true spaghetti style.
- the size of large functional programs was comparable to the imperative equivalent.
- proving the correctness of functional programs, although much easier than imperative ones was not easy at all and would only be possible with proof assistant tools.
- to exploit referential transparency as a way to obtain parallelism one gets too fine granularity of code yielding parallel inefficiency.

5 From 1989 onwards: The Rise of FP

In February 1993, the Department of Trade and Industry (DTI) of the British Government organised a workshop in London to know if functional programming was dead and buried or if there were ways of resurrecting it. The author of this paper was amongst the invitees to this workshop. The FP community arrived to the conclusion that a number of important steps had been made:

- we had learned *how to compile* functional languages in sequential machines.
- the need for large benchmarks was clear. Benchmark suites such as *NoFib* and *Pseudoknot* [50] were developed and allow comparing different implementations.
- the concept of *monads* [106] made possible to express I/O and the concept of state in true functional style.
- a number of "real-world" applications gave very good results:
  - The experiment of David Turner at AMACO (U.S.A.) in translating old oil reservoir programs written in FORTRAN into Miranda, an interpreted functional language, yielded a performance gain of around 1.000%.
  - At ECRC (München) the team led by Mike Reeve obtained a performance gain of about 300% by translating FORTRAN code into ML of the program for simulating chemical pollution in Venice bay.

– Programs for numeric computation written in the single-assignment functional language Sisal [104, 30], developed at Lawrence Livermore National Labs., rival those for Parallel FORTRAN.

The general pessimism at the beginning of the Workshop gave room to a renewed feeling of importance and viability of the functional programming paradigm. Below, we analyse some of the cornerstones on top of which the FP paradigm is being rebuilt.

5.1 Compiling Functional Languages

Language compilation is a much faster implementation technique than interpretation. An important step towards the the efficient implementation of strict functional languages was Cardelli's FAM [31].

The G-Machine [75, 97] was developed by Johnsson and Augustsson, in the *Chalmers Institute of Technology*, Götemburg, Sweden, with the aim of providing efficient implementation of lazy functional languages in von-Neumann machines. The G-machine merged the best features of the SECD and Turner machines.

The original G-Machine is very simple. We can say that this machine works as an interpreter with *lazy graph generation*, i.e. the original graph is replaced by code which when executed generates a graph to be interpreted.

Suppose, for example, that we want to evaluate the following expression, which returns the list of the squares of each Natural number.

$$\begin{array}{l} \mathit{list}\ 0,\ \text{where} \\ \mathit{list}\ n = \mathit{square}\ n : \mathit{list}\ (\mathit{suc}\ n) \end{array}$$

In the definition above, the symbol : denotes the infix list construction operator and *square* and *suc* are pre-defined functions and are defined as follows,

$$\begin{array}{rcl} \mathit{square}\,x & = & x \times x \\ \mathit{suc}\,x & = & x + 1 \end{array}$$

The expression, *list* 0, will be represented in the G-Machine as in Figure 1(a).

Using the definition of the function *list* as a rewriting law, this expression is reduced to Figure 1(b). The integer 0 is the shared argument to *square* and *suc*. The resulting list expression is in canonical form, but neither its head nor its tail are. After reducing a graph to its canonical form, the next step is printing the result. Printing a list means to print its head and then to print his tail. As only ground type expressions can be printed, the machine will, first, reduce the head of the list, print it and then reduce its tail and print it. Using the definition of *square* as a rewriting law, the head of the list is then rewritten (for shorter) to the square of the integer 0, that is also 0. This rewriting step can be seen in Figure 1(c).

The sequence of reductions presented schematically above present "what" the G-Machine does. Now, as the head of list is on canonical form, it can be printed and removed from the graph (Figure 1(d)).

The evaluation of the tail continues in a similar fashion(Figure 1(e)). Please observe that computation does not terminate. Now the argument to *square* and *suc* is the expression *suc* 0. Again, the machine will try to print it. The definition of *square* reduces the expression to the square of the integer denoted by the expression

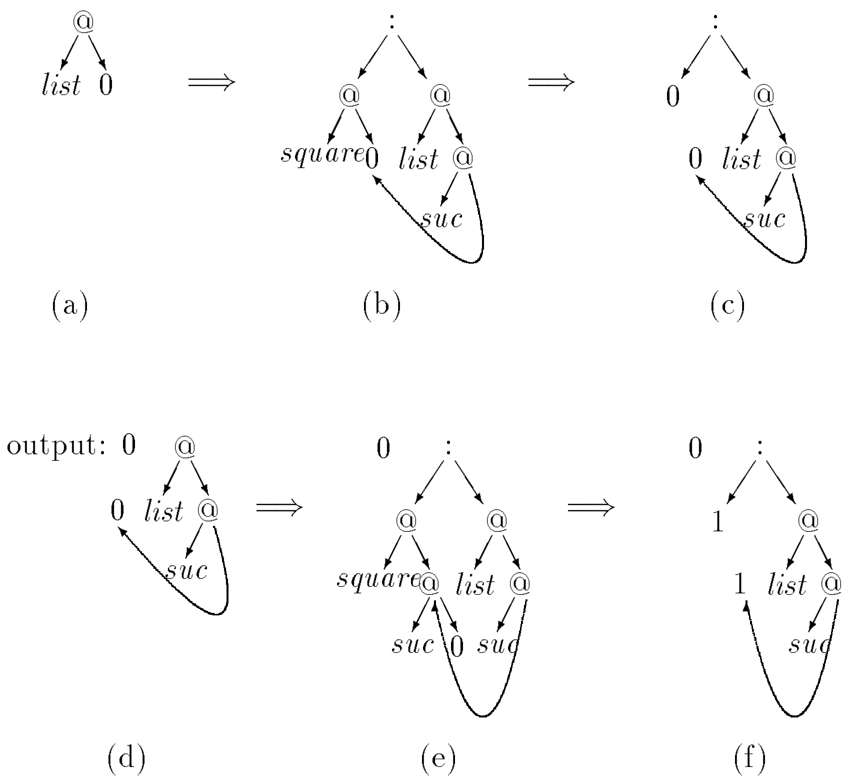

Figure 1: Example of the Evaluation Mechanism

suc 0. First, the graph representing *suc* 0 is reduced to its canonical form, the integer 1. As *suc* 0 is shared, all expressions that reference it will benefit from the reduction accomplished. Then, *square* is applied on the resulting expression, yielding the integer 1. The graph after these rewritings is as in Figure 1(f).

Again, as the head is now in canonical form, it can be printed and dropped from the graph. Reduction continues in the same manner.

The G-machine [75] provided the first fast implementation for lazy functional languages. The code generated by the G-Machine when executed produces time and space performance at least an order of magnitude faster than interpreted functional languages. Lazy graph generation makes the graph to be generated only if needed (in the case of higher-order functions and lazy evaluation). It also opens the possibility of a myriad of code optimisation strategies, avoiding graph generation at all. The original G-Machine implemented at Chalmers, by Johnsson and his colleagues [75], generated code in VAX-780 Assembly language, which made implementation extremely hard and machine dependent. It was common sense in the community of implementation of functional languages that assembly language implementation was the price to pay if one wanted efficiency. Chalmers LML compiler is still a reference in terms of performance of lazy functional languages. The G-machine way of controlling the execution flow and evaluation was followed by most of other implementations, even the ones based on different abstract machines as the Spineless G-Machine [26], the Spineless Tagless G-machine [116], TIM [48], and GM-C [107].

5.2 Haskell

In the late 1980s there was more than a dozen non-strict, purely functional programming languages, all similar in expressive power and semantic underpinnings. There was a strong feeling amongst the community that more widespread use of this class of functional languages was being hampered by the lack of a common language.

In September of 1987, a meeting was held at the conference on Functional Programming Languages and Computer Architecture in Portland, Oregon, to discuss this unfortunate situation. It was decided that a committee should be formed to design such a common language, providing faster communication of new ideas, a stable foundation for real applications development, and a vehicle through which others would be encouraged to use functional languages.

This language was called Haskell [70] and had as goals:

- to be suitable for teaching, research, and applications, including building large systems.
- to be completely described via the publication of a formal syntax and semantics.
- to be freely available, implemented and distributed.
- to be based on ideas that enjoy a wide consensus.
- to reduce unnecessary diversity in functional programming languages.
- to serve as a basis for future research in language design.

A detailed analysis of Haskell is out of the scope of this paper. We recommend [136] as an introduction to Haskell and functional programming.

5.3 Benchmarking

Due to the inefficiency of the implementations of functional languages and also the limitations of architectures available, the benchmarks used either in sequential or parallel machines were too small. Implementation performance were measured in *fibs per second*, the number of Fibonacci calculations performed in one second. Most implementations were able to calculate **fib 20** as their largest Fibonacci number.

A classic divide-and-conquer program, a variant on the naive Fibonacci program, called `nfib` was used to benchmark parallel implementations:

```
nfib n = if n <= 1 then 1
         else 1 + nfib(n-1) + nfib(n-2)
```

Since the two recursive calls to `nfib` are independent, they can each be executed in parallel. If it is done naively an exponentially large number ($O(2^n)$) of tasks is created, in equal number to the result of the program.

The implementation maturity reached by many functional compilers and interpreters allowed larger programs to be written. Serious benchmarks which could really acess the compilation and execution times of functional languages in "real-world" applications were needed. It would also be relevant to compare functional and imperative languages. We quote Will Partain on benchmarking functional languages [112]:

"Inattention to benchmarking is not just sloppy, it end up as self-delusion. Assertions that various functional-languages compilers "...generate code that is competitive with that generated by conventional language compilers... [8] are simple false by any common-sense measure; what's more, when they are repeated by Respected People, they are downright harmful: they detract from the urgency of building better implementations."

We describe briefly the experience with *Pseudoknot* and *NoFib*, the two most important serious benchmarking efforts promoted by the FP community.

5.3.1 Pseudoknot

At the Dagstuhl Workshop on Applications of Functional Programming in the Real World in May 1994, several interesting applications of functional languages were presented. One of these applications, the pseudoknot problem [50] is derived from a "real-world" molecular biology application that computes the three-dimensional structure of part of a nucleic acid molecule. The program exhaustively searches a discrete space of shapes and returns the set of shapes that respect some structural constraint. After the workshop, Pieter Hartel and Martin Alt decided to coordinate what became the largest collaboration action in the FP community. Details of the languages which were benchmarked may be found in Table 1. The first column of the table gives the name of the language. The second column gives the source (i.e. a University or a Company) if a language has been developed in one particular place. Some languages were designed by a committee, which is also shown. The third column of the table gives a key reference to the language definition.

The last four columns describe some important properties of the languages. The typing discipline may be *strong* (and static), *dynamic*, or *weak*; a strong typing discipline may be monomorphic (*mono*), polymorphic (*poly*), or parametrically polymorphic (*param*). The evaluation strategy may be *eager*, *lazy* or *eager* with *non-strict* evaluation [108]. The language may be *first order* or *higher order*. Accessing components of data structures may be supported by either pattern-matching on function arguments, local definitions and/or as part of case expressions (*pattern*, *case*), by compiler generated access functions to destructure data (*access*), or not at all (*none*). The pseudoknot program uses mostly algebraic data types as data structures and does not explicitly depend on laziness for its correct behaviour. It is thus ideal for comparing lazy and strict implementations of functional languages. C was used as a reference language in order to allow comparisons with imperative languages.

The Pseudoknot benchmark was tested using 28 different implementations of 18 functional languages and variants. Several aspects were analysed, including compile and execution time ellapsed. The results showed that more effort had to be put in order to make functional languages efficient. When appropriate strictness annotations were used the lazy languages and strict ones presented the same run-time behaviour, and in some cases the performance approached that of C. Unprecedented collaboration took place within the FP community and people learned about each other work at a higher rate than ever.

[8] Citation withheld to protect the guilty!

language	source	ref.	typing	evaluation	order	match
SML family						
CAML	INRIA	[150]	strong, poly	eager	higher	pattern
SML	Committee	[105]	strong, poly	eager	higher	pattern
Haskell family						
Clean	Nijmegen	[118]	strong, poly	lazy	higher	pattern
Gofer	Yale	[76]	strong, poly	lazy	higher	pattern
LML	Chalmers	[75]	strong, poly	lazy	higher	pattern
Haskell	Committee	[70]	strong, poly	lazy	higher	pattern
Miranda	Kent	[141]	strong, poly	lazy	higher	pattern
RUFL	Rhodes	[151]	strong, poly	lazy	higher	pattern
LISP family						
Common Lisp	Committee	[132]	dynamic	eager	higher	access
Scheme	Committee	[121]	dynamic	eager	higher	access
Parallel and Concurrent languages						
Erlang	Ericsson	[6]	dynamic	eager	first	pattern
FACILE	ECRC	[134]	strong, poly	eager	higher	pattern
ID	MIT	[108]	strong, poly	non-strict eager	higher	pattern
Sisal	LLNL	[104]	strong, mono	non-strict eager	first	none
Intermediate languages						
ΓCMC	Recife	[95]	strong, poly	lazy	higher	access
Stoffel	Amsterdam	[13]	strong, poly	lazy	higher	case
Other functional languages						
Epic	CWI	[145]	strong, param	eager	first	pattern
Opal	TU Berlin	[42]	strong, param	eager	higher	pattern
Trafola	Saarbrücken	[3]	strong, poly	eager	higher	pattern

Table 1: Language characteristics.

Program	Description	Origin
anna	strictness analyser	Julian Seward (Manchester)
calc	arbitrary-precision calculator	Liang & Mirani (Yale)
compress	text compression	Paul Sanders (BT)
fluid	fluid-dynamics program	Xiaoming Zhang (Swansea)
gamteb	Monte Carlo photon transport	Pat Fasel (Los Alamos)
gg	graphs from GRIP statistics	Iain Checkland (York)
hpg	Haskell program generator	Nick North (NPL)
infer	Hindley-Milner type inference	Phil Wadler (Glasgow)
lift	fully-lazy lambda lifter	David Lester (Manchester) & Simon Peyton Jones (Glasgow)
maillist	mailing-list generator	Paul Hudak (Yale)
mkhprog	Haskell program skeletons	Nick North (NPL)
parser	partial Haskell parser	Julian Seward (Manchester)
pic	particle in cell	Pat Fasel (Los Alamos)
prolog	mini-Prolog interpreter	Mark Jones (Oxford)
reptile	Escher tiling program	Sandra Foubister (York)
veritas	theorem-prover	Gareth Howells (Kent)

Table 2: Real Subset of the NoFib Suite

5.3.2 The Haskell NoFib Suite

The very important paper [112] on benchmarking by Will Partain shows the relevance of having test programs that really reflect the kind of operations that are used by people in the "real world". His motivation was to give the FP community a set of benchmarks to "encourage implementors to tacle the problems that actually make Haskell programs large and slow, thus hastening solutions to those problems". The NoFib programs are divided into three subsets: Real, Imaginary, and Spectral (somewhere between Real and Imaginary).

The Real subset, which appears on table 2 is considered the most important and observe the following criteria:

- Written in standard Haskell.
- Written by someone trying to get a job done, not by someone trying to make a pedagogical or stylistic point.
- Performs some useful task such that someone other than the author might want to execute the program for other that watch-a-demo reasons.
- Neither implausibly small or impossibly large (the Glasgow Haskell compiler, written in Haskell, fall in the later category).
- The run time and space for the compiled program must neither be too small (e.g., time less than five secs.) or too large (e.g., such that a research student in a typical academic setting could not run it).

The Spectral subset tend to fall into Hennessy and Patterson's category of "kernel" benchmarks, being "small, key pieces from real programs" [64]. The Imaginary subset of the suite is the usual toy benchmarks such as *primes*, *queens*, *quicksort*, etc.

6 FP and Parallelism in the 1990s

As we mentioned before, only recently one can consider that we have learned how to compile functional languages for uniprocessor machines. The experience gained with the machines built in the 1980s proved that building special purpose hardware is costly and, most of times, is too slow a process to meet the advances of commercial general purpose machines. This is the main reason why parallel functional programming has concentrated in language implementation on commercially available hardware. Parallel garbage collection algorithms are also sought as ways to improve the performance of parallel programs.

Recent implementations of parallel functional languages are being built on top of fast sequential functional compilers. A number of ideas are being experimented. Implementations of skeletons are starting to appear [39, 21], which seem to be promising. ZAPP has achieved good results for divide-and-conquer parallelism on networks of transputers [54, 101]. Some recent approaches to exploit data parallelism in functional languages are POD comprehensions [65], which aim to combine data parallelism with lazy evaluation, bidirectional fold and scan [111], and the data parallel language NESL, which provides a mechanism for nesting parallelism [19, 20].

Two new versions of Haskell seem to be very promising: Concurrent Haskell [117] and GUM [137]. Concurrent Haskell is a concurrent extension to Haskell, which aims to provide a more expressive substrate upon which to build sophisticated I/O-performing programs, notably ones that support graphical user interfaces for which the usefulness of concurrency is well established. The goal of the designers of Concurrent Haskell is to attain implicit, semantically transparent parallelism, but the version available now uses explicit parallelism.

GUM is a portable, parallel implementation of Haskell, which uses the PVM communications harness. As a result, GUM is available both on shared-memory (Sun SPARCserver multiprocessors) and distributed-memory (networks of workstations) architectures. The high message-latency of distributed machines is ameliorated by sending messages asynchronously, and by sending large packets of related data in each message. Initial performance figures demonstrate absolute speedups relative to the best sequential compiler technology.

Parallel functional programming is a broad area [99]. Roe's thesis [123] presents a useful introduction up to 1991 with examples of different styles of parallelism. Schreiner's annotated bibliography [126], with over 400 entries, is relatively complete and highly useful. The works of Ben-Dyke [16] and Hammond [59], despite their overoptimism sometimes, provide useful guidelines to the area.

6.1 Parallel Garbage Collection

Garbage collection [77] has gained importance with its use by object-oriented languages and a wide variety of software platforms and tools.

The parallel garbage collection algorithms developed in the 1970s and 1980s for shared memory architectures are still being implemented in the machines built now. An efficient algorithm for concurrent collection was proposed by Appel-Ellis-Li [5] for machines that support virtual memory. This algorithm uses paging information provided by the operating system to synchronise the operations between processors. A number of optimisations were made to Appel-Ellis-Li's algorithm [5] and these seem to be one of the most promising directions in the area [77].

The previously mentioned algorithms are based on marking and scanning the workspace. A different strategy is offered by reference counting [34]. The first reference counting algorithm for shared memory architectures is presented by Kakuta-Nakamura-Iida in reference [79]. A much simpler architecture was proposed by Lins [91] and later generalised into a multiprocessor architecture described in [92].

Mark-scan has also been used in distributed architectures. Hudak and Keller's algorithm [69] works in two steps. First, all processors cooperate to mark all accessible cells, and then all processors collect unmarked cells. Ali [2] presents algorithms which allow each processor to mark-scan its own heap independently. At the end of such a local garbage collection the processor informs all other processors which remote pointers it retains, and the other processors then treat these as roots that must be marked during their own garbage collection. Hughes [72] gives an algorithm which has lower storage overheads than Ali's, but is also likely to take longer to recover remotely-referenced garbage. Neither Ali's nor Hughes' algorithms are truly real-time since any particular computation may be delayed for a long time while its processor does a garbage collection. The algorithm described in reference [129] uses mark-scan locally in each processor. External references are avoided by transferring an object (data) between processors whenever a local mark-scan discovers that there are no locally held references to the object, thereby increasing the locality of objects. This algorithm can present a very high communication overhead, depending on the size of the objects.

Weighted Reference Counting make reference counting suitable for use in loosely-coupled multiprocessor architectures [18, 148]. A cyclic version of it is presented in [94].

Garbage collection in distributed systems is still in the very beginning [77]. Hierarchical schemes seem to be more promising in delivering good performance.

7 Some like it HOT

Computer science, as all branches of human knowledge, is sensitive to changes in the "environment". As we argued before, the cold war had a huge influence in the development of formal methods and functional programming. The same kind of influences occur within sciences. The development of an area in computer science does not ignore the evolution in other areas. The functional programming paradigm was highly influential to the other paradigms. There is no room for dogmas in science. Pragmatism must always be central to scientific matters. In the views of people concerned with the correctness of programs, assignments and jumps make their proof extremely more difficult. It is not the case that programmers that make use of languages with explicit control flow commands will be send to hell, a place where programs never work, regardless of the debugging effort made. In a recent editorial of the Journal of

Functional Programming, P. Wadler [143] says:

> "Bob Harper coined the acronym HOT to summarise much of what functional programmers have to offer the world: expertise in languages that are Higher-Order and Typed. Bob argued for a broad interpretation of these terms, so that Higher-Order includes languages where objects contain methods (even though functions are not first-class citizens), and Typed includes both static and dynamic typing. By these criteria Java is HOT, and so are Haskell, ML and Scheme".

Another general trend in programming languages is the adoption of small, domain specific languages, resurrecting the ideas of Grace Hopper. Ericsson has designed the functional and concurrent language Erlang [6] to program phone switches. Erlang is a first-order language with dynamic types. Despite the fact that Erlang was byte-code interpreted and slow, Ericsson observed a large speed-up in programming time, such that they are investing in the development of a fast native code compiler.

Natural Expert by Software AG is a product that supports the development of knowledge-based systems. It includes a functional language called Natural Expert Language [73], which is reported as being the first successful implementation of a lazy functional language in a database-oriented mainframe environment. It is in use today by more than 20 customer sites on a variety of platforms, and the applications are in every day use by several hundred people. One of the customers thought that Natural Expert increased their productivity by a factor of ten.

8 Conclusions

The functional programming paradigm has been highly important and influential to many branches of computer science. No doubt it still has a lot to offer to software engineering:

- It is still easier to prove functional programs correct than in any other paradigm. We envisage that in the near future proof assistant tools will make this task much easier.
- Functional programs are executable specifications that provide prototypes for free.
- Program transformation is much easier due to the "algebraic" nature of functions. This may allow new code optimisation mechanisms to bring better performance in sequential and parallel architectures.

Although no "killer application" for functional programming has been found so far, there is a growing number of cases in which its adoption in the real-world was advantageous.

9 Acknowledgements

This work was sponsored by CNPq (Brazil) research grants 52.2248/95-0, 52.2429/94-7 and FINEP-RECOPE.

References

[1] H. H. Aiken and G. M. Hopper, *The Automatic Sequence Controlled Calculator*. Electrical Engineering 65, pp. 384-391, 449-454, 522-528, 1946

[2] K.A.M.Ali. *Object-Oriented Storage Management and Garbage Collection in Distributed Processing Systems*. PhD thesis, Royal Institute of Technology, Stockholm, December 1984.

[3] M. Alt, C. Fecht, C. Ferdinand, and R. Wilhelm. The Trafola-S subsystem. In B. Hoffmann and B. Krieg-Brückner, editors, *Program development by specification and transformation, LNCS 680*, pages 539–576. Springer-Verlag, Berlin, May 1993.

[4] P. Anderson, C. L. Hankin, P. R. J. Kelly, P. E. Osmon, and M. J. Shute. COBWEB-2: Structured Specification of a Wafer Scale Supercomputer. In *PARLE '87*, pages 51-67. Springer-Verlag LNCS 258, 1987.

[5] A.W. Appel, J.R. Ellis, and K.Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[6] J. Armstrong, M. Williams, and R. Virding. *Concurrent programming in Erlang*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[7] Arvind, K.P.Gostelow, W.Plouffe. An Asynchronous Programming Language and Computing Machine, University of California at Irvine, December 1978.

[8] Arvind, V. Kathail, and K. K. Pingali. A Dataflow Architecture with Tagged Tokens. Technical Report LCS Memo TM-174, MIT, 1980.

[9] Arvind and R. S. Nikhil. Can Dataflow Subsume von Neumann Computing? Technical Report CSG Memo 292, MIT, November 1988.

[10] E.A.Ashcroft, W.W.Wadge. LUCID, a non-procedural language with iteration, CACM, Vol 20(7):519-526, July 1997.

[11] J.Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Comm. ACM* 21 (8): 613-641, August 1978.

[12] H.P. Barendregt *The Lambda Calculus its Syntax and Semantics Studies in Logic and the Foundation of Mathematics* North-Holland (1984 - 2nd. ed.)

[13] M. Beemster. The lazy functional intermediate language Stoffel. Technical report CS-92-16, Dept. of Comp. Sys, Univ. of Amsterdam, Dec 1992.

[14] M.Ben-Ari. On-the-fly garbage collection: new algorithms inspired by program proofs. *Automata, languages and programming. Ninth colloquium*, 14–22, Springer-Verlag, 1982.

[15] M.Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.

[16] A. D. Ben-Dyke. The History of Parallel Functional Programming. FTPable from ftp.cs.bham.ac.uk, August 1994.

[17] K.J. Berkling. Reduction Languages for Reduction Machines. In 2nd. Annual ACM Symp. on Comp. Arch., pages 133-140, 1975.

[18] D.I.Bevan. Distributed garbage collection using reference counting. In *PARLE Parallel Architectures and Languages Europe*, pages 176–187. Springer Verlag, LNCS 259, June 1987.

[19] G. E. Blelloch. NESL: A Nested Data-Parallel Language (Version 2.6). Technical Report CMUCS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[20] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J.Sipelstein, and M. Zagha. Implementation of a Portable Nested Data-Parallel Language. In *Principles and Practices of Parallel Programming*, pages 102-111, 1993.

[21] T. A. Bratvold. A Skeleton-Based Parallelising Compiler for ML. In (1O4), pages 23-34.

[22] W. H. Burge. *McG - A Functional Programming System*, IBM Research Division, Yorktown Heights, August 1968.

[23] W. H. Burge. *Recursive Programming Techniques.* Addison-Wesley, 1975.

[24] G. L. Burn. *Lazy Functional Languages. Abstract Interpretation and Compilation.* Research Monographs in Parallel and Distributed Computing. Pitman, 1991.

[25] G. L. Burn. Evaluation Transformers - A Model for the Parallel Evaluation of Functional Languages (Extended Abstract). In *FPCA '87*, pages 446-470. Springer-Verlag LNCS 274, 1987.

[26] G.L.Burns, S.L.Peyton Jones and J.D.Robson. The spineless g-machine. In *Proc.ACM Conference on Lisp and Functional Programming*, pages 244–258, S-nowbird, USA, 1988.

[27] R. M. Burstall, D. B. MacQueen, and D. T, Sannella. Hope: An Experimental Applicative Language, Proceedings LISP Conference, Stanford 1980.

[28] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. *In FPCA '81*, pages 187-194, 1981.

[29] Warren Burton. Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs, ACM *TOPLAS*, 6(2),1984.20

[30] D. C. Cann. The optimizing SISAL compiler: version 12.0. Manual UCRL-MA-110080, Lawrence Livermore National Laboratory, Livermore, California, Apr 1992.

[31] L.Cardelli. The functional abstract machine. *Polymorphism*, 1, 1983.

[32] A.Church, A set of postulates for the foundation of logic, *Annals of Math.* (2)33, pp.346–366.

[33] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[34] G.E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, Dec. 1960.

[35] G.Cousineau P-L.Curien and M.Mauny. The categorical abstract machine. In J-P.Jouannaud, editor, *Functional Programming Languages and Computer Architecture.* SLNCS 201, 1985.

[36] S. Cox, S.-Y. Huang, P. H. J. Kelly, J. Liu, and F. Taylor. Program Transformation for Static Process Network. In *PARLE '92*, pages 497-512. Springer-Verlag LNCS 605, 1992.

[37] D. Culler and Arvind. Resource Requirements of Dataflow Programs. In *15th. Annual ACM Symp.* on Comp. Arch., 1988.

[38] P-L.Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science. Pitman Publishing Ltd., 1986.

[39] J. Darlington, A, J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel Programming using Skeleton Functions. In *PARLE '93*, pages 146-160. Springer-Verlag LNCS 694, 1993.

[40] J. Darlington and M. J. Reeve. ALICE: A Multiple-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages. In *FPCA '81*, pages 65-76, 1981.

[41] M. K. Davis and R, J. M. Hughes, editors. *Glasgow Workshop on Functional Programming.* Springer-Verlag WICS, 1989.

[42] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and Implementation of an Algebraic Programming Language, In J. Gutknecht, editor, Programming *Languages and System Architectures, Zurich, Switzerland*, pages 228-244. Springer-Verlag LNCS 782, March 1994.

[43] E.W. Dijkstra. Notes on a real time garbage collection system. From a conversation with D.E.Knuth (private collection of D.E.Knuth), 1975.

[44] E.W. Dijkstra, L.Lamport, A.J. Martin, C.S. Scholten, and E.F.M.Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46.* Springer-Verlag, New York, 1976.

[45] E.W. Dijkstra, L.Lamport, A.J. Martin, C.S. Scholten, and E.F.M.Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[46] J. Darlington et al. An Introduction to the FLAGSHIP Programing Environment. In *CONPAR '88, Manchester*. Cambridge University Press, 1988.

[47] A. Evans. *A language designed for teaching programming linguistics*, Proceeding of the ACM National Conference, 1968.

[48] J.Fairbairn and S.Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *Proceedings of Third International Conference on Fuctinal Programming and Computer Architecture*, pages 34–45. LNCS 274, Springer Verlag, 1987.

[49] J. M. Foster. *List Processing*, American Elsevier Publishing Company, Inc., 1967.

[50] P. Hartel et al. Benchmarking implementations of functional languages with 'Pseudoknot', a float-intensive benchmark. *J. of Functional Programming*, vol 6(4):621-657, Cambridge University Press, 1996.

[51] C.Ghezzi and M.Jazayeri, *Programming Language Concepts*. John Wiley and Sons, 1982.

[52] B. Goldberg and P. Hudak. Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor. In *Workshop* on Graph *Reduction, Santa FE, New Mexico*, pages 94-113. Springer-Verlag LNCS 279, September 1986.

[53] B. F. Goldberg. Multiprocessor Execution of Functional Programs. Intl. *Journal of Parallel* Programming, 17(5):425- 473,1988.

[54] R. G. Goldsmith, D. L. McBurney, and M. R. Sleep. Parallel Execution of Concurrent Clean on ZAPP. In [128], chapter 21.

[55] D.Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, Dec. 1977.

[56] D.Gries. *The Science of Programming* Springer Verlag, 1981.

[57] J. R. Gurd, C. C, Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Comm. ACM*, 28(1):34-52, January 1985.

[58] J. R. Gurd, C.C. Kirkham, and J. R. W. Glauert. A Multilayered Data Flow Computer Architecture. Technical report, Manchester University, 1978.

[59] K.Hammond. Parallel Functional Programming: An Introduction. FTPable from ftp.dcs.glasgow.ac.uk, August 1994.

[60] K. Hammond and J. T. O'Donnell, editors. *Glasgow Workshop on Functional Programming*. Springer-Verlag WICS, 1993.

[61] Chris L. Hankin, P. E. Osmon, and M. J. Shute. COBWEB - a combinator reduction architecture. In *FPCA '85*, pages 99-112, September 1985.

[62] P. G. Harrison and M. J. Reeve. The Parallel Graph Reduction Machine, Alice. In *Work-shop on Graph Reduction, Santa FE, New Mexico*, pages 181-202. Springer-Verlag LNCS 279, September 1986.

[63] P.Henderson. *Functional Programming: Application and Implementation*, Prentice-Hall, 1980.

[64] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[65] J. M. D. Hill. The AIM is Laziness in a Data Parallel Language, In [60], pages 83-99.

[66] A. W. Holt. *General Purpose Programming Systems.* Communications of the ACM 1, pp. 7-9, 1958.

[67] P. Hudak. Para-Functional Programming in Haskell. In Boleslaw K. B. K. Szymanski, editor,*Parallel Functional Languages and Compilers*, Fontier Series, chapter 5, pages 159-196. ACM Press, 1991.

[68] P. Hudak and B. Goldberg. Serial Combinators: "Optimal" Grains of Parallelism. In *FPCA '85*, pages 382-399, September 1985.

[69] P. Hudak and R. M. Keller. Garbage Collection and Task Deletion in Distributed Applicative Systems. In *ACM Symp. on Lisp and Functional Programming*, pages 168-178, 1982.

[70] P. Hudak, S. L. Peyton Jones, and P. L. Wadler. Report on the Programming Language Haskell: a Non-Strict, Purely Functional Language. *Special Issue of SIGPLAN Notices*, 16(5), May 1992.

[71] P. Hudak and L. Smith. Para-functional Programming: A Paradigm for Programming Multiprocessor Systems. In *ACM POPL*, pages 243-254, January 1986.

[72] R. J. M. Hughes. A Distributed Garbage Collection Algorithm. In *FPCA '85*, pages 256-272, September 1985.

[73] N. W. O. Hutchison, U.Neuhaus, M.S.-S., C.V.Hall. Natural Expert: a commercial functional programming environment Journal of Functional Programming, 7(2): 163-182, March 1997.

[74] K. E. Iverson. *A Programming Language.* John Wiley and Sons, 1962.

[75] T.Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweend, January 1987.

[76] M.P. Jones. *The implementation of the Gofer functional programming system.* Technical Report YALEU/DCS/RR-1030, Dept. of Computer Science, Yale University, 1994.

[77] R.E.Jones and R.D.Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley and Sons, 1996, ISBN 0 471 94148 4.

[78] H. G. Kahrimanian. *Analytic Differentiation by a Digital Computer.* 1954 Symposium, pp. 6-14.

[79] K.Kakuta, H.Nakamura, and S.Iida. Parallel reference counting algorithm. *Information Processing Letters*, 23(1):33–37, 1986.

[80] R. M, Keller, F.C.H. Lin, and J. Tanaka, Rediflow multiprocessing. In *IEEE Compcon*, pages 410-417, February 1984.

[81] P. Kelly. *Functional Programming for Loosely coupled Multiprocessors.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[82] J. M. Kewley and K. Glynn. Evaluation Annotations for Hope+. In [41], pages 329-337.

[83] D.E. Knuth. *The art of computer programming*, volume I: Fundamental algorithms, chapter 2. Addison-Wesley, Reading, Ma., 2nd edition, 1973.

[84] H.T. Kung and S.W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*, pages 120–131. IEEE, 1977.

[85] J.Lambek. From lambda-calculus to cartesian closed categories. In J.P.Seldin and J.R.Hindley, editors, *in To H.B.Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism.* Academic Press, 1980.

[86] L.Lamport. Garbage collection with multiple processes: An exercise in parallelism. In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 50–54, 1976.

[87] P.J.Landin. The mechanical evaluation of expressions, Computer Journal, 6(4):308-320, 1964.

[88] P. J. Landin. *The next 700 programming languages*, CACM Vol 9(3), pp 157–165, March 1966.

[89] R.D.Lins. Categorical multi-combinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 60–79. Springer-Verlag, September 1987. LNCS 274.

[90] R.D.Lins.

[91] R.D. Lins. A shared memory architecture for parallel cyclic reference counting. *Microprocessing and Microprogramming*, 34:31–35, September 1991.

[92] R.D. Lins. A multi-processor shared memory architecture for parallel cyclic reference counting. *Microprocessing and Microprogramming*, 35:563–568, September 1992.

[93] R.D.Lins, G.G.Cruz Neto & R.F.Lima. Implementing and Optimising ΓCMC, Proceedings of Euromicro'94, pp.353-361, IEEE Computer Society Press, Sep. 1994.

[94] R.D.Lins and R.E.Jones. Cyclic Weighted Reference Counting, Proceedings of *WP&DP'93*, K.Boyanov (editor), North-Holland, 1993.

[95] R.D.Lins & B.O.Lira. ΓCMC: A Novel Way of Implementing Functional Languages, *Journal of Programming Languages*, 1:19-39, Chapmann & Hall, January 1993.

[96] R.D.Lins, S.J.Thompson and S.Peyton Jones, On the Equivalence between CM-C and TIM, *Journal of Functional Programming*, 4(1):47-63, Cambridge University Press, January/1994.

[97] R.D.Lins & P.G.Soares. Some Performance Figures for the G-Machine and its Optimisations. Microprocessing and Microprogramming 37(1993) 163-166, North-Holland.

[98] R.D.Lins & S.J.Thompson. Implementing SASL using categorical multi-combinators. *Software — Practice and Experience*, 20(8):1137–1165, November 1990.

[99] R.D.Lins. Functional Programming and Parallel Processing (Invited Paper), Proceedings of Vecpar'96, LNCS 1215, pp. 429-457, Springer Verlag, 1997.

[100] G. A. Magoo and D. F. Stanat. The FFP Machine. In *High-Level Language Computer Architectures*, pages 430-468, 1989.

[101] D. L. McBurney and M. R. Sleep. Transputer-Based Experiments with the ZAP-P Architecture. In *PARLE '87*, pages 242 259. Springer-Verlag LNCS 258, 1987.

[102] J.McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[103] J. McCarthy. A basis for a mathematical theory of computation. *Proc. Western Joint Comp. Conf., Los Angeles, May 1961, 225-238, and Proc. IFIP Congress 1962*, North Holland Publ. Co., Amsterdam, 1963.

[104] J. R. McGraw *et al.*, S. K. Skedzielewski, S. Allan, R. Oldehoeft, J. R. W. Glauert, C. Kirkham, B. Noyce, and R. Thomas. Sisal: Streams and iteration in a single assignment language reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, California, Mar 1985.

[105] R.Milner. Standard ML proposal. *The ML/LCF/Hope Newsletter*, 1(3), January 1984.

[106] E.Moggi. Computational lambda calculus and monads, in Logic in Computer Science, California, IEEE Press, June 1989.

[107] M.A.Musicante & R.D.Lins. GMC: A Graph Multi-Combinator Machine. *Microprocessing and Microprogramming*, 31:31–35, North-Holland, April 1991.

[108] R. S. Nikhil. Id (version 90.1) reference manual. Technical Report CSG Memo 284-2, Lab. for Computer Science, MIT, July 1991.

[109] R. S. Nikhil, Arvind, and J. Hicks. pH Language Proposal (Preliminary), 1st. September 1993. Electronic communication ou comp.lang.functional.

[110] R. S. Nikhil G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *19th. ACM Annual Symp. on Comp. Arch.*, pages 156-167, 1992.

[111] J. T. O'Donnell. Bidirectional Fold and Scan. In [60], pages 193-200.

[112] W.Partain. *The nofib Benchmark Suite of Haskell Programs.* University of Glasgow, retrievable from ftp.dcs.glasgow.ac.uk in pub/haskell/glasgow.

[113] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, Laboratory for Computer Science, MIT, August 1988.

[114] S. L. Peyton Jones, C. Clack, and J. Salkid. High-Performance Parallel Graph Reduction. In *PARLE '89*, pages 193-206, Eindhoven, The Netherlands, June 12-16, 1989. Springer-Verlag LNCS 365,

[115] S. L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP - a High-Performance Architecture for Parallel Graph Reduction. In *FPCA '87*, pages 98-112. Springer-Verlag LNCS 274, 1987.

[116] S.L.Peyton Jones and J.Salkild. The spineless tagless g-machine. In *Proc.ACM Conference on Functional Programming Languages and Computer Architecture*, pages 184–201, Snowbird, USA, 1989.

[117] S.L.Peyton Jones, A.Gordon and S.Finne. Concurrent Haskell. In Proc. of 23rd ACM Symposium on Principles of Programming Languages (POPL'96), Florida, 1996.

[118] M. J. Plasmeijer and M. C. J. D. van Eekelen. *Concurrent Clean - version 1.0 - Language Reference Manual, draft version.* Dept. of Comp. Sci, Univ. of Nijmegen, The Netherlands, Jun 1994.

[119] F.J. Pollack, G.W. Cox, D.W. Hammerstein, K.C. Kahn, K.K. Lai, and J.R. Rattner. Supporting Ada memory management in the iAPX–432. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 117–131. SIGPLAN Notices (ACM) 17,4, 1982.

[120] J. C. Reynolds. *GEDANKEN: a simple typeless language which permits functional data structures and co-routines.* Argonne National Laboratory, September 1969.

[121] J. A. Rees and W. Clinger. *Revised^4 Report on the Algorithmic Language Scheme.* MIT, Cambridge, Massachusetts, Nov 1991.

[122] H. Richards. An Overview of Burroughs NORMA. Technical report, Austin Research Centre, Burroughs Corp., January 1985.

[123] P. Roe. Parallel Programming *using Functional Languages.* PhD thesis, Glasgow University, April 1991.

Univ., 1994.

[124] S. Rosen *Programming Systems and Languages — A Historical Survey.* Programming Systems and Languages, McGraw-Hill Book co., 1967.

[125] V. Sarkar and J. Hennessy. Partitioning Parallel Programs for Macro-Dataflow. In *ACM Symp. on Lisp and Functional Programming*, pages 202-211, 1986.

[126] W. Schreiner. Parallel Functional Programming - an Annotated Bibliography, Technical Report 93-24, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1993.

[127] D.Scott. Relating theories of the lambda-calculus. In J.P.Seldin and J.R.Hindley, editors, *in To H.B.Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism.* Academic Press, 1980.

[128] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors. Tem *Graph Rewriting: Theory and Practice.* Wiley, 1993.

[129] M.Shapiro, O.Gruber and D.Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Technical Report 1320, Rapports de Recherche, INRIA-Rocqencourt, Novembre 1990.

[130] G.L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[131] G.L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976.

[132] G. L. Steele Jr. *Common Lisp the Language.* Digital Press, Bedford, second edition, 1990.

[133] W. R. Stoye. The *Implementation of Functional Languages using Custom Hardware.* PhD thesis, University of Cambridge, 1985.

[134] B.Thomsen, L.Leth, S.Prasad, T.-S.Kuo, A.Kramer, F.Knabe, and A.Giacalone. *Facile antigua release - programming guide.* Technical report ECRC-93-20, European Industry Research Centre, Munich, Germany, 1993.

[135] S.J.Thompson & R.D.Lins. The Categorical Multi-Combinator Machine: CMCM, *The Computer Journal*, vol 35(2): 170-176, Cambridge University Press, April 1992.

[136] S.J.Thompson. Haskell: The craft of functional programming. Addison-Wesley, 1996

[137] P.W.Trinder, K.Hammond, J.S.Mattson Jr., A.S.Partridge. GUM: a portable parallel implementation of Haskell, Proc. of Programming Language Design and Implementation, Philadelfia, USA, May, 1996.

[138] D.A. Turner. A new implementation technique for applicative languages. *Software — Practice and Experience*, 9, 1979.

[139] D.A.Turner. *Aspects of the implementation of programming languages (the compilation of an applicative language to combinatory logic)*. Ph.D. thesis, University of Oxford, February 1981.

[140] D.A.Turner. *KRC manual*, The University of Kent at Canterbury, 1981.

[141] D.A.Turner.
Functional Programming as executable Specifications, *Phil.Transactions of the Royal Society of London 312*, pp.363-388, 1984.

[142] P.L. Wadler. Analysis of an algorithm for real-time garbage collection. *Communications of the ACM*, 19(9):491–500, September 1976.

[143] P. Wadler. A HOT opportunity. (Editorial), Journal of Functional Programming, 7(2): 127-128, March 1997.

[144] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*, Oxford University D.Phil. Thesis, 1971.

[145] H.R.Walters and J.F.Th.Kamperman. Epic: Implementing language processors by equational programs. Technical Report in preparation. Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

[146] I. Watson. Simulation of a Physical EDS Machine Architecture. Technical report, Department of Computer Science, University of Manchester, UK, September 1989.

[147] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J, Sargeant. Flagship: a Parallel Architecture for Declarative Programming. In *15th. Annual ACM Symp, on Comp. Arch.*, page 124, 1988.

[148] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architectures. In *PARLE '87*, pages 432-443. Springer Verlag LNCS 259, 1987.

[149] P. Watson and I. Watson. Evaluating Functional Programs on the FLAGSHIP Machine. In *FPCA '87*, pages 80-97. Springer-Verlag LNCS 274, September 1987.

[150] P.Weis and X.Leroy. *Le Langage CAML*. InterÉditions.

[151] E. P. Wentworth. RUFL reference manual. Technical report 92/1, Dept. of Comp. Sci, Rhodes Univ., Jan 1992.

[152] A.N.Whitehead and B.Russell. *Principia Mathematica*; Cambridge (1910-1913) (2nd edition 1925-1927).

[153] T. Yuba, T. Shimada, K. Hiraki, and H. Kashiwagi. SIGMA-1: A Dataflow Computer for Scientific Computations. *Computer Physics Communications*, pages 141-148, 1985.