NASA Technical Memorandum 89019

A Survey of Functional Programming Language Principles

C. Michael Holloway

September 1986

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665

TABLE OF CONTENTS

1. INTRODUCTION

Research in the area of functional programming languages has been conducted for a number of years. The amount of this research has increased greatly in the past eight years since John Backus' Turing Award Lecture on the topic was published[1]. Despite this attention by the research community, the average programmer is either completely unaware of the ideas of functional languages, or, if he has some familiarity with the topic, believes such languages to be impossible to understand and devoid of practical value.

The literature on the subject has contributed little to altering this state of affairs. Most of it is either difficult to find or extremely difficult to comprehend. The purpose of this paper is to present a survey of the topic of functional languages that is both comprehensive and understandable. The paper assumes the reader has a knowledge of the basic principles of traditional programming languages, and is comfortable with mathematics, but does not assume any prior knowledge of the ideas of functional languages.

The organization of the paper is as follows. First, the basic principles of functional languages are discussed. A

---

[1] John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," **Communications of the ACM**, Vol 21, No 8, Aug 1978, pp 613-641.

definition of the term "functional language" is given, and a simple functional language is defined using the language style introduced by Backus. Also, the relationship of Backus' style and the lambda style of functional language is discussed.

Second, the reasons for developing functional languages are discussed. The statements of the advocates of the language style are presented without comment on the validity of the claims.

Third, the issues involved in the language style are examined. Topics discussed include methods of expressing concurrency, the algebra of functional languages, program transformation techniques, the inclusion of data types in functional languages, the technique of lazy evaluation, the implementation of functional languages on conventional machines, and new architectures specifically designed to support functional languages.

Fourth, several existing languages that are claimed to be functional are mentioned briefly. Fifth, and finally, comments and opinions as to the future practical value of functional programming languages are given.

2. BASIC PRINCIPLES OF FUNCTIONAL LANGUAGES

Many different opinions as to what constitutes a functional programming language exist. Functional languages have been called "assignment-less" and "variable-less" languages[2]. Just as calling structured programming languages "goto-less" languages is an oversimplification, so too are such descriptions of functional languages. The definition that will be used in this paper is discussed below.

2.1 Definition of "Functional Language"

The major distinguishing characteristic of functional languages is the approach to problem solving they encourage. Traditional programming languages such as FORTRAN, Pascal, and Ada[3] support an approach that can be described by this statement:

> A program in an Imperative Language is used to convey a list of commands to be executed in some particular order, such that on completion of the commands the required behavior has been produced.[4]

In other words, a traditional language program is simply a set of commands.

---

[2] Bruce J. MacLennan, "A Simple Software Environment Based on Objects and Relations," **85 Symposium on Language Issues in Programming Environments**, pp 199.

[3] Ada is a registered trademark of the U. S. Government: Ada Joint Program Office.

[4] Hugh Glaser, Chris Hankin, David Till, **Principles of Functional Programming** (Prentice-Hall International: London, 1984), p 4.

The functional approach is different. It can be categorized by the following statement:

> A program in a Functional Language is used to define an expression which is the solution to a set of problems; this definition can then be used by a machine to produce an answer to a particular problem from the set of problems.[1]

That is, a functional program is not a series of commands. Instead, it is an expression that represents a function that takes a particular set of objects (such as a list of names) and produces another set of objects (such as a list of phone numbers)[2]. A functional programming language is a language in which such programs are required or encouraged.

2.2 A Simple Functional Language

The definitions given above do not adequately convey the nature of functional languages. The best way to illustrate the functional language approach is to present a simple language taking the approach. The informal language system introduced by Backus in his Turing Award paper is suitable for defining such a language. The particular language that will be defined lacks several features, that make it inadequate for use in practical applications, but it is sufficient for illustration.

The languages introduced by Backus (called FP languages) are made up of five elements: objects, the "application" operator, functions, functional forms, and definitions.

---

[1] **Glaser, et al.**, p 4.

[2] John H. Williams, "Notes on the FP Style of Functional Programming", in J. Darlington, Peter Henderson, and D. A. Turner, editors, **Functional Programming and its Applications** (Cambridge University Press: Cambridge, England, 1982), p 73.

Choosing the components of these elements defines a single FP language. Below, each of the five elements is discussed, and, using these elements, a simple language called "ExFP" is defined[1].

The first necessary constituent of an FP language is a set of objects. An object can be one of three entities: a single atom, a sequence of atoms, or the undefined element $\perp$ (called "bottom"). The set of atoms may be chosen as desired. For ExFP, the set is chosen as all strings made up of letters, digits, and other characters not used otherwise in the language. The atom T will represent the boolean value true, and the atom F the value false.

A sequence is denoted by

$$\langle x_1 , x_2 , \ldots , x_n \rangle$$

where each x_j is an object. The empty sequence is represented by ϕ, which is both an atom and a sequence. Any sequence with $\perp$ is identical to $\perp$ itself.

Another part of an FP language is the single operator "application" denoted by :. For a function f and the object x, $f:x$ represents the object resulting from applying f to x.

Primitive functions make up another constituent of an FP

---

[1] This discussion is based on **Backus, CACM**, pp 620-622, and John Backus, "Function-level Computing", **IEEE Spectrum**, Vol 19, No 8, Aug 1982, p24. See also, **Glaser, et al.**, pp 195-202, and **Williams** in **Darlington, et al.**, pp 73-77. ExFP is similar to Backus' specific FP language, but has fewer primitive functions and functional forms.

language. These functions can be defined as desired. For ExFP, the following are defined[1]:

1. **Identity function**: this function produces its argument. For example,
   id:<7, 6, 5> = <7, 6, 5>

2. **Selector functions**: the selector function i gives the i^{th} element of its object, if that element exists, and $\perp$ otherwise. For example,
   3:<7, 6, 5> = 5
   4:<7, 6, 5> = $\perp$

3. **Tail**: this function produces the object sequence with its first element removed. If the object to which **tail** is applied is a sequence with one object (a singleton), ϕ is produced. If the object is not a sequence, $\perp$ is produced. For example,
   tail:<7, 6, 5> = <6, 5>
   tail:<7> = ϕ
   tail:7 = $\perp$

4. **Distribute functions**: these functions produce a sequence of pairs[2] of objects from a pair of objects. **distr** distributes from the right and requires the first object of the pair to be a sequence. **distl** distributes from the left and requires the second object to be a sequence. For both forms, if the requirement is not met, $\perp$ is produced. For example,
   distr:<<a, m>, 7> = <<a, 7>, <m, 7>>
   distl:<7, <a, m>> = <<7, a>, <7, m>>
   distr:<7, 7> = $\perp$

5. **Append functions**: these functions produce a sequence from a pair consisting of a sequence and an object. **appendr** requires the first object to be a sequence. **appendl** requires the second object to be a sequence. If the argument does not conform, $\perp$ is produced. For example,
   appendr:<<7, 6>, 5> = <7, 6, 5>
   appendr:<7, <6, 5>> = $\perp$
   appendl:<<7>, <6, 5>> = <<7>, 6, 5>

6. **Arithmetic functions**: these are the standard functions +, -, ÷, and * (for multiplication). The object to which any of these functions is applied must be a pair,

---

[1] Note: Any function applied to $\perp$ yields $\perp$.

[2] A pair is a sequence consisting of 2 objects.

each of which evaluates to a number; otherwise, ⊥ is produced. For example,

+:<7, 3> = 10
*:<7, 3> = 21
+:7 = ⊥
÷:<7, 0> = ⊥

7. **Eq**: this function produces T if its object consists of a pair of identical objects, F if the object consists of a pair of different objects, and ⊥ otherwise. For example,

eq:<7, 7> = T
eq:<7, 6> = F
eq:<7, 7, 7> = ⊥

8. **Boolean functions**: these are the standard boolean functions **and**, **or**, and **not**. The object to which either **and** or **or** is applied must be a pair of objects, each of which evaluates to either T or F. The object to which **not** is applied must be a singleton whose element evaluates to either T or F. If these conditions are not met, ⊥ is produced. For example,

and:<T, F> = F
or:<T, F> = T
not:<F> = T
and:<7, 6> = ⊥

Additional primitive functions have been defined by Backus, but these are sufficient for the example language.

In addition to the primitive functions, an FP language provides functional forms by which new functions can be built from existing ones. Functional forms, also called combining forms, are expressions denoting functions. They are produced using what Backus calls program-forming operations (or, PFOs, for short). Listed below are the PFOs and the resulting functional forms that are included in ExFP[1]:

---

[1] In the discussion, ≡ means "is defined as".

1. **Constant**: this is denoted by % in front of an object[1]; it yields that object, except if it is applied to $\perp$, in which case $\perp$ is produced. Formally, that is

$$\%y:x \equiv \perp \text{ if } x = \perp$$
$$\equiv y \text{ otherwise}$$

2. **Composition**: denoted by $\circ$ between two functions; the composition of two functions f and g applied to a function x is equivalent to applying f to the result of applying g to x. That is

$$(f \circ g):x \equiv f:(g:x)$$

3. **Construction**: this is denoted by [ ] enclosing a group of functions; a construction applied to an object x is equivalent to the sequence produced by applying each function in the construction to x. That is

$$[f_1, \ldots, f_n]:x \equiv \langle f_1:x, \ldots, f_n:x \rangle$$

4. **Apply to all**: this PFO is denoted by α in front of a function; if the object to which the function is applied is a sequence, the form is equivalent to the sequence produced by applying the function to each object in the sequence; if the object is not a sequence, the form is equivalent to $\perp$. That is

$$\alpha f:x \equiv \langle f:x_1, \ldots, f:x_n \rangle \text{ if } x \text{ is a sequence}$$
$$\equiv \perp \text{ if } x \text{ is not a sequence}$$

5. **Insert**: denoted by / in front of a function; if the object to which the function is applied is not a sequence, the form is equivalent to $\perp$; if the object is a sequence with a single member, the form is equivalent to that member; if it is a sequence with at least two members, the form is equivalent to applying the function to the sequence containing the first member of the original sequence followed by the **insert** form applied to the remainder of the sequence. Formally, that is

$$/f:x \equiv \perp \text{ if } x \text{ is not a sequence}$$
$$/f:\langle x_1, x_2, \ldots, x_n \rangle$$
$$\equiv x_1 \text{ if } n = 1$$
$$\equiv f:\langle x_1, /f:\langle x_2, \ldots, x_n \rangle \rangle \text{ if } n \geq 2$$

6. **Condition**: denoted by three functions in the form (p

---

[1] This notation is from Scott E. Baden, "Berkely FP User's Manual, Rev. 4.1", **UNIX Programmer's Manual: Supplementary Documents**, 1980, p 28. Backus uses a bar over the object instead.

->f; g) applied to an object x[1]; to evaluate a **condition** p:x is computed, if T is produced the form is equivalent to f:x; if F is produced the form is equivalent to g:x; if neither T nor F is produced, the **condition** is equivalent to ⊥. That is

$$(p \to f;g):x \equiv f:x \text{ if } p:x = T$$
$$\equiv g:x \text{ if } p:x = F$$
$$\equiv \bot \text{ otherwise}$$

7. **Loop**: this combining form is denoted by two functions in the form (p -> loop *f*) applied to an object x[2]; to evaluate this form, p:x is computed, if it yields T, the **loop** is equivalent to the form applied to *f* applied to x; if p:x = F, the **loop** is equivalent to x; otherwise the expression yields ⊥. Symbolically, that is

$$(p \to \text{loop } f):x$$
$$\equiv (p \to \text{loop } f):(f:x) \text{ if } p:x = T$$
$$\equiv x \text{ if } p:x = F$$
$$\equiv \bot \text{ otherwise}$$

Again, more functional forms can be defined, but these are sufficient for ExFP.

The final necessary part of an FP system is a way to define new functions. The syntax is the following:

Def l ≡ r

where l is the new function name and r is a function or

---

[1] Notation from Ibid. Backus uses a solid right arrow. The ()'s are necessary only when an expression would otherwise be ambiguous.

[2] This notation is quite different from Backus. He uses (while p *f*):x. The notation here seems more consistent with the **condition** combining form, since the -> denotes a test in both cases. The ()'s are necessary only when an expression would otherwise be ambiguous.

functional form[1]. For example, a function to return T if an object is equal to 0 and F otherwise could be defined in ExFP as

Def eq0 ≡ eq ° [id, %0]

The above discussion has shown the elements that are necessary to make up an FP language. The components of these elements have been chosen, yielding the specific language ExFP. The semantics of the language are completely specified by stating how to compute *f*:x for any function *f* and object x. This can be done as follows[2]:

1. If *f* is a primitive function, the function is applied as described in the language definition.

2. If *f* is a functional form, the description of the form is used to rewrite *f*:x. The resulting expression is then computed according to the semantic rules.

3. If *f* is a defined function, given by Def *f* ≡ r, r:x is computed using the semantic rules.

4. If *f* is neither a primitive function, a functional form, nor a defined function, or if the use of these rules continues infinitely for *f*:x, then the value ⊥ is assigned to *f*:x.

An example should illustrate the technique of application of these semantic rules. Given the function **length** defined in ExFP as[3]

Def length ≡ /+ ° α%1

---

[1] The form of definition is extended in John Backus, "The algebra of functional programs: Function level reasoning, linear equations and extended definitions," **Lecture Notes in Computer Science 107** (Springer-Verlag: Berlin, 1981), pp 27-37. That work is discussed in Section 4.4.2.

[2] **Backus, CACM**, pp 622.

[3] This function definition is taken from **Williams** in **Darlington, et al.**, p 78.

consider the computation of

length:<7, 6, 5>

The computation would proceed as follows:

1. The definition of **length** gives
   /+ ° α%1:<7, 6, 5>

2. **Composition** produces
   /+: α%1:<7, 6, 5>

3. **Apply to all** gives
   /+: <%1:7, %1,6, %1,5>

4. Application of **constant** yields
   /+: <1, 1, 1>

5. **Insert** produces
   +:<1, /+:<1, 1>>

6. The internal **insert** yields
   +:<1, +:<1, /+<1>>>

7. The remaining **insert** produces
   +:<1, +:<1, 1>>

8. Addition results in
   +:<1, 2>

9. The final addition yields
   3

Thus, length:<7, 6, 5> = 3, as desired.

As mentioned above, the informal functional language discussed above is not a production quality language. Nevertheless, its structure shows sufficiently the basic principles of functional languages based on Backus´ approach[1].

---

[1] The careful reader has surely noticed that this language has no assignment statement, and that the only variables are the defined function names. This fact is the basis for the "assignment-less" and "variable-less" definitions mentioned previously. See Section 5 for a discussion of some languages that do provide these features.

2.3 Relationship of the FP and Lambda Styles

Backus' style is not the only basis for functional programming languages. The other primary model is based on the lambda calculus developed by Church in the 1940's[1]. In fact, the most widely known functional language, LISP, is based, in part, on this style[2].

The primary difference between the FP and lambda styles is the way in which programs are developed[3]. As mentioned above, for FP-based languages such as ExFP, programs are created by using program forming operations to combine existing programs. Lambda style languages generally have only one PFO, called "lambda abstraction." In place of PFOs, the languages have a large number of object forming operations. These are the primary mechanisms for program building.

As an example of the consequences of this difference, consider the construction of a specific program[4]. Suppose the functions p, g, h, r, and s have been previously defined, and a function f is to be created. This program is to use p as a test. If it yields T, g is to be applied to the argument object; if the

---

[1] For a discussion of the development of the lambda calculus, see J. Barkley Rosser, "Highlights of the History of the Lambda-Calculus," **1982 Symposium on LISP & Functional Programming**, pp 216-225.

[2] See Herbert Stoyan, "Early LISP History (1956 - 1959)," **1984 Symposium on LISP & Functional Programming**, pp 299-310 for a discussion of the development of LISP.

[3] This discussion is based on **Backus, LNinCS107**, pp 8-13.

[4] The example problem and programs are from Ibid, pp 8,9.

application of p yields F, h is to be applied to the result of applying both r and s to the argument.

To construct f using ExFP, the combining forms **condition**, **composition**, and **construction** are used to combine the given programs. This yields the following program:

p -> g; h ° [r, s]

Using the lambda style, the given programs can not be combined directly. Instead, an object x must be created. This object is used to form p:x, g:x, r:x, and s:x, which are also objects. The objects r:x and s:x are combined to produce the object h(r:x, s:x). These three objects are then combined using the object combining form **condition**. This result is still an object. To make it into a function, **lambda abstraction** is used. Thus, the lambda equivalent to the ExFP program is the following:

Lam x.(p:x -> g:x; h(r:x, s:x))

where Lam is the **lambda abstraction**[1].

As another example, consider the ExFP program[2]

[r ° t, s ° t]

The equivalent lambda program is the following:

Lam y.<Lam x.(r:(t:x)):y, Lam x.(s:(t:x)):y>

These two examples should be sufficient to show that the consequence of the lack of program forming operations in lambda

---

[1] This is traditionally denoted by the Greek letter Lambda; however, the word processor being used for this paper cannot print that character.

[2] This example is from Ibid, p 10.

style languages is that programs are hard to read. FP style programs, with PFOs, are much easier to understand.

Another difference between the FP and lambda styles is in the number of arguments that a function can accept. In the FP style, functions can have only one argument; multiple arguments are expressed as elements of a single sequence. Lambda style languages, however, allow functions of more than one argument.

As an example of what this difference means, consider the composition of two functions h and g. Suppose g produces a pair <y, z>, and h is a function of two arguments in the lambda style. The expression of the composition of the two functions using the lambda style looks like the following:

Lam y.h(1:(g:x), 2:(g:x))

where 1 and 2 are **selector** functions as in ExFP.

In the FP style, h would not be of two arguments, but rather would be a function on pairs. Thus, the function f can be expressed in ExFP simply as

g ° h

Once again, the FP style program is much to understand.

Lambda style languages do have an advantage over FP style languages: they are more powerful, in a sense. The single PFO **lambda abstraction** is able to express any FP style PFO that can be devised.

Backus has suggested that this relationship is similar to that between FORTRAN and structured languages such as Pascal[1].

[1] Ibid, pp 12, 13.

Any structured statement can be modeled using if-then's and goto's. Nevertheless, writing clear programs is much easier in a language like Pascal than it is in FORTRAN. The same relationship exists between FP style languages and lambda style ones.

This brief discussion has highlighted the fundamental differences between the FP and lambda style of functional languages. Because FP style languages tend to produce programs that are easier to read, the remainder of the paper will concentrate on languages of that style[1]. Some comments will be made about lambda style languages in Section 5.[2].

---

[1] For more information on the lambda style, see K. J. Berkling and E. Fehr, "A modification of the Lambda-Calculus as a base for functional programming languages," **Lecture Notes in Computer Science, Volume 140** (Springer-Verlag: Berlin, 1982), pp 35-47, and W. H. Burge, **Recursive Programming Techniques** (Addison-Wesley: Reading, Mass., 1975). Also, consult the Annotated Bibliography for more references.

[2] Note: Some authors call lambda based languages "applicative languages" and FP based languages "functional languages"; other authors consider both styles to be a type of applicative language. For this reason, the term "applicative language" is generally avoided in this paper.

3. REASONS FOR FUNCTIONAL LANGUAGES

Clearly, functional languages are radically different from traditional programming languages. In fact, the primary motivation behind the research on functional languages is a dissatisfaction with traditional programming languages. John Backus wrote

> Programming languages appear to be in trouble. ... Each new language claims new and fashionable features, ... but the plain fact is that few languages make programming sufficiently cheaper or more reliable to justify the cost of producing and learning to use them. ... there is a desperate need for a powerful methodology to help us think about programs, and no conventional language even begins to meet that need. In fact, conventional languages create unnecessary confusion in the way we think about programs.[1]

Backus and other advocates of functional languages believe that such languages offer significant advantages over conventional ones[2]. Perhaps the best way to illustrate some of these claimed advantages of functional languages is an example that contrasts a traditional program with a functional one. A program to determine if a given object is an element of a vector is a suitable problem for such an example.

An Ada[3] program fragment to perform this operation might be written as

---

[1] **Backus, CACM**, p 614.

[2] As mentioned in the introduction, no judgments as to the reality of the advantages are made in this section; the claims are simply presented. The reader is encouraged to form his own opinion. For the author's opinions, see Section 6.

[3] The choice of Ada is arbitrary; almost any other traditional language could be used here, without changing the discussion.

```
found := false;
i := 1;
while (not found) and (i <= n) loop
     found := (obj = vect[i]);
     i := i + 1;
end loop;
put(found);
```

An ExFP program for the same operation might be written as

```
Def find ≡ /or ° αeq ° distl
```

Four major differences between the two programs exist[1]. First, the ExFP program's effect is easy to understand, assuming one understands the individual components. This is not true of the Ada fragment. In order to understand its effect, one must mentally or manually execute it.

Second, the functional program is built from the three existing programs **or**, **eq**, and **distl**. The Ada fragment has no such hierarchical structure[2].

Third, the ExFP program makes no mention of arguments. It can be used on any object, vector pair, and the vector can have any length. The Ada program fragment can only be used on the vector "vect" of length "n", with subscripts beginning at 1. In order to make the Ada fragment general, it must be embedded in a procedure, function, or package. This introduces the complexity of parameter passing issues.

---

[1] This discussion is based on **Backus, CACM**, pp 616-7, and **Backus, IEEE**, p 24. He uses a different problem, but the discussion is similar.

[2] Large Ada programs using procedure, functions, and packages do possess some such structure, but not to the degree that functional programs do.

Fourth, the **apply to all** combining form of the ExFP program expresses the inherent parallelism of the pair wise comparisons naturally. The Ada fragment does not; it gives the impression that the comparisons must be done sequentially. Recognizing implicit parallelism in traditional languages requires sophisticated techniques.

This example has pointed out several of the advantages that functional style languages appear to offer over traditional languages. Additional advantages have been suggested as well.

One such advantage that functional languages offer over traditional languages is that they operate in the same domain as the problem to be solved[1]. Programming problems involve changing one set of objects into another. To accomplish this task with traditional languages, all the objects must first be represented by data stored in certain memory locations. Manipulations are then done on these memory locations. In other words, instead of being concerned with objects, one is concerned with variables denoting memory locations. In contrast, functional languages are concerned with objects, as desired.

Another advantage of the functional language style is that a language possesses algebraic properties itself[2]. These

---

[1] John Backus, "Is computer science based on the wrong fundamental concept of 'program'?" in **Algorithmic Languages: Proceedings of the International Symposium on Algorithmic Languages** (North Holland Publishing Company: Amsterdam, 1981), pp 141-2.

[2] These properties are discussed more fully in the next section of the paper.

properties can be used in proofs of correctness, making such proofs much simpler than is currently the case. In traditional languages, only expressions have any mathematical properties. Statements, the major part of programs, have no such properties. This means that proofs of correctness must be done in a logical language different than the programming language. That is, reasoning about the language cannot be done in the language.

A final advantage that functional languages seem to offer over traditional ones is that they promise to be able to take better advantage of technological and architectural innovation[1]. Conventional style languages are tightly tied to the traditional computer design. Functional languages are not. In fact, since the conventional computer design does not appear to be suitable for efficient evaluation of functional languages[2], functional languages encourage the development of new architectures.

Seven claimed advantages of functional languages have been mentioned above. To summarize, these are the following:

functional language programs are easier to understand than non-functional ones;

functional languages allow the building of new programs from existing ones in a hierarchical fashion;

functional languages encourage the development of general programs;

---

[1] D. A. Turner, "Recursion Equations as a Programming Language" in **Darlington, et al.**, p 1. The research into new architectures is discussed in more detail in Section 4.7.

[2] Section 4.6 discusses implementations for conventional machines.

functional languages allow the expression of inherent parallelism more naturally than traditional languages;

functional languages operate in the same domain as programming problems;

functional languages possess algebraic properties that allow proofs of correctness to be conducted in the language itself;

functional languages encourage architectural innovation and are better suited to take advantage of such innovation than non-functional ones.

4. TOPICS IN FUNCTIONAL LANGUAGES

Whether or not one accepts the claimed advantages as true, that functional programming languages are different from conventional languages is clear. The differences require that functional language researchers address some topics not encountered by traditional language researchers, and that they address some conventional topics in unconventional ways. Several topics of this nature are discussed below. These are the following: methods of expressing parallelism; the algebra associated with a given functional language; techniques for transforming one program into another; methods for including data types in functional languages; the technique of lazy evaluation; implementation techniques on conventional computers; and, new computer architectures to support functional languages.

4.1 Parallelism in Functional Languages

The ability to express inherent parallelism was listed above as a claimed advantage of functional languages. In this section, methods of doing this in the simple language ExFP are discussed.

In ExFP, the **apply to all** and **construction** combining forms allow the expression of many inherently parallel operations. As an example of the use of **apply to all**, consider adding together each pair of a set of pairs of numbers. Conceptually, each of the pairs can be added in parallel. This problem can be solved in ExFP by defining the following program:

Def pairadd ≡ α+

To see how this simple program expresses parallelism, consider the computation of

pairadd:<<1,4>, <2, 7>, <7, 7>>

Substituting for the definition gives

α+:<<1,4>, <2, 7>, <7, 7>>

The definition of the α functional form yields

<+<1, 4>, +<2, 7>, +<7, 7>>

which expresses the parallelism of the problem clearly.

A similar example shows the use of the **construction** functional form. Consider a program that not only does the pair wise additions as above, but also does pair wise multiplication, subtraction, and division. Conceptually, all four of these can be done in parallel. If **pairmult**, **pairsub**, and **pairdiv** are defined in ways analogous to **pairadd**, an ExFP program for the operation can be written as the following:

Def pairarith = [pairadd, pairmult, pairsub, pairdiv]

Substitution of the definition into

pairarith:<<1, 4>, <2, 7>, <7, 7>

yields

```
[pairadd, pairmult, pairsub, pairdiv]:
                <<1, 4>, <2, 7>, <7, 7>
```

Evaluation of the **construction** combining form produces

```
<pairadd: <<1, 4>, <2, 7>, <7, 7>>,
 pairmult:<<1, 4>, <2, 7>, <7, 7>>,
 pairsub: <<1, 4>, <2, 7>, <7, 7>>,
 pairdiv: <<1, 4>, <2, 7>, <7, 7>>>
```

This clearly shows the intrinsic parallelism of the problem.

As defined, however, ExFP does not allow a large class of intrinsically parallel problems to be expressed clearly. As an example, consider the program **length**, defined above as

Def length ≡ /+ ° α%1

The computation of

length:<7, 6, 5, 4>

yields, after several steps,

+:<1, +:<1, +:<1, 1>>>

Conceptually, the computation could, at the same point, be done as

+:<+:<1, 1>, +:<1, 1>>

which exhibits a greater degree of parallelism.

In order to handle such computations, a new functional form can be defined, and ExFP extended to include it[1]. The intent is that this functional should not require computations to proceed entirely to the right as does **insert**. Instead, the functional should allow computations to be done in two parallel paths when possible. A suitable definition for such a combining form is the following:

> **Parinsert**: denoted by ¦¦ in front of a function; if the object to which the function is applied is not a sequence, the form is equivalent to ⊥; if the object is a sequence with a single member, the result is equivalent to that member; if the object is a sequence with a least two members, the form is equivalent to applying the function to the sequence containing the **parinsert** functional applied to the first half (where

[1] This discussion is based on **Williams** in **Darlington, et al.**, pp 79-82. The name and notation of the new functional form is different than that chosen by Williams, but the definition is the same.

half is defined as the smallest integer greater than or equal to the number of elements in the sequence divided by two) of the sequence, followed by the **parinsert** functional applied to the remainder of the sequence. That is

$$
\begin{aligned}
&||f:x \equiv \perp \text{ if } x \text{ is not a sequence}\\
&||f:\langle x_1, \ldots, x_n\rangle\\
&\qquad \equiv x_1 \text{ if } n = 1\\
&\qquad \equiv f:\langle ||f:\langle x_1, \ldots, x_m\rangle,\\
&\qquad\qquad ||f:\langle x_{m+1}, \ldots, x_n\rangle\rangle\\
&\qquad\quad \text{if } n \geq 2,\ m = \text{ceil}(n/2)
\end{aligned}
$$

To see how this works, consider redefining length as

Def length ≡ ||+ ° α%1

The computation of

length:<7, 6, 5, 4>

yields, after several steps

||+:<1, 1, 1, 1>

Continuing with the computation gives

```
+:<||+:<1, 1>, ||+:<1, 1>>
+:<+:<||+:<1>, ||+:<1>>,
   +:<||+:<1>, ||+:<1>>
+:<+:<1, 1>, +:<1, 1>>
```

which is what is desired.

This discussion has shown that a functional language as simple as ExFP can express a large class of intrinsically parallel problems easily. More advanced languages can be even more powerful in their ability to express parallelism.

4.2 The Algebra of Functional Programs

Another topic in functional language research is the algebraic properties of the languages. These properties allow reasoning about programs to be done in the language itself. Applying some other language, such as a predicate calculus, is

not necessary[1]. The intent in developing an algebra for a given functional language is to allow programmers to use the laws of the algebra to prove programs correct and to help develop new programs, without requiring them to know anything about the mathematical foundation of the algebra[2]. In order to examine the ideas of the algebra for programs, an algebra for ExFP programs is discussed below.

The algebra that will be defined can be broken into three categories of statements: laws, derived theorems, and expansion theorems. Laws are statements that can be proven directly from the definitions of the language's primitive functions and functional forms. Derived theorems are statements that can be proven from the laws of the algebra. Expansion theorems are statements that allow recursive programs to be converted to nonrecursive ones[3].

There are two types of laws: ones that hold for all objects, and ones that hold for a restricted class of objects. A law stating that the functions f and g are equivalent for all objects is written as $f \equiv g$.

---

[1] For information on techniques for reasoning with conventional languages, see David Gries, **The Science of Programming** (Springer-Verlag: New York, 1981). For comments on the need for a mathematical basis in languages, see George T. Ligler, "A Mathematical Approach to Language Design," **Second Symposium on Prin. of Prog. Langs.**, 1975, pp 41-53, and Dana Scott, "Mathematical Concepts in Programming Language Semantics," **AFIPS Conference Proceedings**, Vol 40, 1972, pp 225-234.

[2] **Backus, CACM**, p 624. For a discussion of the foundation of the algebra presented here, see p 630 of that paper.

[3] **Williams** in **Darlington, et al.**, p 83.

In order to express the second type of law, additional notation must be used. One such notation is the following[1]:

p => *f* ≡ g

This means that *f*:x ≡ g:x for all x such that p:x = T.

To express that a law holds for any x that is defined (that is, not equal to ⊥), the following definition can be made[2]:

Def defined ≡ %T

(Recall from the definition of the **constant** combining form that %T:x is equivalent to T if x does not equal ⊥, and is equivalent to ⊥ if x equals ⊥) Thus a law that states that two functions *f* and g are equivalent so long as *f*:x is not equal to ⊥ could be written as

defined ° *f* => *f* ≡ g

There are a large number of laws that can be written for ExFP. For purpose of example, three such laws will be stated and proven, and eight laws will be stated without proof.

The first law that will be proven is the following[3]:

1. [*f* ° h, g ° h] ≡ [*f*, g] ° h

A proof of this statement must show that it is true for all functions *f*, g, and h, and for all objects. To do this, the left hand side is applied to the general object x, as follows

[*f* ° h, g ° h] : x

---

[1] **Backus, CACM**, p 625. The notation used here is different. He uses two solid right arrows.

[2] Ibid.

[3] **Backus, CACM**, p 625, proves the law in the opposite direction.

From the definition of the **construction** combining form, this can be rewritten as

$$\langle (f \circ h):x,\ (g \circ h):x \rangle$$

By the definition of the **composition** form, this can in turn be rewritten as

$$\langle f:(h:x),\ g:(h:x) \rangle$$

But this expression is simply the right hand side of a **construction**, so it can be written as

$$[f,\ g]:(h:x)$$

This, in turn, is the right hand side of a **composition**. A final rewriting yields

$$([f,\ g] \circ h):x$$

Thus, the law has been proven.

The second law that will be proven is the following[1]:

2. $\alpha f \circ [g_1, \ldots, g_n] \equiv [f \circ g_1, \ldots, f \circ g_n]$

To prove this law, the two sides must be shown to be equivalent for all functions f, g_1, ..., g_n, and any object x. Thus, the right hand side is written as

$$\alpha f \circ [g_1, \ldots, g_n]:x$$

Applying in order the definitions of **composition**, **construction**, **apply to all**, **composition**, and **construction** yields the following proof:

$$\begin{aligned}
&\alpha f \circ [g_1, \ldots, g_n]:x && = \\
&\alpha f:\ [g_1, \ldots, g_n]:x && = \\
&\alpha f:\ \langle g_1:x, \ldots, g_n:x \rangle && = \\
&\langle f:g_1:x, \ldots, f:g_n:x \rangle && =
\end{aligned}$$

---

[1] The law is from **Backus, CACM**, p 625. The proof is original.

$<(f \circ g_1):x, \ldots, (f \circ g_n):x> =$
$[f \circ g_1, \ldots, f \circ g_n]:x$ as desired.

The final law that will be proven is the following[1]:

3. defined $\circ$ g => 1 $\circ$ [f, g] $\equiv$ f

A proof of this law must show that for any function f, any object x, and any function g not yielding $\perp$ when applied to x, the statement holds. The following does just that, using in order **composition**, **construction**, and the **selector function 1** in conjunction with the assumption that g is defined:

$1 \circ [f, g]:x =$
$1: [f, g]:x =$
$1: <f:x, g:x> =$
$f:x$ as desired.

Several other laws that can be proven in ways similar to those shown above are the following[2]:

4. $/f \circ [g_1, \ldots, g_n] \equiv$
$f \circ [g_1, /f \circ [g_1, \ldots, g_n]]$ for $n \geq 2$

5. $(p \rightarrow f;g) \circ h \equiv$
$p \circ h \rightarrow f \circ h;\ g \circ h$

6. $h \circ (p \rightarrow f;g) \equiv$
$p \rightarrow h \circ f;\ h \circ g$

7. $[f, (p \rightarrow g;h)] \equiv$
$p \rightarrow [f, g];\ [f;\ h]$

8. defined $\circ$ g => %x $\circ$ g $\equiv$ x

9. $f \rightarrow (f \rightarrow g;\ h);\ j \equiv$
$f \rightarrow g;\ j$

10. $\alpha f \circ$ appendl $\circ$ [g, h] $\equiv$
appendl $\circ$ [f $\circ$ g, $\alpha f \circ$ h]

---

[1] The law is from **Backus, LNinCS107**, p 7. The proof is original.

[2] The first 4 laws are from **Backus, LNinCS107**, p 7. The remainder are from **Williams** in **Darlington, et al.**, pp 84-5.

11. $/f \circ \text{appendl} \circ [g, h] \equiv f \circ [g, /f \circ h]$

As mentioned, many more laws can be inferred. These laws can then be used to derive theorems, which make up the second category of statements in the algebra. Once derived, the theorems can be used in the same way as the laws. The larger the body of laws and proven theorems, the easier the task of reasoning about programs becomes.

As an example of the derivation of a theorem, consider the following program:[1]

length ° appendl ° [tail, id]

This program computes the length of the sequence constructed by appending to the front of the argument sequence, the tail of that sequence. The result of applying the program to any sequence can be determined by using the algebraic laws described above.

First, the definition of **length** is used to rewrite the program as

/+ ° α%1 ° appendl ° [tail, id]

Law 10 can used with f = %1, g = tail, and h = id, to yield

/+ ° appendl ° [%1 ° tail, α%1 ° id]

Since **tail** is defined for any sequence, law 8 can be used with x = 1 and g = tail, to produce

/+ ° appendl ° [%1, α%1 ° id]

Application of Law 11 with f = +, g = %1, and h = α%1 ° id, gives

---

[1] This example is based on **Williams** in **Darlington, et al.**, p 85. He uses two general functions within the construction, but the form used here seems easier to understand.

$+ \circ [\%1, /+ \circ \alpha\%1 \circ id]$

Finally, the definition of **length** can be used to yield

$+ \circ [\%1, length \circ id]$

So, the program applied to a sequence always yields the length of the argument sequence plus one, as expected[1].

The final category of statements in the algebra is that of expansion theorems. Expansion theorems are too complex to discuss in any detail here. As an example, a linear expansion theorem is given below[2]:

IF

$f = p \rightarrow q; Hf$

where Hf is a function involving f, and $H:\perp = \perp$, and there exists an H_1 such that for all g, h, and j, $H(g \rightarrow h; j) \equiv H_1 g \rightarrow Hh; Hj$

THEN

$f = p \rightarrow q; \ldots; H_1^n p \rightarrow H^n q; \ldots$

The algebra of functional programs is a complex topic. This section has only touched on the basic ideas, but the presentation should provide sufficient groundwork for further study[3].

---

[1] If this does not seem correct, consult the definition of **appendl** given above.

[2] John H. Williams, "On the Development of the Algebra of Functional Programs," **ACM Transactions on Programming Languages and Systems**, Vol 4, No 4, Oct 1982, p 737. This same theorem is given in **Williams** in **Darlington, et al.**, p 84, **Backus, CACM**, p 627, and **Backus, LNinCS107**, p 26. See these references for more information on expansion theorems.

[3] Besides the references already mentioned in this section, the interested reader should see Toni A. Cohen and Thomas J. Myers, "Towards an Algebra of Nondeterministic Programs," **1982 Symposium on LISP and Functional Programming**, pp 29-36.

4.3 Program Transformation

One consequence of the algebra of programs is the possibility of transforming programs from one form to another using the algebra. The primary motivation behind program transformation is the observation that writing easily understood programs and writing efficient programs are often conflicting goals. The program transformation method of development is to first write a program that is clear and easy to understand, and then transform it into one that runs as efficiently as possible on the available hardware[1].

A transformation has the form

$$C, \quad P_{in} \text{ --» } P_{out}$$

where C is a set of conditions under which the transformation is valid, P_{in} is the template, and P_{out} is the program equivalent to the template[2].

The process of using a transformation consists of the following steps:

1. Recognition of a program or segment of program that matches a known template.
2. Verifying that the conditions for the application of the template are satisfied by the program or segment.
3. Converting the program or segment to the equivalent form given in the transformation.

---

[1] John Darlington, "Program Transformation", **Darlington et al.**, p 193.

[2] This representation uses a different notation from, but is equivalent to, the representation of John S. Givler and Richard B. Kieburtz, "Schema Recognition for Program Transformations," **1984 Symposium on LISP & Functional Programming**, p 75.

Clearly, the laws and theorems of the algebra of a functional language provide an initial set of transformations. For example, given the language ExFP and its laws given above, the following simple transformation can be stated:

$$\phi,\ h \circ (p \rightarrow f; g) \twoheadrightarrow p \rightarrow (h \circ f);\ (h \circ g)$$

where the ϕ signifies that there are no conditions that must be satisfied before the transformation can be applied.

Another simple transformation that does have a qualifying condition is the following:

$$g \text{ defined},\ \%x \circ g \twoheadrightarrow \%x$$

This means that the transformation holds so long as the application of the function g does not yield $\perp$.

As an illustration of the use of these two transformations, consider the following program:

$$\div \circ (eq0 \circ tail \rightarrow [1,\ \%1 \circ id];\ id)$$

This program is similar to a simple divide, except that it checks to see if the second element in the object sequence is 0, and if it is, sets it to 1. The program can be rewritten by using the two transformations given above.

The program matches the template of the first transformation in the following way:

$h = \div$,
$p = (eq0 \circ tail)$,
$f = [1,\ \%1 \circ id]$, and
$g = id$

Applying this yields

$$eq0 \circ tail \rightarrow (\div \circ [1,\ \%1 \circ id]);\ (\div \circ id)$$

Part of this program matches the template of the second transformation with x = 1 and g = id. Since id is defined for all non-$\perp$ arguments, the transformation can be applied. This produces

```
eq0 ° tail -> (÷ ° [1, %1]); (÷ ° id)
```

Not only can transformations be used that convert one program in a given language into another program in the same language, but transformations can be developed that relate one language to another. An example of such a transformation that converts an ExFP program to an Ada program fragment is the following:

```
p:x = T or p:x = F, (p -> g; h) --»
                    if P then
                         G;
                    else
                         H;
                    end if;
```

where P is an Ada boolean expression corresponding to the ExFP function p, G is an Ada procedure corresponding to the ExFP function g, and H is an Ada procedure corresponding to the ExFP function h.

One final application of program transformations is to convert recursive programs into iterative ones. Certain algorithms are most clearly expressed recursively, but most efficiently executed iteratively. Appropriate use of transformations can be used to convert such algorithms[1].

---

[1] See Alberto Pettorossi, "A Powerful Strategy for Deriving Efficient Programs by Transformation," **1984 Symposium on LISP & Functional Programming**, pp 273-281 for a strategy for doing this.

As this discussion has shown, the basic idea of program transformation is simple; however, at least two major hindrances exist to its practical application. Each of these is discussed briefly below.

The least severe of the two problems is the potential difficulty in verifying that a particular transformation's pre-conditions are satisfied. For the simple transformations given here, the task of pre-condition verification was simple, but for more complicated transformations this is not necessarily the case.

The more severe problem is that for any language of sufficient power, recognizing instances of known templates can be very difficult. For the example above, matching the given program to the template was easy, but consider the following program[1]:

```
eq ° [id, tail ° id] -> loop (1 ° distr) °
eq0 -> or ° [2, 3 ° tail]; not 1 -> /+ ° α%7;
eq0 ° - ° [id, %1] -> loop α*
```

This program does not appear to match either of the templates given above, but it does in the following way:

```
h = eq ° [id, tail ° id] -> loop (1 ° distr),
p = eq0 -> or ° [2, 3 ° tail]; not 1,
f = /+ ° α%7, and
g = eq0 ° - ° [id, %1] -> loop α*
```

Recognition of all instances of known templates is beyond the capabilities of most, if not, all people.

---

[1] This program does not compute anything of known significance. It is given simply as an example of the complexity of template matching.

A proposed solution to both these problems is to develop automated systems to carry out the transformations. At the present time, no system exists that is able to perform this task completely on its own[1]. Systems do exist that are able to perform transformations with assist from a programmer[2]. The future utility of program transformation systems is an open question.

4.4 Variables and Data Types

So far, no mention of the roles of variables and data types in functional languages has been made. In fact, languages, such as ExFP, that strictly follow the Backus' style do not explicitly have either. The only variables are the names of defined function. Not all functional languages are so devoid of variables, but mechanisms for providing variables are directly related to each particular language. For this reason, the subject will be discussed in Section 5.

Mechanisms for providing types in functional languages are less language specific. They fall generally into two categories: implicit type inference, and explicit type declaration. The

---

[1] **Darlington** in **Darlington, et al.**, p 209.

[2] The interested reader should see **Givler, Kieburtz**, pp 74-84, Francoise Bellegarde, "Rewriting Systems on FP expressions that Reduce the Number of Sequences They Yield," **1984 Symposium on LISP & Functional Programming**, pp 63-73, and Phillip Wadler, "Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile Time," **1984 LISP & FP**, pp 45-52. Also of interest is R. Kent Dybvig, Bruce T. Smith, "A Semantic Editor," **85 Symposium on Language Issues in Programming Environments**, pp 74-82, which describes an editor for FP that uses transformations.

approach taken in the discussion below is to briefly describe only one specific method for each category, and to give references to others.

4.4.1 Implicit Typing

In implicitly typed systems, determining the type of a given expression is the responsibility of the compiler or interpreter. One method of doing this is called the reduced computation approach[1]. This technique is described briefly below.

The basic idea of the reduced computation method is to consider a function to be a mapping of one data type to another, possibly identical, data type. In other words, for each function f that produces an object y when applied to an object x, there is a function[2] f' that yields the type of y when applied to the type of x. In order to perform type checking and inference for a given function f, the corresponding reduced computation function f' is considered.

As an example of the operation of this method, consider its application to the language ExFP. Before the technique can be used, the desired types must be chosen. For ExFP, a suitable choice is **number**, **character**, **boolean**, and **sequence**. The first

---

[1] This discussion is from Takuya Katayama, "Type Inference and Type Checking for Functional Programming Languages: A Reduced Computation Approach," **1984 Symposium LISP & Functional Programming**, pp 263-272.

[2] "Function" is used here not in a strict mathematical sense. It is possible that the same types yield different result types. See Ibid, p 264-5.

three types mean what one would expect. The **sequence** type is defined as follows:

if $t_1, \ldots, t_n \in T$ then $\langle t_1, \ldots, t_n \rangle \in T$,
where T is the set of types

Given these types, the type expressions for the primitive functions of ExFP can be stated. To express these clearly the following notation is used[1]:

1. x•y represents the sequence <x, y> so long as y is a sequence; if y is not a sequence, the expression is undefined.

2. xϱy represents the sequence <x, y> so long as x is a sequence; if x is not a sequence, the expression is undefined.

3. x^n represents the sequence containing x n times.

4. *f*′:x --> y means that y is produced when the type domain function *f*′ is applied to x.

Using this notation, the type expressions for the ExFP primitive functions are the following (**r**, **s**, and t are type variables; **n** is an integer variable):

1. id′:t --> t That is, the **id** function yields the same type as its argument.

2. 1′:t•s --> t, 2′:r•(s•t) --> s, ... That is, the **selector function i** produces the type of the i^{th} element in the sequence.

3. tail′:t•s --> s

4. distr′:$\langle t^n, s \rangle$ --> $\langle t, s \rangle^n$

---

[1] The notation for 1. is slightly different from Ibid, p 267. The notation for 2. is entirely new; Katayama provides no means for expressing this. The notation for 3. is the same. The notation for 4. is slightly different.

distl′:<s, t^n> --> <s, t>^{n} [1]

5. appendr′:<t, s> --> t⍮s
   appendl′:<t, s> --> t•s

6. +′:number^2 --> number
   -′:number^2 --> number
   *′:number^2 --> number
   ÷′:number^2 --> number
   This means that +, -, *, and ÷ must have as arguments a sequence with two elements each of type number^2.

7. eq′:t^2 --> boolean

8. not′:boolean --> boolean
   and′:boolean^2 --> boolean
   or′ :boolean^2 --> boolean

This approach can be extended and applied to the combining forms to give a complete type inference and checking system for non-recursive programs. Further extensions yield a system for some recursive programs as well[3].

The reduced computation technique for inferring and checking types is not the only method for these purposes[4]. The discussion of this particular method should be sufficient to provide a basic understanding of the principles involved.

---

[1] This assumes each element in the type sequence t^n is of the same type t for both **distr**, and **distl**. Specifying a type expression for these functions without this assumption is much more complex. See Ibid, p 267.

[2] The fact that ÷ cannot have a second argument equal to 0 is not expressed here.

[3] See Ibid. for how these extensions are made.

[4] The interested reader should see Luis Dumas and Robin Milner, "Principle type-schemes for functional programs," **9th ACM Symposium Principles of Programming Languages**, pp 207-212, and John Mitchell, "Coercion and Type Inference," **11th ACM Symposium Prin. of Prog. Langs.**, pp 175-185.

4.4.2 Explicit Typing

The alternative to inferring types is to require explicit declaration of them. As an example of how this might be done, ExFP is extended to include type definitions[1]. Such an extension requires several steps. Each of these is discussed below.

The first thing that must be done is to determine the types that will be added. For this example, the types **number**, **character**, **boolean**, and **sequence** will be used. These types have meanings as would be expected, and have the abbreviations **num**, **char**, **bool**, and **seq** respectively.

Once the types are decided, primitive functions that determine if an object is of a particular type must be added. Given the four types above, the following four functions are added to ExFP: isNum, isChar, isBool, and isSeq. The program isType:x yields T if x is of type Type and F otherwise.

Another necessary extension to ExFP is the combining form **TypeOK**. This combining form is defined in the following way[2]:

> **TypeOK**: this is denoted by TypeOK(*f*) applied to an object x; the combining form yields T if *f*:x yields an acceptable type; the definition of what constitutes an acceptable type depends on *f*.

As an example of the **TypeOK** combining form, consider the following:

TypeOK(+ ° [1, 2])

---

[1] This discussion is based on John Guttag, "Notes on Using Types and Type Abstraction in Functional Programming," in **Darlington, et al.**, pp 116-126.

[2] This definition is based on Ibid, p 119.

This is equivalent to the following expression:

and°[isSeq, and°[isNum°1, isNum°2]]

In order to provide a suitable mechanism for type declarations, the **Def** facility of ExFP must be extended to allow the inclusion of names for parameters on the left hand side[1]. The general form of this extension is the following:

Def f ° $E(x_1, \ldots, x_n) = F(x_1, \ldots, x_n)$

This form makes the reading of function definitions easier. For example, the following definition under the old form

Def f = g ° id

can be written in the new form as

Def f ° x = g ° x

which more clearly shows the dependance of f on the argument object x.

The new form of **Def** also allows clearer specification of restrictions on the form of objects acceptable to a function. For example, consider a function f that is intended to perform g ° [2, 1] only if the argument object consists of a pair. Using the unextended definition, the function could be written as

Def f = pair -> g ° [2, 1]; $\perp$

where **pair** yields T if the object is a sequence of two elements. With the extended definition, the function could be written as follows:

---

[1] This extension is based on **Backus, LNinCS107**, pp 27-37. The description here is informal. For a formal discussion, see the paper. Backus' use of ° in the notation is confusing, but it is used here for lack of anything better.

```
Def f ° [x, y] = g ° [y, x]
```

which is easier to read, and does not require the introduction of the function **pair**.

Another necessary extension is a way to declare the types of the arguments to a function and the type of the object returned by a function. To allow the former, parameters in a function definition are allowed to have a type specification. A type specification is of the following form[1]:

```
x{Type}
```

As an example, consider the function *f* defined as above. If *f* is desired to yield ⊥ for any x and y that are not both numbers, the definition could be written as

```
Def f ° [x{Num}, y{Num}] = g ° [y, x]
```

This is equivalent to the non-extended definition

```
Def f = and°[and° [isNum°1, isNum°2], %T] ->
          g ° [2, 1]; ⊥
```

To allow the declaration of the type of the object produced by a function *f*, the following notation can be used[2]:

```
Def f ° ... yields Type = ...
```

For example, if *f* as defined above should yield either T or F, the definition could be written as

```
Def f°[x{Num}, y{Num}] yields Bool = g°[y, x]
```

---

[1] This notation differs from **Guttag** in **Darlington, et al.** He uses the form x:Type. The use of : here, although consistent with traditional languages, seems inappropriate since the symbol also stands for application.

[2] Ibid, p 118 uses **returns** instead of **yields**.

Given all of these extensions, all that would be necessary to complete the introduction of types to ExFP would be the redefinition of the primitive functions to include type specifications. As an example, the functions **or**, **tail**, and **+** would be of the following forms respectively[1]:

```
or ° [x{Bool}, y{Bool}] yields Bool

tail ° [x{Seq}] yields Seq

+ ° [x{Num}, y{Num}] yields Num
```

Some functions can take arguments of several types and return several different types. Each possibility can be defined separately. For example, **id** can take either a sequence, a number, or a character. The function could be defined in the following form[2]:

```
id ° [x{Seq}] yields Seq
id ° [x{Num}] yields Num
id ° [x{Char}] yields Char
```

The above discussion has shown a technique for adding type declarations to the language ExFP. Other methods exist as well. Section 5 discusses the type schemes of each language mentioned[3].

4.5 Lazy Evaluation

Lazy evaluation is a particular technique for determining when expressions are evaluated. It is used by many functional

---

[1] These are given for example purposes only, not as rigorous definitions.

[2] Again, this is given only as an example.

[3] That is, where such information is available in the literature. See also, D. B. MacQueen and Raui Sethi, "A Semantic Model of Types for Applicative Languages," **1982 Symposium on LISP & Functional Programming**, pp 243-252.

languages. The basic principle of lazy evaluation is that an evaluation is done at the time that the result is needed, and not before. In contrast, in a conventional (or busy) evaluation scheme, all computations are done as soon as possible.

As an example of the method, consider the following ExFP program and application:

```
or ° [%T, eq0 ° ÷]: <7, 0>
```

A conventional evaluation scheme would give the following:

```
or: [%T, eq0 ° ÷]: <7, 0>
or: <%T: <7, 0>, eq0 ° ÷: <7,0>>
or: <T, eq0: ÷: <7, 0>>
or: <T, eq0: ⊥>
or: <T, ⊥>
⊥
```

However, a lazy evaluation scheme would give something like the following:

```
or: [%T, eq0 ° ÷]: <7, 0>
or: <%T: <7, 0>, eq0 ° ÷: <7, 0>>
or: <T, eq0: ÷: <7, 0>>
T
```

One consequence of such a scheme is that a programmer has little control over the order of execution of operations[1]. He must not assume anything about evaluation order. Making such assumptions is generally considered to be a bad practice; so lazy evaluation has the advantage of discouraging it.

This discussion of lazy evaluation should be sufficient to show the basic idea. Much research has been conducted on its

---

[1] Cordelia Hall and John T. O'Donnell, "Debugging in a Side Effect Free Programming Environment," **1985 Symposium on Language Issues in Programming Environments**, p 61.

use, so the interested reader may consult a variety of references for more information[1].

4.6 Implementation on Conventional Architectures

The conventional computer architecture consists of three parts: a processor, memory, and a communications line connecting the processor and memory[2]. Most computers, from small personal computers to large main frames, employ this same basic architecture with only minor differences. In his Turing Award paper, John Backus argues that a major cause of the inadequacies of traditional programming languages is their dependance on this conventional computer model. He claims that new architectures are essential to the improvement of languages[3].

Backus' arguments may well be true; however, the vast majority of computers will almost certainly retain the traditional architecture throughout the foreseeable future. For this reason, the acceptance of functional languages is partially

---

[1] To start, see Peter Henderson and J. Morris, Jr., "A Lazy Evaluator," **3rd Symposiums on Principles of Programming Languages**, pp 95-103, P. A. Subrahmanyam and J. H. You, "Pattern Driven Lazy Reduction: A Unifying Evaluation Mechanism for Functional and Logic Programs," **11th Symposium on Principles of Programming Languages**, pp 228-234, and **Glaser, et al.**, pp 70,71. For a discussion of lazy evaluation for Backus' FP, see Walter Dosch and Bernard Moller, "Busy and Lazy FP with Infinite Objects," **1984 Symposium on LISP & Functional Programming**, pp 282-292.

[2] Philip C. Treleaven, "Computer Architecture For Functional Programming," **Darlington, et al.**, p 290. Such an architecture is often called a "von Neumann" architecture, after one of the men who conceived it.

[3] **Backus, CACM**, p 615.

dependent on whether or not they can be implemented efficiently on conventional machines.

A complete discussion of the topic would be much more complicated than is desirable for this paper. However, basic implementation techniques can be discussed briefly. Most research has concentrated on implementing lambda based languages, so the discussion also concentrates on methods for such languages. Since, as was mentioned in Section 2.3, the lambda calculus can be used to simulate any function or functional form, the lambda methods can be adopted for use for FP style languages.

4.5.1 Translation to Another Language

Perhaps the simplest way to implement a functional language is to translate it into another high-level language. A translator system takes source code from the functional language and produces code in another language for which a compiler or interpreter already exists.

The Berkeley FP system uses this technique[1]. The language of the system is based on the FP language introduced by Backus. Programs written in this language are translated into LISP code. This code may then be either interpreted or compiled. As one might expect, the system does not execute programs with much speed.

Generally, any implementation based on translation to another high-level language will not be very efficient. For this

---

[1] See **Baden**, and Scott E. Baden, Dorab R. Patel, "Berkeley FP -- Experiences with a Functional Programming Language," **Digest of Papers of CompCon 83**, 1983, pp 274-77.

reason, such implementations are not suitable for most practical programming applications.

4.6.2 SECD Machine

The SECD machine is the standard way to implement functional languages that are based on the lambda calculus[1]. It is intended to be created in software on conventional architectures. A SECD machine operates according to the following algorithm:

```
WHILE (an expression is left to be evaluated)
     OR (there is a suspended computation) DO
     IF the current evaluation is done THEN
          resume the last suspended evaluation
     ELSE
          CASE next expression to be evaluated OF
               identifier:push the value onto the evaluation
               stack and pop the next expression from the
               expression stack
               Lambda-exp: push the appropriate closure onto
               the evaluation stack and pop the next
               expression
               application: replace the top of the
               expression stack by the expression
               representing this appl.
               "ap": cause the operator on the evaluation
               stack to be applied to the operand below it.
          END CASE
     END IF
END WHILE
```

As given, the SECD model is not able to support lazy evaluation; however, the machine can be extended to provide this support. Characterizing the efficiency of SECD based implementations is difficult. In general, they are not thought

---

[1] **Glaser, et al.**, p 82. The algorithm is from p 84, with some changes in notation.

to be adequately efficient to allow practical realizations of languages[1].

4.6.3 Combinator Systems

Another method of implementing functional languages is the use of combinators. Combinators are a method for representing lambda calculus expressions in a shorter form. As an example, the single combinator S is used to represent the following lambda expression[2]:

Lam x Lam y Lam z. (x z) (y z)

Combinator systems can be used to implement lambda based languages. The first part of such systems is a translator that converts the lambda expressions into their combinator representation. Many different techniques exist for producing executable code from these representations. Most methods are able to support lazy evaluation.

Studies have shown that these techniques are generally as efficient as direct lambda expression implementations such as the SECD Machine discussed above[3]. Some researchers claim that combinator implementations are much more efficient, enough so to

---

[1] Paul Hudak and David Kranz, "A Combinator-Based Compiler for a Functional Language," **11th Symposium on Principles of Programming Languages**, p 122.

[2] R. J. M. Hughes, "Super Combinators: A New Implementation Method for Applicative Languages," **1982 Symposium on LISP & Functional Programming**, p 2.

[3] Simon L. Peyton Jones, "An Investigation of the Relative Efficiencies of Combinators and Lambda Expressions," **1982 Symposium on LISP & Functional Programming**, pp 150-158.

allow practical realizations of functional languages[1]; however, other researchers dispute such claims.

4.6.4 Stack Based Systems

Some research has suggested that certain lambda based functional languages can be implemented using a run-time stack. This method involves the modification of the conventional run-time stack used for languages like Pascal. These modifications include adding separate stacks for return addresses and intermediate variables, and adding a new pointer to each activation record.

This method is said to be applicable to any functional language that uses copy-rule parameter passing. It is also claimed to be suitably efficient[2].

4.6.5 Remarks

The above discussion has been necessarily brief. Not all implementation methods for conventional machines have been

---

[1] **Hudak, Kranz,** pp 122-132. For more information on combinator implementations, see Paul Hudak and Benjamin Goldberg, "Experiments in Diffused Combinator Reduction," **1984 Symposium on LISP & Functional Programming**, pp 167-176, Steve S. Muchnick and Neil D. Jones, "A Fixed-Program Machine for Combinator Expression Evaluation," **1982 Symposium on LISP & Functional Programming**, pp 11-20, and **Glaser, et al.**, pp 93-104.

[2] See U. Honschapp, W.-M. Lippe, and F. Simon, "Compiling Functional Languages for von-Neumann Machines," **1983 Symposium on Programming Language Issues in Software Systems**, pp 22-27, and M. P. Georgeff, "A Scheme for Implementing Functional Values on a Stack Machine," **1982 Symposium on LISP & Functional Programming**, pp 188-195.

discussed[1]. Those techniques that have been mentioned have been covered in a cursory manner. The discussion has shown that the current techniques for implementing functional languages on conventional architectures do not appear to produce adequate efficiency. The development of new architectures shows more promise; this research is discussed in the next section.

4.7. New Architectures

Since implementations of functional languages on conventional architectures thus far have been inefficient, much research is being done on developing architectures to support functional languages specifically. Three approaches to such architectural innovation are discussed below[2].

4.7.1 Data Flow Machines

The ideas of data flow architectures have been in existence for a number of years. The original motivation for these type machines was not functional languages[3]; however, data flow machines do have properties which make them appear attractive for functional language implementations. One particular data flow machine being developed at the Massachusetts Institute of Technology is discussed below[4].

---

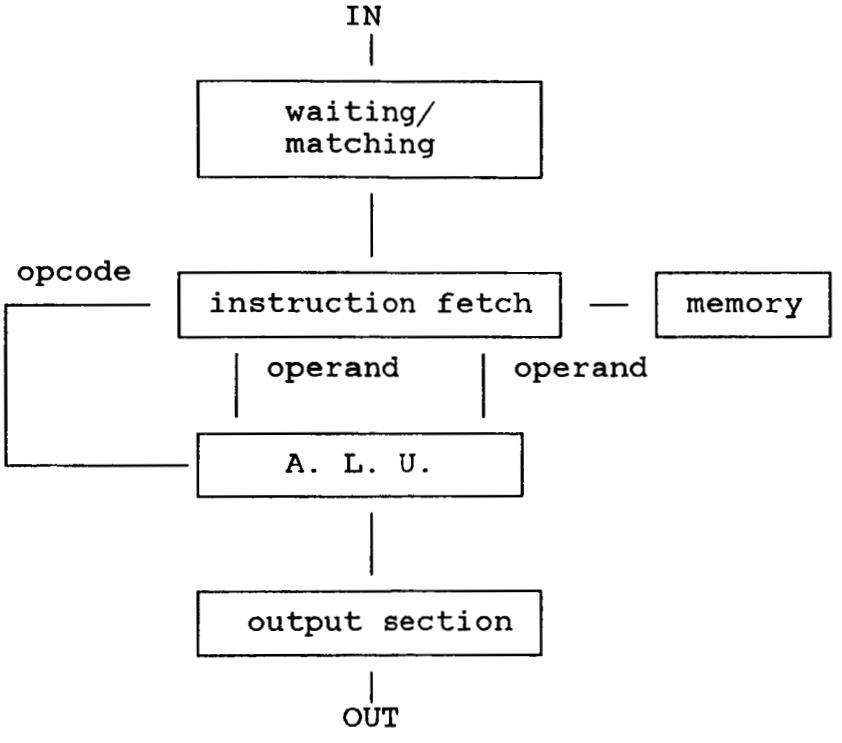
[1] For one interesting additional technique, see Corrado Böhm, "Combinatory Foundation of Functional Programming," **1982 Symposium on LISP & Functional Programming**, pp 29-36.

[2] The interested reader should see **Proceedings of the Conference on Functional Programming Languages and Computer Architecture**, ACM, October, 1981.

[3] **Glaser, et al.**, p 104.

[4] This discussion is based on **Backus, IEEE**, pp 26,27.

The M.I.T. machine consists of a large number of identical processors. These processors are connected by a network that allows each processor to communicate with any other processor in the network. The communication is done by passing packets. Each packet contains the address of the processor for which it is intended, a list of other packets with which the packet's data is to be combined, and data. The structure of an individual processor is shown below[1].



A processor operates in the following manner:

1. A packet arrives as input. It waits in the waiting/matching unit until all packets from which it needs data have arrived.

2. When a complete match is made, the packets are sent to the instruction fetch unit.

---

[1] The figure is from Ibid, p 26.

3. This unit gets the necessary instructions from memory and sends them and the data to the arithmetic and logic unit.

4. The A.L.U. uses the instructions to form a result from the data. This result is sent to the output section.

5. The output section puts the result data in a new packet. The address(es) to which this packet is to be sent is computed from the input packets and the instructions.

6. The packet is sent out onto the network.

The developers of this machine believe that it can be used to implement any type of functional language. At the time of the writing of the reference on which the discussion is based, a 64 processor prototype was expected to be in operation by the end of 1985.

4.7.2 Professor Gyula Mago's Machine

Unlike the previous machine, the architecture being developed by Gyula Mago at the University of North Carolina is intended to be used only for the implementation of functional languages based on Backus' style[1]. This machine has a cellular design. That is, it consists of a large number of interconnected components. These components, called cells, are one of two types: leaf or tree. The tree cells are connected as a binary tree. The leaf cells are connected at the base of the tree. Each leaf cell is also connected to its two neighboring leaf cells.

---

[1] This discussion is based on Ibid, pp 25,26, and Gyula Mago, "Data Sharing in an FFP Machine," **1982 Symposium on LISP & Functional Programming**, pp 201-202.

The structure of the cells is very simple. Each leaf cell consists of a small processor, a small microcode memory, memory for a symbol and its position in the program, and some condition registers. The purpose of these cells is twofold: to store data, and to process data. FP primitive operations and combining forms are implemented directly in the cells' microcode.

Each tree cell is made up of data registers and a simple processor capable of moving data and doing simple operations. The purpose of these cells is to control communication between leaf cells.

To evaluate an FP expression, the expression and the data to which it is applied are mapped onto the leaf cells. Each leaf cell contains zero or one symbol from the program. The machine then proceeds with an execution cycle of three phases[1].

In the first phase, the tree cells partition the representation into each independent subexpression. Each of these consists of a single function and the data to which it is applied. All subexpressions so partitioned can be evaluated in parallel. The process of partitioning configures the machine to match the program and data. This is in contrast to most other approaches which try to match the program and data to the hardware.

In the second phase, each independent sub-tree attempts to evaluate its first application. If this is possible, the function and data to which it is applied are replaced by the

[1] The discussion of the execution cycle is from **Mago**, p 202.

result of the application. Most FP functions can be executed immediately in one cycle.

The final phase of a machine cycle is storage management. The remaining subexpressions and data are moved to the appropriate places among the leaf cells. At the conclusion of this phase, another execution cycle begins. This continues until a result is obtained.

In order to evaluate large FP programs effectively using this architecture, a machine must contain very many cells. The simplicity of an individual cell suggests that providing as many as a million cells at reasonable cost may be feasible using Very Large Scale Integration methods.

4.7.3 SKIM II Processor

The final new architecture that will be mentioned is the SKIM II processor being developed at the University of Cambridge. The discussion of this design will be brief because the available literature says very little about the architecture of the machine[1].

The SKIM II processor is the successor of the SKIM I processor. It is intended for the efficient implementation of combinator methods for functional language evaluation. The processor has separate memory for programs and data. It is

---

[1] The discussion is based on W. R. Stoye, T. J. W. Clarke, and A. C. Norman, "Some Practical Methods for Rapid Combinator Reduction," **1984 Symposium on LISP & Functional Programming**, pp 159-166.

controlled by microcode. This microcode contains the combinator reduction algorithm.

The implementation of combinator evaluation on SKIM II is claimed to be significantly more efficient than the implementation of the method on conventional machines. Improvements in combinator techniques for those machines can be easily incorporated into the processor, so the difference in efficiency should remain regardless of advancements in the algorithms.

In addition to comparing the efficiency of SKIM II to conventional architectures, the developers have compared the efficiency to implementations of non-functional languages. The results suggest the performance of SKIM II is about one-quarter of that of a traditional language compiled on comparable cost hardware[1].

---

[1] Ibid, p 166.

5. EXISTING FUNCTIONAL LANGUAGES

In the previous sections, the only specific functional language discussed has been the example language ExFP. In this section, several actual languages that are claimed to be functional are discussed. The languages are divided into two categories: languages based on Backus' style; and languages not based on Backus' style (these may be based on the lambda style, based on a combination of the two styles, or based on neither style).

5.1 FP Based Languages

The theory of FP based languages is still in its infancy. Most of the research has been in developing the theory, not on designing specific languages. Thus, not many FP based languages are discussed in the available literature. Three language systems that do exist are described below.

5.1.1 Berkeley FP

The Berkeley FP system[1] was mentioned briefly in Section 4.6.1. The language implemented is very similar to the specific FP language described by Backus, and thus is similar to ExFP as well. The differences between Berkeley FP and Backus' language are primarily syntactical. Also, the Berkeley language provides a greater number of primitive functions. The system was designed for experimentation, not for practical programming.

---

[1] See **Baden** and **Baden, Patel**.

5.1.2 μFP

The language μFP[1] is used to describe V.L.S.I. circuit design. It is a variant of Backus' FP. The differences between μFP and Backus' FP are discussed below.

One primary difference between the languages is in the nature of the data accepted and produced. FP functions operate on one input and produce one output. μFP functions operate on a sequence of inputs that varies over time and produce a time varying sequence of outputs.

As illustration, consider the function +. In FP, this function can be applied only to a single pair. For example,

+: <7, 8> = 15

In μFP, the function can be applied to a sequence of pairs, where each pair represents input at a certain time. For example,

+: <<7, 8>, <10, 9>, <1, 2>, <3, 4>, ...> =
<15, 19, 3, 7, ...>

The consequence of this difference in input and output values is that a μFP function *f* is basically equivalent to the FP combining form αf. The only non-equivalence to this relationship is that the μFP function may take a potentially infinite sequence as input, while the FP function's input sequence must be finite[2].

---

[1] Mary Sheeran, "muFP, a language for VLSI design," **1984 Symposium LISP & Functional Programming**, pp 104-112.

[2] That is, for FP as defined by Backus. For a discussion of an extension that allows infinite input sequences, see Tetsuo Ida and Jiro Tanaka, "Functional Programming with Streams," **Information Processing '83: Proceedings of the IFIP Ninth World Computer Congress**, Sept 19-23, 1983, pp 265-270.

The relationship between a μFP expression and its equivalent FP expression can be express using a meaning function, denoted by M[1]. $M\{f\}$ is equal to the FP equivalent of the μFP function f. Thus, the relationship discussed above can be expressed in the following way:

$$M\{f\} = \alpha f$$

In order to express the relationship between the combining forms, the function tran is needed. This is the matrix transpose function. It is used to express the conversion of a stream of sequences to a sequence of streams. The meanings of μFP combining forms is given below:

constant:
$M\{\%y\} = \alpha\%y$

composition:
$M\{f \circ g\} = M\{f\} \circ M\{g\}$

construction:
$M\{[f_1, \ldots, f_n]\} = tran \circ [M\{f_1\}, \ldots, M\{f_n\}]$

apply to all:
$M\{\alpha f\} = tran \circ \alpha M\{f\} \circ tran$

insert:
$M\{/f\} = /(M\{f\} \circ tran) \circ tran$

condition:
$M\{p \rightarrow g; h\} = \alpha(1 \rightarrow 2;3) \circ tran \circ [M\{p\}, M\{g\}, M\{h\}]$

loop:
μFP does not provide this combining form

The other primary difference between μFP and FP is that μFP contains the additional combining form μ. This form introduces a limited memory to the language. The expression μf means that the

---

[1] The notation and definitions are from **Sheeran**, p 105, 106.

next output and next state depend on the current input and current state. In a given cycle, the current state is the second input. Initially, the state is considered to be the unknown value ?.

To illustrate the use of the μ combining form, consider a program to represent a shift register cell[1]. This cell has as output its current state, and as input its new state. The μFP function for this is the following:

μ[2, 1]

Given the input <0, 1, 0, 0, 1, 0, ... >, the output of the program would be <?, 0, 1, 0, 0, 1, 0, ... >.

The introduction of the new combining form does not significantly alter the algebra of programs. Most of the laws and theorems that hold for FP, hold for μFP.

An interpreter for μFP has been written. Also, the language has been combined with a functional geometry system[2] to produce pictures of a design layout.

5.1.3 F Shell

The F shell[3] is not a programming language. Rather, it is a command interpreter in the spirit of the C shell and Bourne shell

---

[1] Ibid, p 107.

[2] Peter Henderson, "Functional Geometry," **1982 Symposium on LISP & Functional Programming**, pp 179-187.

[3] Jon Shultis, "A Functional Shell," **1983 Symposium on Programming Language Issues in Software Systems**, pp 202-211.

of UNIX[1]. It is, however, based on Backus' FP. For that reason, it is discussed below.

The data of F shell programs consist of finite sequences of characters called data streams. These streams may be nameless or labelled with integer values. Also, several separate streams may be combined into one stream with several components; such a stream is called a structured stream.

The shell provides a set of primitive programs. Each primitive program takes a data stream as input and produces a data stream as output.

The F shell has four types of PFOs: composers, structurers, selectors, and powers. There are three types of composers. The first of these is called **composition** and is similar to FP's **composition**. The difference is that the F shell composition is evaluated left to right. That is, the F shell expression $f \circ g$ is analogous to the ExFP expression
$g \circ f$.

The second composer is called **source**. The notation p<f means that a data stream is created from file f and sent as input to program p. The third composer is called **sink**. It is denoted by >f, and means that a data stream is sent to the file f. Thus, to copy file f_1 into file f_2 in the F shell, one writes $f_1 <> f_2$.

The first structurer PFO is called **construction**. The expression $[p_1, \ldots, p_n]$ creates a structured data stream, whose j^{th} component is the result of applying p_j to the input stream.

---

[1] UNIX is a trademark of AT & T Bell Laboratories.

The second structurer is **product**[1]. The expression p_1 * ... * p_n takes a structured stream with n components as input. It produces a structured stream with n components, where the j^{th} component is the result of applying p_j to the j^{th} component of the input. The third structurer is **projection**. This is denoted by 1^{st}, 2^{nd}, ..., where the j^{th} projection selects the j^{th} component of a structured stream as output.

Three selectors exist: **alternation**, **sum**, and **labelling**. **Alternation** is denoted by $\leq p_1, \ldots, p_n \geq$. This expression takes a labelled stream as input and produces as output the result of applying p_j to it, where j is the label. The **sum** PFO is identical to **alternation** except that it retains the label on the stream. It is denoted by $p_1 + \ldots + p_n$. The PFO **label** is denoted by ^1, where 1 can be any program that produces an integer as output.

The final type of PFO in the F shell is that of powers. Any of the three operators °, *, and + can be used in a power. The general form is $p^{\beta n}$, where ß is one of the three operators and n is an integer. The expression is equivalent to the program p ß p ... ß p, where there are n p's.

As an example of the use of these PFOs, consider a program that is intended to do the following:

> apply p to the input; then apply program b, if b produces 1, apply p_1; if b produces 2, apply p_2.

---

[1] The notation used here for **product**, **alteration**, and **label** are different than Ibid. His notation uses symbols not available to this author.

This program can be written in the F shell as follows:

$$p \circ \hat{}b \circ \langle p_1, p_2 \rangle$$

As mentioned above, the F shell obeys algebraic laws. The reader may enjoy attempting to verify the validity of the following representative laws[1]:

$$f \circ (g \circ h) \equiv (f \circ g) \circ h$$

$$[f_1, \ldots, f_n] \circ j^{th} \equiv f_j \quad \text{if } 1 \leq j \leq n$$

$$\hat{}k \circ \langle g_1, \ldots, g_n \rangle \equiv g_k$$

$$f_1 + \ldots + f_n \equiv \langle f_1 \circ \hat{}1, \ldots, f_n \circ \hat{}n \rangle$$

A major problem with using the F shell as a practical command interpreter is that most keyboards are unable to produce many of the characters used by the shell[2]. The notation can be converted to standard characters, but this reduces its readability. A prototype version of the shell has been implemented in this way. Also, research is being conducted into developing a graphical representation for personal computers.

5.2 Non-FP Style Languages

Because the emphasis in this paper has been on FP style languages, languages based on other styles, such as the lambda calculus, will be discussed only briefly. The interested reader can consult the references mentioned for each language for more information.

5.2.1 LISP

---

[1] **Shultis**, pp 210-211.

[2] Clearly, this is a problem with all FP languages.

LISP is one of the oldest languages still in use. Extensive literature on the language has been published, so nothing more will be said about it here[1].

5.2.2 KRC

The three basic properties of KRC are equational definitions, pattern matching, and set abstraction[2]. As an example of equational definitions, the Fibonacci function can be defined in KRC in the following manner:

```
fib n = 1, n = 1
      = 1, n = 2
      = fib(n - 1) + fib(n - 2), n > 2
```

As an example of the pattern matching facility, consider the definition of a function that adds a list of integers. This can be defined in KRC as follows:

```
total [ ] = 0
total (a : x) = a + total x
```

[ ] denotes an empty list. "a : x" matches any non-empty list; "a" denotes the first element of the list, and "x" denotes the remainder of the list. The operator ":" denotes construction of a list from its two operands.

The set abstraction facility of KRC allows sets to be expressed in much the same way as done in mathematics. As an

---

[1] For a simple introduction to the language, see Terrence W. Pratt, **Programming Languages: Design and Implementation**, 2nd edition (Prentice-Hall: Englewood Cliffs, New Jersey, 1984), pp 497-527. The original published discussion of the language was in John McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine Part I," **CACM**, Vol 3, No 4, April 1960, pp 184-195.

[2] This discussion is from **Glaser, et al.**, pp 180-184.

example, the set [[2, 1], [3, 1], [3, 2]] can be expressed in KRC as the following:

```
{ [x, y] | x, y <- [1 .. 3]; x > y}
```

This definition reads "the set of pairs x and y such that x and y are in the range 1 to 3 and x is greater than y."

That KRC is not based on the FP style of language can be seen in the lack of program forming operations. Problems are solved by defining new functions and using them in a manner similar to traditional languages.

5.2.3 HOPE

HOPE uses equational definitions and pattern matching in much the same way as KRC[1]. The major differences between the two languages is that HOPE also provides a typing mechanism and a data abstraction facility.

The typing mechanism of HOPE is partially implicit and partially explicit. The types of functions must be declared; all other types are inferred. As an example, the factorial function might be defined in HOPE in the following way:

```
dec factorial : num -> num;
--- factorial(n) <= n * factorial (n - 1)
--- factorial(0) <= 1;
```

This definition means that **factorial** accepts a single parameter of type **num** and produces a single result of type **num**.

In addition to built in types, HOPE provides a facility for users to define their own types. Also, the language provides

---

[1] Ibid, pp 185-195.

data abstraction by allowing the operations on defined types to be restricted to ones specified in the type definition.

As an example of the use of these facilities, consider the following data abstraction of the familiar stack[1]:

```
module stacks
typevar alpha
pubtype stack (alpha)
pubconst pop, top, empty, push
data stack(alpha) == empty ++ push(alpha, stack(alpha))
dec pop : stack(alpha) -> stack(alpha)
dec top : stack(alpha) -> alpha
--- pop(push(a,b)) <= b
--- pop(empty) <= empty
--- top(push(a,b)) <= a
--- top(empty) <= error
end
```

A programmer can use this module to declare a stack of any type. This stack can be accessed only through the functions pop, top, empty, and push.

As with KRC, HOPE provides no powerful program forming operations. It is a functional language, but not in the FP style.

5.2.4 APL

APL was designed by Kenneth Iverson in the early 1960's[2]. Although it was originally intended as a way to look at programming, not as a specific language, implementations of it were developed. Today, the language has something of a cult following among some programmers.

---

[1] Ibid, p 190.

[2] Kenneth E. Iverson, **A Programming Language** (John Wiley and Sons: New York, 1962).

There are some similarities between APL and the functional approach being discussed in this paper, but it is not truly a functional language. Several of the reasons why this is true are mentioned below[1].

First, the language maintains the distinction between expressions and statements. Expressions are governed by algebraic properties, but statements are not. A large amount of programming must be done using statements.

Second, APL has only three functional forms. This is not really sufficient for full functional programming. Also, the use of these forms is restricted.

These two facts mean that APL, although a step in the functional direction, is more related to traditional languages than to functional ones[2].

5.2.5 Others

Other functional languages besides those mentioned above exist. These include the following:

ML: This language was initially designed as a metalanguage for proofs. It has evolved into a general purpose language. ML provides strong typing through implicit type checking. It is being developed at the same university as HOPE, and shares many of the same attributes[3].

---

[1] These reasons are given by Backus in **Backus, CACM**, p 618.

[2] For a discussion of the type scheme of APL, see W. E. Gull and M. A. Jenkins, "Decisions for ´type´ in APL," **6th Symposium on Principles of Programming Languages**, pp 190-196.

[3] For more information on ML, see Lennart Augustsson, "A Compiler for Lazy ML," pp 218-227, David MacQueen, "Modules for Standard ML," pp 198-207, and Robin Milner, "A Proposal for Standard ML," pp 184-197, all in **1984 Symposium on LISP &**

Daisy: This language is similar to LISP. It is based on the lambda style. It allows variables, but does not have type checking[1].

Poplar: This is an experimental language, designed for use in text and list processing. It has properties in common with LISP, with the addition of string matching facilities[2].

Artic: This language is intended to be used for the implementation of real time control systems. Real time constraints are expressed as time valued functions[3].

---

[1] For more information, see **Hall, O'Donnell** and John T. O'Donnell, "Dialogues: A Basis for Constructing Programming Environments," **1985 Symposium on Language Issues in Programming Environments**, pp 19-27.

[2] For more information, see James H. Morris, "Real Programming in Functional Languages" in **Darlington, et al**, pp132-153.

[3] For more information, see Roger B. Dannenburg, "Artic: A Functional Language for Real-Time Control," **1984 Symposium on LISP & Functional Programming**, pp 96-103.

6. CONCLUDING REMARKS

This paper has presented a survey of the ideas of functional programming languages. Not all the possible topics have been covered[1], but enough information has been given to provide a suitable background for advanced study. The paper concludes with an assessment of the future of functional programming languages.

In assessing the potential of functional languages, several questions need to be answered. These include the following:

1. Does the functional language style actually offer the advantages claimed for it?
2. Can practical languages based on the style be developed?
3. If so, will such languages be accepted?

The author's opinions answers to these three questions follow.

Does the functional language style offer the advantages claimed for it?

Section 3 of this paper mentioned seven advantages that functional languages have been claimed to have over traditional languages. These were the following:

1. functional language programs are easier to understand than non-functional ones
2. functional languages allow the building of new programs from existing ones in a hierarchical fashion
3. functional languages encourage the development of general programs

---

[1] Two particular topics not discussed were mechanisms for allowing user defined combining forms, and methods of adding history sensitivity. The first issue is more complex than thought suitable for this paper; little published research has been done on the second. The interested reader should see **Backus, CACM**, and John H. Williams, "Formal Representations for Recursively Defined Functional Programs," **LNinCS107**, pp 460-470.

4. functional languages allow the expression of inherent parallelism

5. functional languages operate in the same domain as programming problems

6. functional languages possess algebraic properties that allow proofs of correctness to be conducted in the language itself

7. functional languages encourage architectural innovation and are better suited to take advantage of such innovation than non-functional ones.

In the author's opinion, these advantages are provided by the functional language style discussed in this paper. The hierarchical nature of programs, the generality of programs, the ability to express parallelism, the operation in the problem domain, the existence of algebraic properties, and the encouragement of architectural innovation have been shown, at least in part, in previous sections. Whether or not functional programs are easier to understand than traditional language ones is almost entirely a matter of opinion. The author believes that once the combining forms and primitive functions are understood, functional programs are easy to read.

Can practical languages based on the style be developed?

Although the functional language style of Backus may offer significant advantages in theory, the idea is not very useful unless practical languages based on it can be developed. Whether or not production quality functional programming languages can be created depends on several factors. In particular, two such factors are the success of architectural improvements, and the development of suitable methods for incorporating knowledge of

the past into the languages. Architectural improvements appear to be necessary to allow adequately efficient implementations of functional languages. A knowledge of past results is certainly necessary for applications such as database systems, payroll calculations, and word processors.

The time is perhaps too soon to tell if these necessary advances will occur. The architectural research discussed in Section 4.7 looks promising, but its success is by no means assured. The research into history sensitivity appears less promising. In fact, the topic was not covered in this paper because of the lack of available information. A primary problem is to introduce history sensitivity without destroying algebraic properties and simplicity.

Will such languages be accepted?

Even if practical languages are developed, their success depends on their acceptance by the programming community. History suggests that the probability of such acceptance is low.

In 1975, Peter Naur offered the following opinion concerning the future of programming language development:

> ... the split between the more academic, pure computer science oriented study of programming languages and the world of practical programming will persist indefinitely; the era of influential programming language construction is past, FORTRAN and COBOL will retain their dominance....[1]

The ten years since that writing have done little to prove this prophecy wrong. The "world of practical programming" is

---

[1] Peter Naur, "Programming Languages, Natural Languages, and Mathematics," **2nd Symposium on Principles of Programming Languages**, pp 137-148.

extremely resistent to change. To believe that something as radically different as functional languages will be accepted by this world, is to ignore the last twenty years of history.

Lest this paper end on such a discouraging note, let it be said that history does not always accurately predict the future. A large part of the resistance to change may be caused by the lack of solid evidence to show that a new language or technique is significantly better than existing languages and techniques. If functional languages can be developed fully, and if the advantages of such languages can be adequately demonstrated to the programming community, then, perhaps, they will be accepted.

7. ANNOTATED BIBLIOGRAPHY

Augustsson, Lennart, "A Compiler for Lazy ML" in **Conference Record of the 1984 ACM Symposium on LISP and Functional Programming** (1984 ACM Sym LISP & FP), Austin, Texas, August 6-8, 1984, pp 218-227.

This paper describes a compiler for the functional language Lazy ML. Several benchmark tests are presented.

Baden, Scott E., "Berkeley FP User's Manual, Rev. 4.1" in **UNIX Programmer's Manual: Supplementary Documents**, 1980, Chapter 4, 33 pages.

This is the user's manual for the University of California at Berkeley's FP system. Included is a discussion of the differences between Backus' FP and Berkeley FP.

Baden, Scott E. and Dorab R. Patel, "Berkeley FP -- Experiences with a Functional Programming Language" in **Digest of Papers of CompCon 83**, San Francisco, California, Feb 28 - Mar 3, 1983, pp 274-77.

This paper briefly describes the Berkeley FP system. As in the User's Manual, the differences between Berkeley FP and Backus' FP are discussed.

Backus, John, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs" in **Communications of the ACM** (CACM), Vol 21, No 8, Aug 1978, pp 613-641.

This is Backus' Turing Award Lecture that brought the idea of functional programming languages into prominence. In it, Backus discusses the problems with conventional languages and presents functional languages as a potential solution.

Backus, John, "Function-level Computing" in **IEEE Spectrum**, Vol 19, No 8, Aug 1982, pp 22-27.

This paper briefly covers topics similar to the Turing Award Lecture. It also discusses research in developing computer architectures specifically designed to support functional languages.

Backus, John, "Is computer science based on the wrong fundamental concept of `program´?" in **Algorithmic Languages: Proceedings of the International Symposium on Algorithmic Languages**, North-Holland Publishing Company, Amsterdam, 1981, pp 133-165.

This paper is similar to the Turing Award paper in topics. The approach is different: discussing the notion of `program´ rather than `language´.

Backus, John, "Programming Language Semantics and Closed Applicative Languages" in **Conference Record of the ACM Symposium on Principles of Programming Languages**, Boston, Mass. Oct 1-3, 1973, pp 71-86.

This paper describes some of Backus´ early work in applicative languages. This work eventually led to the ideas of the Turing Award paper.

Backus, John, "The algebra of functional programs: Function level reasoning, linear equations and extended definitions" in **Formalization of Programming Concepts**, Lecture Notes in Computer Science Volume 107 (LN in CS Vol 107), Springer--Verlag, Berlin, 1981, pp 1-43.

The paper discusses the relationship between the FP style and the lambda style of programming. Backus also extends the discussion of the algebra of FP, and proposes a revised form of function definition intended to be clearer than the form given in the Turing Award Lecture.

Bellegarde, Francoise, "Rewriting Systems on FP expressions that Reduce the Number of Sequences They Yield" in **1984 ACM Sym LISP & FP**, pp 63-73.

This paper describes a program transformation system that minimizes intermediate lists. The system produces a set of rules that give a normal form for each term in a program. The system is not claimed to be optimal.

Berkling, K. J. and E. Fehr, "A modification of the Lambda-Calculus as a base for functional programming languages" in **Automata, Languages and Programming**, Lecture Notes in Computer Science Volume 140, Springer-Verlag, Berlin, 1982, pp 35-47.

This paper describes the Berkling Reduction Language, a language based on a modification of the lambda-calculus.

Böhm, Corrado, "Combinatory Foundation of Functional Programming" in **Conference Record of the 1982 ACM Symposium on LISP and Functional Programming** (1982 ACM Sym LISP & FP), Pittsburgh, PA, Aug 15-18, 1982, pp 29-36.

This paper proposes a way to embed Backus' FP into a combinatory logic system. This is intended to aid in implementing FP on conventional machines. It also is claimed to reduce the number of necessary primitives.

Broy, Manfred, "Applicative Real-Time Programming" in **Information Processing 83: Proceedings of the IFIP Ninth World Computer Congress** (IP 83), Paris, France, Sep 19-23, pp 259-264.

An applicative programming language with time-related functions is described in this paper. The language is stated to be too restrictive to be suitably implemented.

Burge, W. H., **Recursive Programming Techniques**, Addison-Wesley Publishing Company, Reading, Mass., 1975.

This book describes a method of programming using a language based on the lambda calculus.

Cardelli, Luca, "Compiling a Functional Language" in **1984 ACM Sym LISP & FP**, pp 208-217.

This paper summarizes the author's experiences in developing a compiler for the language ML. The implementation was different than that for any traditional compiler.

Cartwright, Robert and James Donahue, "The Semantics of Lazy (and Industrious) Evaluation" in **1982 ACM Sym LISP & FP**, pp 253-264.

This paper describes a semantic theory for lazy evaluation. It also discusses results derived from this theory.

Chiarini, A., "On FP Languages Combining Forms" in **SIGPLAN Notices**, Vol 15, No 9, Sep 1980, pp 25-27.

This article describes several combining forms implemented in an FP system developed by the author. The syntactic form of the language differs slightly from Backus' FP.

Cohen, A. Toni and Thomas J. Myers, "Towards an Algebra of Nondeterministic Programs" in **1982 ACM Sym LISP & FP**, pp 235-242.

In this paper, the authors extend Backus' algebra of programs to include nondeterministic operators. Laws applying to these new operators are also introduced.

Dannenburg, Roger B., "Artic: A Functional Language for Real-Time Control" in **1984 ACM Sym LISP & FP**, pp 96-103.

This paper describes a functional language with mechanisms for specifying and implementing real-time control systems. The language is claimed to be especially attractive because it eliminates the need for programmer concern with execution sequence.

Darlington, J., Peter Henderson, and D. A. Turner, editors, **Functional Programming and its Applications**, Cambridge University Press, Cambridge, England, 1982.

This book is a textbook for a course in functional programming. Topics include the functional style of programming, functional programming in languages other than Backus' FP, methods of program transformation, and specialized architectures for functional languages.

Dosch, Walter and Bernhard Moller, "Busy and Lazy FP with Infinite Objects" in **1984 ACM Sym LISP & FP**, pp 282-292.

A variant of Backus' FP is introduced in this paper. This language allows infinite trees by relaxing the requirement that all functions be strict.

Dumas, Luis and Robin Milner, "Principle type-schemes for functional programs" in **Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages**, Albuquerque, NM, Jan 25-27, 1982, pp 207-212.

This paper discusses the type scheme of the language ML. The scheme does not require a programmer to declare types; rather, it requires the compiler to infer types.

Dybvig, R. Kent and Bruce T. Smith, "A Semantic Editor" in **Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments**, Seattle, WA, June 25-28,

1985, in **SIGPLAN Notices**, Vol 20, No 7, July 1985 (85 Sym Language Issues), pp 74-82.

This paper describes an editor for Backus' FP. The editor allows manipulation of programs based on the algebra of programs.

Eggert, Paul R. and D. Val Schorre, "Logic enhancement: a method for extending logic programming languages" in **1982 ACM Sym LISP & FP**, pp 74-80.

This paper discusses methods for extending the logic language Prolog. One proposed extension is the adoption of functional notation based on the FP style.

Feldman, Gary, "Functional Specifications of a Text Editor" in **1982 ACM Sym LISP & FP**, pp 37-46.

This paper describes a formal specification technique based on Backus' FP. The technique is demonstrated by specifying a text editor.

Georgeff, M. P., "A Scheme for Implementing Functional Values on a Stack Machine" in **1982 ACM Sym LISP & FP**, pp 188-195.

In this paper, Georgeff proposes a method for implementing function valued expressions. This method is uses a traditional run-time stack for evaluation, and is claimed to produce good efficiency.

Givler, John S. and Richard B. Kieburtz, "Schema Recognition for Program Transformations" in **1984 ACM Sym LISP & FP**, pp 74-83.

In this paper, an FP dialect is used to examine program transformations. Also, algorithms are presented for determining if a given function is an instance of a schema for which a transformation is known.

Glaser, Hugh, Chris Hankin, and David Till, **Principles of Functional Programming**, Prentice-Hall International, London, 1984.

This book describes the basic principles of functional programming, specialized computer architectures, and several specific functional languages. The emphasis is on languages based on the lambda calculus.

Gries, David, **The Science of Programming**, Springer-Verlag, New York, 1981.

This book describes principles of correctness proofs for conventional programming languages.

Gull, W. E. and M. A. Jenkins, "Decisions for `Type´ in APL" in **Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages** (6th ACM Sym Prin Prog Lang), San Antonio, Texas, Jan 29-31, 1979, pp 190-196.

This paper discusses the APL notion of types.

Hall, Cordelia and John T. O´Donnell, "Debugging in a Side Effect Free Programming Environment" in **85 Sym Language Issues**, pp 60-68.

This paper describes debugging methods for the lambda based language Daisy.

Henderson, Peter, "Functional Geometry" in **1982 ACM Sym LISP & FP**, pp 179-187.

In this paper, a method for describing pictures is introduced. The method results in a functional program.

Henderson, Peter, **Functional Programming: Application and Implementation**, Prentice-Hall, London, 1980.

This book was not available to the author during the time the paper was being written. The reader who has access to it, should consult it.

Henderson, Peter and J. Morris, Jr., "A Lazy Evaluator" in **Conference Record of the Third Annual ACM Symposium on Principles of Programming Languages**, 1976, pp 95-103.

In this paper, the authors introduce the evaluation technique of lazy evaluation.

Hoffman, Christoph M. and Michael J. O´Donnell, "An Interpreter Generator Using Tree Pattern Matching" in **6th ACM Sym Prin Prog Lang**, pp 169-179.

This paper describes a technique for developing interpreters for nonprocedural languages. Equations are used as the basis, in a way analogous to context-free grammars for procedural languages.

Hogger, Christopher John, **Introduction to Logic Programming**, Academic Press, London, 1984.

This book explains the basic concepts of logic programming, using the language PROLOG.

Honschapp, U., W-M. Lippe, and F. Simon, "Compiling Functional Languages for von-Neumann Machines" in **Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems**, San Francisco, CA, June 27-29, 1983 in **SIGPLAN Notices**, Vol 18, No 6, June 1983 (83 Sym Prog Lang Issues), pp 22-27.

In this paper, a compiler for the language LISP/N is described. The compiler assumes a conventional von-Neumann computer architecture.

Hudak, Paul and Benjamin Goldberg, "Experiments in Diffused Combinator Reduction" in **1984 ACM Sym LISP & FP**, pp 167-176.

This paper presents a model of an architecture designed for implementing functional languages. This architectures uses a network of closely-coupled processors.

Hudak, Paul and David Kranz, "A Combinator-Based Compiler for a Functional Language" in **Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages** (11th ACM Sym Prin Prog Lang), Salt Lake City, Utah, Jan 15-18, 1984, pp 122-132.

In this paper, the authors propose that lambda based functional languages can be implemented efficiently by appropriate optimizing compilers. They also describe a method for building such compilers based on combinators. A progress report on their efforts to build a compiler is given, also.

Hughes, R. J. M., "Super Combinators: A New Implementation Method for Applicative Languages" in **1982 ACM Sym LISP & FP**, pp 1-10.

This paper describes a method for implementing lambda

calculus based functional languages. The method is called super combinators, and is stated to be faster than previously described methods.

Ida, Tetsuo and Jiro Tanaka, "Functional Programming with Streams" in **IP 83**, pp 265-270.

The authors introduce the notion of "streams" into Backus' FP. This extension is intended to eleminate nontermination of certain recursive equations, and to reduce computational complexity of functions.

Iverson, Kenneth E., **A Programming Language**, John Wiley and Sons, New York, 1962.

This book is the basis of the language APL. It was not intended to define a new language, but to introduce a style of programming.

Johnson, Steven D., "Applicative Programming and Digital Design" in **11th ACM Sym Prin Prog Lang**, pp 218-227.

This paper discusses the use of a functional programming language to describe circuit designs.

Jones, Simon L. Peyton, "An Investigation of the Relative Efficiencies of Combinators and Lambda Expressions" in **1982 ACM Sym LISP & FP**, pp 150-158.

This paper presents an experimental comparison of the combinator and traditional reducer methods of implementing lambda expressions. The results suggest that the combinator method is at least as efficient as conventional methods.

Katayama, Takuya, "Type Inference and Type Checking for Functional Programming Languages: A Reduced Computation Approach" in **1984 ACM Sym LISP & FP**, pp 263-272.

In this paper, the author proposes an approach to type inference and checking in functional languages. The approach is based on reducing computations on data to computations on types.

Kennaway, J. R. and M. R. Sleep, "Expressions as Processes" in **1982 ACM Sym LISP & FP**, pp 21-28.

This paper introduces a notation for expressing applicative expressions as processes. These processes are said to be implementable using data flow techniques.

Ligler, George T., "A Mathematical Approach to Language Design" in **Conference Record of the Second ACM Symposium on Principles of Programming Languages** (2nd ACM Sym Prin Prog Lang), Palo Alto, CA, Jan 20-22, 1975, pp 41-53.

This paper suggests that programming languages should be designed with mathematical proof techniques in mind. It is not directly related to functional programming, but the idea of taking a mathematical approach to language design is related.

MacLennan, Bruce J., "A Simple Software Environment Based on Objects and Relations" in **85 Sym Language Issues**, pp 199-207.

This paper introduces a programming environment that supports both functional and object-oriented programming. The functional language for the system is not described in detail.

MacQueen, David, "Modules for Standard ML" in **1984 ACM Sym LISP & FP**, pp 198-207.

This paper describes the module facility of the language ML.

MacQueen, D. B. and Raui Sethi, "A Semantic Model of Types for Applicative Languages" in **1982 ACM Sym LISP & FP**, pp 243-252.

In this paper, the authors develop a model for adding types to applicative languages.

Mago, Gyula, "Data Sharing in an FFP Machine" in **1982 ACM Sym LISP & FP**, pp 201-207.

This paper discusses the implementation of the Peterson-Wegman data-sharing algorithm for formal FP systems. The author's innovative computer architecture is also briefly described.

Malachi, Yonathan, Zohor Manna, and Richard Waldinger, "TABLOG: The Deductive-Tableau Programming Language" in **1984 ACM Sym**

LISP & FP, pp 323-330.

This paper describes the programming language TABLOG. The language combines functional and logic programming styles.

McCarthy, John, "Recursive Functions of Symbolic Expressions and Their Computation by Machine Part I" in **CACM**, Vol 3, No 4, Apr 1960, 184-195.

In this paper, McCarthy describes a formal method for defining recursive functions. He also discusses the original LISP system and implementation. Note: No Part II was ever published.

Metayer, D. Le, "Mechanical Analysis of Program Complexity" in **85 Sym Language Issues**, pp 69-73.

This paper describes a system to automatically evaluate the complexity of FP programs.

Milner, Robin, "A Proposal for Standard ML" in **1984 ACM Sym LISP & FP**, pp 184-197.

This paper describes a proposed standard for the strongly typed functional language ML.

Mishra, Prateek and Robert M. Keller, "Static inference of properties of applicative programs" in **11th ACM Sym Prin Prog Lang**, pp 235-244.

This paper discusses a method for deducing properties from programs written in a class of functional languages. The primary property mentioned is that of types.

Mitchell, John C., "Coercion and Type Inference" in **11th ACM Sym Prin Prog Lang**, pp 175-184.

This paper describes the principles of type inference and proposes a method for doing it.

Muchnick, Steve S. and Neil D. Jones, "A Fixed-Program Machine for Combinator Expression Evaluation" in **1982 ACM Sym LISP & FP**, pp 11-20.

An evaluation mechanism for combinator expressions is proposed in this paper. The method is claimed to be easily

implemented on conventional computers.

Naur, Peter, "Programming Languages, Natural Languages, and Mathematics" in **2nd ACM Sym Prin Prog Lang**, pp 137-48.

This paper relates the author's opinions concerning the future of programming language development. He suggests that no further major developments will occur. This is directly contradictory to the motivations behind functional programming languages.

O'Donnell, John T., "Dialogues: A Basis for Constructing Programming Environments" in **85 Sym Language Issues**, pp 19-27.

This paper discusses the description of communications in programming environments. The functional language Daisy is used for the description.

Pettorossi, Alberto, "A Powerful Strategy for Deriving Efficient Programs by Transformation" in **1984 ACM Sym LISP & FP**, pp 273-281.

This paper presents a technique for transforming recursive programs into iterative ones.

Pratt, Terrence W., **Programming Languages: Design and Implementation**, 2nd edition, Prentice-Hall, Englewood Cliffs, NJ, 1984.

This book is a general introduction to programming languages.

Proceedings of the Conference on Functional Programming Languages and Computer Architecture, ACM, October, 1981.

These proceedings contain several papers cited in works listed here.

Rosser, J. Barkley, "Highlights of the History of the Lambda-Calculus" in **1982 ACM Sym LISP & FP**, pp 216-255.

This paper describes the origins and development history of the lambda calculus.

Scott, Dana, "Mathematical Concepts in Programming Language Semantics" in **AFIPS Conference Proceedings**, Vol 40, 1972, pp 225-234.

This paper describes an application of mathematical concepts to defining language semantics. Although a von Neumann computer is assumed, the emphasis on mathematics is similar to that in functional programming.

Sheeran, Mary, "muFP, a language for VLSI design" in **1984 ACM Sym LISP & FP**, pp 104-112.

In this paper, Sheeran describes a VLSI design language that is a variant of Backus' FP.

Shultis, Jon, "A Functional Shell" in **83 Sym Prog Lang Issues**, pp 202-211.

This paper describes a command language interpreter based on Backus' FP. A partial implementation of the shell is discussed.

Stoyan, Herbert, "Early LISP History (1956-1959)" in **1984 ACM Sym LISP & FP**, pp 299-310.

This paper discusses the early history of LISP.

Stoye, W. R., T. J. W. Clarke, and A. C. Norman, "Some Practical Methods for Rapid Combinator Reduction" in **1984 ACM Sym LISP & FP**, pp 159-166.

This paper describes the SKIM II processor. SKIM II was designed for rapid evaluation of functional languages. The paper includes performance measures.

Subrahmanyam, P. A. and J. H. You, "Pattern Driven Lazy Reduction: A Unifying Evaluation Mechanism for Functional and Logic Programs" in **11th ACM Sym Prin Prog Lang**, pp 228-234.

This paper introduces an evaluation method that is claimed to be applicable to both functional and logic languages.

Wadler, Phillip, "Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile Time" in **1984 ACM Sym LISP & FP**, pp 45-52.

In this paper, the author introduces a program transformation system for functional languages. This system eliminates all intermediate lists whenever possible. The author states that the system is applicable to a large class of programs, but not to all programs.

Williams, John H., "Formal Representations for Recursively Defined Functional Programs" in **LN in CS Vol 107**, pp 460-470.

This paper proposes an algorithm for producing formal representations of any informal FP function.

Williams, John H., "On the Development of the Algebra of Functional Programs" in **ACM Transactions on Programming Languages and Systems**, Vol 4, No 4, Oct 1982, pp 733-757.

In this paper, Williams develops further Backus' algebra of programs. He proves some new expansion theorems related to non-linear functions. These theorems extend the class of functions for which expansion theorems are applicable.

Standard Bibliographic Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-89019		

4. Title and Subtitle	5. Report Date
A Survey of Functional Programming Language Principles	September 1986
	6. Performing Organization Code: 505-65-11-02

7. Author(s)	8. Performing Organization Report No.
C. Michael Holloway	

9. Performing Organization Name and Address	10. Work Unit No.
NASA Langley Research Center Hampton, Virginia 23665-5225	11. Contract or Grant No.

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Washington, DC 20546	Technical Memorandum
	14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

Research in the area of functional programming languages has intensified in the 8 years since John Backus' Turing Award Lecture on the topic was published. The purpose of this paper is to present a survey of the ideas of functional programming languages. The paper assumes the reader is comfortable with mathematics and has knowledge of the basic principles of traditional programming languages, but does not assume any prior knowledge of the ideas of functional languages.

A simple functional language is defined and used to illustrate the basic ideas. Topics discussed include the reasons for developing functional languages, methods of expressing concurrency, the algebra of functional programming languages, program transformation techniques, and implementations of functional languages. Existing functional languages are also mentioned. The paper concludes with the author's opinions as to the future of functional languages. An annotated bibliography on the subject is also included.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
Programming Languages Functional Programming	Unclassified-Unlimited Subject Category 61

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	85	A05

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA Langley Form 63 (June 1985)

Standard Bibliographic Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-89019		

4. Title and Subtitle	5. Report Date
A Survey of Functional Programming Language Principles	September 1986
	6. Performing Organization Code 505-65-11-02

7. Author(s)	8. Performing Organization Report No.
C. Michael Holloway	

9. Performing Organization Name and Address	10. Work Unit No.
NASA Langley Research Center Hampton, Virginia 23665-5225	11. Contract or Grant No.

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Washington, DC 20546	Technical Memorandum
	14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

@ABS
Research in the area of functional programming languages has intensified in the 8 years since John Backus' Turing Award Lecture on the topic was published. The purpose of this paper is to present a survey of the ideas of functional programming languages. The paper assumes the reader is comfortable with mathematics and has knowledge of the basic principles of traditional programming languages, but does not assume any prior knowledge of the ideas of functional languages.

A simple functional language is defined and used to illustrate the basic ideas. Topics discussed include the reasons for developing functional languages, methods of expressing concurrency, the algebra of functional programming languages, program transformation techniques, and implementations of functional languages. Existing functional languages are also mentioned. The paper concludes with the author's opinions as to the future of functional languages. An annotated bibliography on the subject is also included.

@ABS Author.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
Programming Languages Functional Programming	Unclassified-Unlimited Subject Category 61

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	85	A05

For sale by the National Technical Information Service, Springfield, Virginia 22161