The 1977 ACM Turing Award was presented to John Backus at the ACM Annual Conference in Seattle, October 17. In introducing the recipient, Jean E. Sammet, Chairman of the Awards Committee, made the following comments and read a portion of the final citation. The full announcement is in the September 1977 issue of *Communications,* page 681.

"Probably there is nobody in the room who has not heard of Fortran and most of you have probably used it at least once, or at least looked over the shoulder of someone who was writing a Fortran program. There are probably almost as many people who have heard the letters BNF but don't necessarily know what they stand for. Well, the B is for Backus, and the other letters are explained in the formal citation. These two contributions, in my opinion, are among the half dozen most important technical contributions to the computer field and both were made by John Backus (which in the Fortran case also involved some colleagues). It is for these contributions that he is receiving this year's Turing award.

The short form of his citation is for 'profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran, and for seminal publication of formal procedures for the specifications of programming languages.'

The most significant part of the full citation is as follows:

'. . . Backus headed a small IBM group in New York City during the early 1950s. The earliest product of this group's efforts was a high-level language for scientific and technical computations called Fortran. This same group designed the first system to translate Fortran programs into machine language. They employed novel optimizing techniques to generate fast machine-language programs. Many other compilers for the language were developed, first on IBM machines, and later on virtually every make of computer. Fortran was adopted as a U.S. national standard in 1966.

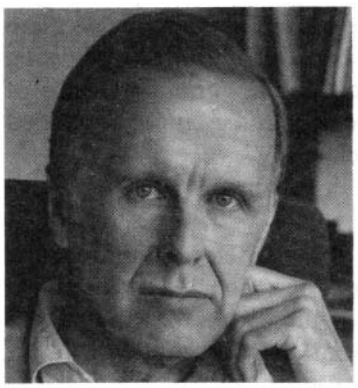
During the latter part of the 1950s, Backus served on the international committees which developed Algol 58 and a later version, Algol 60. The language Algol, and its derivative compilers, received broad acceptance in Europe as a means for developing programs and as a formal means of publishing the algorithms on which the programs are based.

In 1959, Backus presented a paper at the UNESCO conference in Paris on the syntax and semantics of a proposed international algebraic language. In this paper, he was the first to employ a formal technique for specifying the syntax of programming languages. The formal notation became known as BNF—standing for "Backus Normal Form," or "Backus Naur Form" to recognize the further contributions by Peter Naur of Denmark.

Thus, Backus has contributed strongly both to the pragmatic world of problem-solving on computers and to the theoretical world existing at the interface between artificial languages and computational linguistics. Fortran remains one of the most widely used programming languages in the world. Almost all programming languages are now described with some type of formal syntactic definition.' "

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

Associated with the functional style of programming is an algebra of programs whose variables range over programs and whose operations are combining forms. This algebra can be used to transform programs and to solve equations whose "unknowns" are programs in much the same way one transforms equations in high school algebra. These transformations are given by algebraic laws and are carried out in the same language in which programs are written. Combining forms are chosen not only for their programming power but also for the power of their associated algebraic laws. General theorems of the algebra give the detailed behavior and termination conditions for large classes of programs.

A new class of computing systems uses the functional programming style both in its programming language and in its state transition rules. Unlike von Neumann languages, these systems have semantics loosely coupled to states—only one state transition occurs per major computation.

Key Words and Phrases: functional programming, algebra of programs, combining forms, functional forms, programming languages, von Neumann computers, von Neumann languages, models of computing systems, applicative computing systems, applicative state transition systems, program transformation, program correctness, program termination, metacomposition

CR Categories: 4.20, 4.29, 5.20, 5.24, 5.26

Introduction

I deeply appreciate the honor of the ACM invitation to give the 1977 Turing Lecture and to publish this account of it with the details promised in the lecture. Readers wishing to see a summary of this paper should turn to Section 16, the last section.

1. Conventional Programming Languages: Fat and Flabby

Programming languages appear to be in trouble. Each successive language incorporates, with a little cleaning up, all the features of its predecessors plus a few more. Some languages have manuals exceeding 500 pages; others cram a complex description into shorter manuals by using dense formalisms. The Department of Defense has current plans for a committee-designed language standard that could require a manual as long as 1,000 pages. Each new language claims new and fashionable features, such as strong typing or structured control statements, 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.

Since large increases in size bring only small increases in power, smaller, more elegant languages such as Pascal continue to be popular. But 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.

For twenty years programming languages have been steadily progressing toward their present condition of obesity; as a result, the study and invention of programming languages has lost much of its excitement. Instead, it is now the province of those who prefer to work with thick compendia of details rather than wrestle with new ideas. Discussions about programming languages often resemble medieval debates about the number of angels that can dance on the head of a pin instead of exciting contests between fundamentally differing concepts.

Many creative computer scientists have retreated from inventing languages to inventing tools for describing them. Unfortunately, they have been largely content to apply their elegant new tools to studying the warts and moles of existing languages. After examining the appalling type structure of conventional languages, using the elegant tools developed by Dana Scott, it is surprising that so many of us remain passively content with that structure instead of energetically searching for new ones.

The purpose of this article is twofold; first, to suggest that basic defects in the framework of conventional languages make their expressive weakness and their cancerous growth inevitable, and second, to suggest some alternate avenues of exploration toward the design of new kinds of languages.

2. Models of Computing Systems

Underlying every programming language is a model of a computing system that its programs control. Some models are pure abstractions, some are represented by hardware, and others by compiling or interpretive programs. Before we examine conventional languages more closely, it is useful to make a brief survey of existing models as an introduction to the current universe of alternatives. Existing models may be crudely classified by the criteria outlined below.

2.1 Criteria for Models

2.1.1 Foundations. Is there an elegant and concise mathematical description of the model? Is it useful in proving helpful facts about the behavior of the model? Or is the model so complex that its description is bulky and of little mathematical use?

2.1.2 History sensitivity. Does the model include a notion of storage, so that one program can save information that can affect the behavior of a later program? That is, is the model history sensitive?

2.1.3 Type of semantics. Does a program successively transform states (which are not programs) until a terminal state is reached (state-transition semantics)? Are states simple or complex? Or can a "program" be successively reduced to simpler "programs" to yield a final

"normal form program," which is the result (reduction semantics)?

2.1.4 Clarity and conceptual usefulness of programs. Are programs of the model clear expressions of a process or computation? Do they embody concepts that help us to formulate and reason about processes?

2.2 Classification of Models

Using the above criteria we can crudely characterize three classes of models for computing systems—simple operational models, applicative models, and von Neumann models.

2.2.1 Simple operational models. Examples: Turing machines, various automata. *Foundations*: concise and useful. *History sensitivity*: have storage, are history sensitive. *Semantics*: state transition with very simple states. *Program clarity*: programs unclear and conceptually not helpful.

2.2.2 Applicative models. Examples: Church's lambda calculus [5], Curry's system of combinators [6], pure Lisp [17], functional programming systems described in this paper. *Foundations*: concise and useful. *History sensitivity*: no storage, not history sensitive. *Semantics*: reduction semantics, no states. *Program clarity*: programs can be clear and conceptually useful.

2.2.3 Von Neumann models. Examples: von Neumann computers, conventional programming languages. *Foundations*: complex, bulky, not useful. *History sensitivity*: have storage, are history sensitive. *Semantics*: state transition with complex states. *Program clarity*: programs can be moderately clear, are not very useful conceptually.

The above classification is admittedly crude and debatable. Some recent models may not fit easily into any of these categories. For example, the data-flow languages developed by Arvind and Gostelow [1], Dennis [7], Kosinski [13], and others partly fit the class of simple operational models, but their programs are clearer than those of earlier models in the class and it is perhaps possible to argue that some have reduction semantics. In any event, this classification will serve as a crude map of the territory to be discussed. We shall be concerned only with applicative and von Neumann models.

3. Von Neumann Computers

In order to understand the problems of conventional programming languages, we must first examine their intellectual parent, the von Neumann computer. What is a von Neumann computer? When von Neumann and others conceived it over thirty years ago, it was an elegant, practical, and unifying idea that simplified a number of engineering and programming problems that existed then. Although the conditions that produced its architecture have changed radically, we nevertheless still identify the notion of "computer" with this thirty year old concept.

In its simplest form a von Neumann computer has three parts: a central processing unit (or CPU), a store, and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store). I propose to call this tube the *von Neumann bottleneck*. The task of a program is to change the contents of the store in some major way; when one considers that this task must be accomplished entirely by pumping single words back and forth through the von Neumann bottleneck, the reason for its name becomes clear.

Ironically, a large part of the traffic in the bottleneck is not useful data but merely names of data, as well as operations and data used only to compute such names. Before a word can be sent through the tube its address must be in the CPU; hence it must either be sent through the tube from the store or be generated by some CPU operation. If the address is sent from the store, then *its* address must either have been sent from the store or generated in the CPU, and so on. If, on the other hand, the address is generated in the CPU, it must be generated either by a fixed rule (e.g., "add 1 to the program counter") or by an instruction that was sent through the tube, in which case *its* address must have been sent ... and so on.

Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself but where to find it.

4. Von Neumann Languages

Conventional programming languages are basically high level, complex versions of the von Neumann computer. Our thirty year old belief that there is only one kind of computer is the basis of our belief that there is only one kind of programming language, the conventional—von Neumann—language. The differences between Fortran and Algol 68, although considerable, are less significant than the fact that both are based on the programming style of the von Neumann computer. Although I refer to conventional languages as "von Neumann languages" to take note of their origin and style, I do not, of course, blame the great mathematician for their complexity. In fact, some might say that I bear some responsibility for that problem.

Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic.

The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does.

Consider a typical program; at its center are a number of assignment statements containing some subscripted variables. Each assignment statement produces a one-word result. The program must cause these statements to be executed many times, while altering subscript values, in order to make the desired overall change in the store, since it must be done one word at a time. The programmer is thus concerned with the flow of words through the assignment bottleneck as he designs the nest of control statements to cause the necessary repetitions.

Moreover, the assignment statement splits programming into two worlds. The first world comprises the right sides of assignment statements. This is an orderly world of expressions, a world that has useful algebraic properties (except that those properties are often destroyed by side effects). It is the world in which most useful computation takes place.

The second world of conventional programming languages is the world of statements. The primary statement in that world is the assignment statement itself. All the other statements of the language exist in order to make it possible to perform a computation that must be based on this primitive construct: the assignment statement.

This world of statements is a disorderly one, with few useful mathematical properties. Structured programming can be seen as a modest effort to introduce some order into this chaotic world, but it accomplishes little in attacking the fundamental problems created by the word-at-a-time von Neumann style of programming, with its primitive use of loops, subscripts, and branching flow of control.

Our fixation on von Neumann languages has continued the primacy of the von Neumann computer, and our dependency on *it* has made non-von Neumann languages uneconomical and has limited their development. The absence of full scale, effective programming styles founded on non-von Neumann principles has deprived designers of an intellectual foundation for new computer architectures. (For a brief discussion of that topic, see Section 15.)

Applicative computing systems' lack of storage and history sensitivity is the basic reason they have not provided a foundation for computer design. Moreover, most applicative systems employ the substitution operation of the lambda calculus as their basic operation. This operation is one of virtually unlimited power, but its complete and efficient realization presents great difficulties to the machine designer. Furthermore, in an effort to introduce storage and to improve their efficiency on von Neumann computers, applicative systems have tended to become engulfed in a large von Neumann system. For example, pure Lisp is often buried in large extensions with many von Neumann features. The resulting complex systems offer little guidance to the machine designer.

5. Comparison of von Neumann and Functional Programs

To get a more detailed picture of some of the defects of von Neumann languages, let us compare a conventional program for inner product with a functional one written in a simple language to be detailed further on.

5.1 A von Neumann Program for Inner Product

```
c := 0
for i := 1 step 1 until n do
  c := c + a[i]×b[i]
```

Several properties of this program are worth noting:

a) Its statements operate on an invisible "state" according to complex rules.

b) It is not hierarchical. Except for the right side of the assignment statement, it does not construct complex entities from simpler ones. (Larger programs, however, often do.)

c) It is dynamic and repetitive. One must mentally execute it to understand it.

d) It computes word-at-a-time by repetition (of the assignment) and by modification (of variable i).

e) Part of the data, n, is in the program; thus it lacks generality and works only for vectors of length n.

f) It names its arguments; it can only be used for vectors a and b. To become general, it requires a procedure declaration. These involve complex issues (e.g., call-by-name versus call-by-value).

g) Its "housekeeping" operations are represented by symbols in scattered places (in the **for** statement and the subscripts in the assignment). This makes it impossible to consolidate housekeeping operations, the most common of all, into single, powerful, widely useful operators. Thus in programming those operations one must always start again at square one, writing "**for** i := ..." and "**for** j := ..." followed by assignment statements sprinkled with i's and j's.

5.2 A Functional Program for Inner Product

Def Innerproduct
≡ (Insert +)∘(ApplyToAll ×)∘Transpose

Or, in abbreviated form:

Def IP ≡ (/+)∘(α×)∘Trans.

Composition (∘), Insert (/), and ApplyToAll (α) are *functional forms* that combine existing functions to form new ones. Thus $f \circ g$ is the function obtained by applying first g and then f, and αf is the function obtained by applying f to every *member* of the argument. If we write $f{:}x$ for the result of applying f to the object x, then we can explain each step in evaluating Innerproduct applied to the pair of vectors <<1, 2, 3>, <6, 5, 4>> as follows:

```
IP:<<1,2,3>, <6,5,4>> =
Definition of IP          ⇒ (/+)∘(α×)∘Trans: <<1,2,3>, <6,5,4>>
Effect of composition, ∘  ⇒ (/+):((α×):(Trans:
                                  <<1,2,3>, <6,5,4>>))
```

Applying Transpose	⇒ (/+):((α×): <<1,6>, <2,5>, <3,4>>)
Effect of ApplyToAll, α	⇒ (/+): <×: <1,6>, ×: <2,5>, ×: <3,4>>
Applying ×	⇒ (/+): <6,10,12>
Effect of Insert, /	⇒ +: <6, +: <10,12>>
Applying +	⇒ +: <6,22>
Applying + again	⇒ 28

Let us compare the properties of this program with those of the von Neumann program.

a) It operates only on its arguments. There are no hidden states or complex transition rules. There are only two kinds of rules, one for applying a function to its argument, the other for obtaining the function denoted by a functional form such as composition, $f \circ g$, or ApplyToAll, αf, when one knows the functions f and g, the *parameters* of the forms.

b) It is hierarchical, being built from three simpler functions (+, ×, Trans) and three functional forms $f \circ g$, αf, and $/f$.

c) It is static and nonrepetitive, in the sense that its structure is helpful in understanding it without mentally executing it. For example, if one understands the action of the forms $f \circ g$ and αf, and of the functions × and Trans, then one understands the action of $\alpha\times$ and of $(\alpha\times) \circ$Trans, and so on.

d) It operates on whole conceptual units, not words; it has three steps; no step is repeated.

e) It incorporates no data; it is completely general; it works for any pair of conformable vectors.

f) It does not name its arguments; it can be applied to any pair of vectors without any procedure declaration or complex substitution rules.

g) It employs housekeeping forms and functions that are generally useful in many other programs; in fact, only + and × are not concerned with housekeeping. These forms and functions can combine with others to create higher level housekeeping operators.

Section 14 sketches a kind of system designed to make the above functional style of programming available in a history-sensitive system with a simple framework, but much work remains to be done before the above applicative style can become the basis for elegant and practical programming languages. For the present, the above comparison exhibits a number of serious flaws in von Neumann programming languages and can serve as a starting point in an effort to account for their present fat and flabby condition.

6. Language Frameworks versus Changeable Parts

Let us distinguish two parts of a programming language. First, its *framework* which gives the overall rules of the system, and second, its *changeable parts,* whose existence is anticipated by the framework but whose particular behavior is not specified by it. For example, the **for** statement, and almost all other statements, are part of Algol's framework but library functions and user-defined procedures are changeable parts. Thus the framework of a language describes its fixed features and provides a general environment for its changeable features.

Now suppose a language had a small framework which could accommodate a great variety of powerful features entirely as changeable parts. Then such a framework could support many different features and styles without being changed itself. In contrast to this pleasant possibility, von Neumann languages always seem to have an immense framework and very limited changeable parts. What causes this to happen? The answer concerns two problems of von Neumann languages.

The first problem results from the von Neumann style of word-at-a-time programming, which requires that words flow back and forth to the state, just like the flow through the von Neumann bottleneck. Thus a von Neumann language must have a semantics closely coupled to the state, in which every detail of a computation changes the state. The consequence of this semantics closely coupled to states is that every detail of every feature must be built into the state and its transition rules.

Thus every feature of a von Neumann language must be spelled out in stupefying detail in its framework. Furthermore, many complex features are needed to prop up the basically weak word-at-a-time style. The result is the inevitable rigid and enormous framework of a von Neumann language.

7. Changeable Parts and Combining Forms

The second problem of von Neumann languages is that their changeable parts have so little expressive power. Their gargantuan size is eloquent proof of this; after all, if the designer knew that all those complicated features, which he now builds into the framework, could be added later on as changeable parts, he would not be so eager to build them into the framework.

Perhaps the most important element in providing powerful changeable parts in a language is the availability of combining forms that can be generally used to build new procedures from old ones. Von Neumann languages provide only primitive combining forms, and the von Neumann framework presents obstacles to their full use.

One obstacle to the use of combining forms is the split between the expression world and the statement world in von Neumann languages. Functional forms naturally belong to the world of expressions; but no matter how powerful they are they can only build expressions that produce a one-word result. And it is in the statement world that these one-word results must be combined into the overall result. Combining single words is not what we really should be thinking about, but it is a large part of programming any task in von Neumann languages. To help assemble the overall result from single words these languages provide some primitive combining forms in the statement world—the **for, while,** and **if-then-else** statements—but the split between the

two worlds prevents the combining forms in either world from attaining the full power they can achieve in an undivided world.

A second obstacle to the use of combining forms in von Neumann languages is their use of elaborate naming conventions, which are further complicated by the substitution rules required in calling procedures. Each of these requires a complex mechanism to be built into the framework so that variables, subscripted variables, pointers, file names, procedure names, call-by-value formal parameters, call-by-name formal parameters, and so on, can all be properly interpreted. All these names, conventions, and rules interfere with the use of simple combining forms.

8. APL versus Word-at-a-Time Programming

Since I have said so much about word-at-a-time programming, I must now say something about APL [12]. We owe a great debt to Kenneth Iverson for showing us that there are programs that are neither word-at-a-time nor dependent on lambda expressions, and for introducing us to the use of new functional forms. And since APL assignment statements can store arrays, the effect of its functional forms is extended beyond a single assignment.

Unfortunately, however, APL still splits programming into a world of expressions and a world of statements. Thus the effort to write one-line programs is partly motivated by the desire to stay in the more orderly world of expressions. APL has exactly three functional forms, called inner product, outer product, and reduction. These are sometimes difficult to use, there are not enough of them, and their use is confined to the world of expressions.

Finally, APL semantics is still too closely coupled to states. Consequently, despite the greater simplicity and power of the language, its framework has the complexity and rigidity characteristic of von Neumann languages.

9. Von Neumann Languages Lack Useful Mathematical Properties

So far we have discussed the gross size and inflexibility of von Neumann languages; another important defect is their lack of useful mathematical properties and the obstacles they present to reasoning about programs. Although a great amount of excellent work has been published on proving facts about programs, von Neumann languages have almost no properties that are helpful in this direction and have many properties that are obstacles (e.g., side effects, aliasing).

Denotational semantics [23] and its foundations [20, 21] provide an extremely helpful mathematical understanding of the domain and function spaces implicit in programs. When applied to an applicative language (such as that of the "recursive programs" of [16]), its foundations provide powerful tools for describing the language and for proving properties of programs. When applied to a von Neumann language, on the other hand, it provides a precise semantic description and is helpful in identifying trouble spots in the language. But the complexity of the language is mirrored in the complexity of the description, which is a bewildering collection of productions, domains, functions, and equations that is only slightly more helpful in proving facts about programs than the reference manual of the language, since it is less ambiguous.

Axiomatic semantics [11] precisely restates the inelegant properties of von Neumann programs (i.e., transformations on states) as transformations on predicates. The word-at-a-time, repetitive game is not thereby changed, merely the playing field. The complexity of this axiomatic game of proving facts about von Neumann programs makes the successes of its practitioners all the more admirable. Their success rests on two factors in addition to their ingenuity: First, the game is restricted to small, weak subsets of full von Neumann languages that have states vastly simpler than real ones. Second, the new playing field (predicates and their transformations) is richer, more orderly and effective than the old (states and their transformations). But restricting the game and transferring it to a more effective domain does not enable it to handle real programs (with the necessary complexities of procedure calls and aliasing), nor does it eliminate the clumsy properties of the basic von Neumann style. As axiomatic semantics is extended to cover more of a typical von Neumann language, it begins to lose its effectiveness with the increasing complexity that is required.

Thus denotational and axiomatic semantics are descriptive formalisms whose foundations embody elegant and powerful concepts; but using them to describe a von Neumann language can not produce an elegant and powerful language any more than the use of elegant and modern machines to build an Edsel can produce an elegant and modern car.

In any case, proofs about programs use the language of logic, not the language of programming. Proofs talk *about* programs but cannot involve them directly since the axioms of von Neumann languages are so unusable. In contrast, many ordinary proofs are derived by algebraic methods. These methods require a language that has certain algebraic properties. Algebraic laws can then be used in a rather mechanical way to transform a problem into its solution. For example, to solve the equation

$$ax + bx = a + b$$

for x (given that $a+b \neq 0$), we mechanically apply the distributive, identity, and cancellation laws, in succession, to obtain

$$(a + b)x = a + b$$
$$(a + b)x = (a + b)1$$
$$x = 1.$$

Thus we have proved that x = 1 without leaving the "language" of algebra. Von Neumann languages, with their grotesque syntax, offer few such possibilities for transforming programs.

As we shall see later, programs can be expressed in a language that has an associated algebra. This algebra can be used to transform programs and to solve some equations whose "unknowns" are programs, in much the same way one solves equations in high school algebra. Algebraic transformations and proofs use the language of the programs themselves, rather than the language of logic, which talks about programs.

10. What Are the Alternatives to von Neumann Languages?

Before discussing alternatives to von Neumann languages, let me remark that I regret the need for the above negative and not very precise discussion of these languages. But the complacent acceptance most of us give to these enormous, weak languages has puzzled and disturbed me for a long time. I am disturbed because that acceptance has consumed a vast effort toward making von Neumann languages fatter that might have been better spent in looking for new structures. For this reason I have tried to analyze some of the basic defects of conventional languages and show that those defects cannot be resolved unless we discover a new kind of language framework.

In seeking an alternative to conventional languages we must first recognize that a system cannot be history sensitive (permit execution of one program to affect the behavior of a subsequent one) unless the system has some kind of state (which the first program can change and the second can access). Thus a history-sensitive model of a computing system must have a state-transition semantics, at least in this weak sense. But this does *not* mean that every computation must depend heavily on a complex state, with many state changes required for each small part of the computation (as in von Neumann languages).

To illustrate some alternatives to von Neumann languages, I propose to sketch a class of history-sensitive computing systems, where each system: a) has a loosely coupled state-transition semantics in which a state transition occurs only once in a major computation; b) has a simply structured state and simple transition rules; c) depends heavily on an underlying applicative system both to provide the basic programming language of the system and to describe its state transitions.

These systems, which I call applicative state transition (or AST) systems, are described in Section 14. These simple systems avoid many of the complexities and weaknesses of von Neumann languages and provide for a powerful and extensive set of changeable parts. However, they are sketched only as crude examples of a vast area of non-von Neumann systems with various attractive properties. I have been studying this area for the past three or four years and have not yet found a satisfying solution to the many conflicting requirements that a good language must resolve. But I believe this search has indicated a useful approach to designing non-von Neumann languages.

This approach involves four elements, which can be summarized as follows.

a) *A functional style of programming without variables.* A simple, informal functional programming (FP) system is described. It is based on the use of combining forms for building programs. Several programs are given to illustrate functional programming.

b) *An algebra of functional programs.* An algebra is described whose variables denote FP functional programs and whose "operations" are FP functional forms, the combining forms of FP programs. Some laws of the algebra are given. Theorems and examples are given that show how certain function expressions may be transformed into equivalent infinite expansions that explain the behavior of the function. The FP algebra is compared with algebras associated with the classical applicative systems of Church and Curry.

c) *A formal functional programming system.* A formal (FFP) system is described that extends the capabilities of the above informal FP systems. An FFP system is thus a precisely defined system that provides the ability to use the functional programming style of FP systems and their algebra of programs. FFP systems can be used as the basis for applicative state transition systems.

d) *Applicative state transition systems.* As discussed above. The rest of the paper describes these four elements, gives some brief remarks on computer design, and ends with a summary of the paper.

11. Functional Programming Systems (FP Systems)

11.1 Introduction

In this section we give an informal description of a class of simple applicative programming systems called functional programming (FP) systems, in which "programs" are simply functions without variables. The description is followed by some examples and by a discussion of various properties of FP systems.

An FP system is founded on the use of a fixed set of combining forms called functional forms. These, plus simple definitions, are the only means of building new functions from existing ones; they use no variables or substitution rules, and they become the operations of an associated algebra of programs. All the functions of an FP system are of one type: they map objects into objects and always take a single argument.

In contrast, a lambda-calculus based system is founded on the use of the lambda expression, with an associated set of substitution rules for variables, for building new functions. The lambda expression (with its substitution rules) is capable of defining all possible computable functions of all possible types and of any number of arguments. This freedom and power has its

disadvantages as well as its obvious advantages. It is analogous to the power of unrestricted control statements in conventional languages: with unrestricted freedom comes chaos. If one constantly invents new combining forms to suit the occasion, as one can in the lambda calculus, one will not become familiar with the style or useful properties of the few combining forms that are adequate for all purposes. Just as structured programming eschews many control statements to obtain programs with simpler structure, better properties, and uniform methods for understanding their behavior, so functional programming eschews the lambda expression, substitution, and multiple function types. It thereby achieves programs built with familiar functional forms with known useful properties. These programs are so structured that their behavior can often be understood and proven by mechanical use of algebraic techniques similar to those used in solving high school algebra problems.

Functional forms, unlike most programming constructs, need not be chosen on an ad hoc basis. Since they are the operations of an associated algebra, one chooses only those functional forms that not only provide powerful programming constructs, but that also have attractive algebraic properties: one chooses them to maximize the strength and utility of the algebraic laws that relate them to other functional forms of the system.

In the following description we shall be imprecise in not distinguishing between (a) a function symbol or expression and (b) the function it denotes. We shall indicate the symbols and expressions used to denote functions by example and usage. Section 13 describes a formal extension of FP systems (FFP systems); they can serve to clarify any ambiguities about FP systems.

11.2 Description

An FP system comprises the following:
1) a set O of *objects*;
2) a set F of *functions* f that map objects into objects;
3) an operation, *application*;
4) a set **F** of *functional forms*; these are used to combine existing functions, or objects, to form new functions in F;
5) a set D of *definitions* that define some functions in F and assign a name to each.

What follows is an informal description of each of the above entities with examples.

11.2.1 Objects, O. An *object* x is either an *atom*, a *sequence* $\langle x_1, \ldots, x_n \rangle$ whose *elements* x_i are objects, or $\perp$ ("bottom" or "undefined"). Thus the choice of a set A of atoms determines the set of objects. We shall take A to be the set of nonnull strings of capital letters, digits, and special symbols not used by the notation of the FP system. Some of these strings belong to the class of atoms called "numbers." The atom ϕ is used to denote the empty sequence and is the only object which is both an atom and a sequence. The atoms T and F are used to denote "true" and "false."

There is one important constraint in the construction of objects: if x is a sequence with $\perp$ as an element, then $x = \perp$. That is, the "sequence constructor" is "$\perp$-preserving." Thus no proper sequence has $\perp$ as an element.

Examples of objects

$\perp \quad 1.5 \quad \phi \quad AB3 \quad \langle AB, 1, 2.3 \rangle$
$\langle A, \langle\langle B\rangle, C\rangle, D\rangle \quad \langle A, \perp\rangle = \perp$

11.2.2 Application. An FP system has a single operation, application. If f is a function and x is an object, then $f{:}x$ is an *application* and denotes the object which is the result of applying f to x. f is the *operator* of the application and x is the *operand.*

Examples of applications

$+{:}\langle 1,2\rangle = 3 \quad \text{tl}{:}\langle A,B,C\rangle = \langle B,C\rangle$
$1{:}\langle A,B,C\rangle = A \quad 2{:}\langle A,B,C\rangle = B$

11.2.3 Functions, F. All functions f in F map objects into objects and are *bottom-preserving*: $f{:}\perp = \perp$, for all f in F. Every function in F is either *primitive*, that is, supplied with the system, or it is *defined* (see below), or it is a *functional form* (see below).

It is sometimes useful to distinguish between two cases in which $f{:}x{=}\perp$. If the computation for $f{:}x$ terminates and yields the object $\perp$, we say f is *undefined* at x, that is, f terminates but has no meaningful value at x. Otherwise we say f is *nonterminating* at x.

Examples of primitive functions

Our intention is to provide FP systems with widely useful and powerful primitive functions rather than weak ones that could then be used to define useful ones. The following examples define some typical primitive functions, many of which are used in later examples of programs. In the following definitions we use a variant of McCarthy's conditional expressions [17]; thus we write

$$p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n; e_{n+1}$$

instead of McCarthy's expression

$$(p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n, T \rightarrow e_{n+1}).$$

The following definitions are to hold for all objects x, x_i, y, y_i, z, z_i:

Selector functions

$$1{:}x \equiv x{=}\langle x_1, \ldots, x_n\rangle \rightarrow x_1; \perp$$

and for any positive integer s

$$s{:}x \equiv x = \langle x_1, \ldots, x_n \rangle \ \& \ n \geq s \rightarrow x_s; \perp$$

Thus, for example, $3{:}\langle A,B,C\rangle = C$ and $2{:}\langle A\rangle = \perp$. Note that the function symbols 1, 2, etc. are distinct from the atoms *1*, *2*, etc.

Tail

$$\text{tl}{:}x \equiv x{=}\langle x_1\rangle \rightarrow \phi;$$
$$x{=}\langle x_1, \ldots, x_n\rangle \ \& \ n \geq 2 \rightarrow \langle x_2, \ldots, x_n\rangle; \perp$$

Identity

$$\text{id}{:}x \equiv x$$

Atom
atom:$x \equiv x$ is an atom $\rightarrow T$; $x \neq \perp \rightarrow F$; $\perp$

Equals
eq:$x \equiv x=\langle y,z\rangle$ & $y=z \rightarrow T$; $x=\langle y,z\rangle$ & $y \neq z \rightarrow F$; $\perp$

Null
null:$x \equiv x=\phi \rightarrow T$; $x \neq \perp \rightarrow F$; $\perp$

Reverse
reverse:$x \equiv x=\phi \rightarrow \phi$;
$x=\langle x_1, \ldots, x_n\rangle \rightarrow \langle x_n, \ldots, x_1\rangle$; $\perp$

Distribute from left; distribute from right
distl:$x \equiv x=\langle y,\phi\rangle \rightarrow \phi$;
$x=\langle y,\langle z_1, \ldots, z_n\rangle\rangle \rightarrow \langle\langle y,z_1\rangle, \ldots, \langle y,z_n\rangle\rangle$; $\perp$
distr:$x \equiv x=\langle \phi,y\rangle \rightarrow \phi$;
$x=\langle\langle y_1, \ldots, y_n\rangle,z\rangle \rightarrow \langle\langle y_1,z\rangle, \ldots, \langle y_n,z\rangle\rangle$; $\perp$

Length
length:$x \equiv x=\langle x_1, \ldots, x_n\rangle \rightarrow n$; $x=\phi \rightarrow 0$; $\perp$

Add, subtract, multiply, and divide
$+:x \equiv x=\langle y,z\rangle$ & y,z are numbers $\rightarrow y+z$; $\perp$
$-:x \equiv x=\langle y,z\rangle$ & y,z are numbers $\rightarrow y-z$; $\perp$
$\times:x \equiv x=\langle y,z\rangle$ & y,z are numbers $\rightarrow y \times z$; $\perp$
$\div:x \equiv x=\langle y,z\rangle$ & y,z are numbers $\rightarrow y \div z$; $\perp$
(where $y \div 0 = \perp$)

Transpose
trans:$x \equiv x=\langle \phi, \ldots, \phi\rangle \rightarrow \phi$;
$x=\langle x_1, \ldots, x_n\rangle \rightarrow \langle y_1, \ldots, y_m\rangle$; $\perp$
where
$x_i=\langle x_{il}, \ldots, x_{im}\rangle$ and
$y_j=\langle x_{1j}, \ldots, x_{nj}\rangle$, $1 \leq i \leq n$, $1 \leq j \leq m$.

And, or, not
and:$x \equiv x=\langle T,T\rangle \rightarrow T$;
$x=\langle T,F\rangle \vee x=\langle F,T\rangle \vee x=\langle F,F\rangle \rightarrow F$; $\perp$
etc.

Append left; append right
apndl:$x \equiv x=\langle y,\phi\rangle \rightarrow \langle y\rangle$;
$x=\langle y,\langle z_1, \ldots, z_n\rangle\rangle \rightarrow \langle y,z_1, \ldots, z_n\rangle$; $\perp$
apndr:$x \equiv x=\langle \phi,z\rangle \rightarrow \langle z\rangle$;
$x=\langle\langle y_1, \ldots, y_n\rangle,z\rangle \rightarrow \langle y_1, \ldots, y_n,z\rangle$; $\perp$

Right selectors; Right tail
1r:$x \equiv x=\langle x_1, \ldots, x_n\rangle \rightarrow x_n$; $\perp$
2r:$x \equiv x=\langle x_1, \ldots, x_n\rangle$ & $n \geq 2 \rightarrow x_{n-1}$; $\perp$
etc.
tlr:$x \equiv x=\langle x_1\rangle \rightarrow \phi$;
$x=\langle x_1, \ldots, x_n\rangle$ & $n \geq 2 \rightarrow \langle x_1, \ldots, x_{n-1}\rangle$; $\perp$

Rotate left; rotate right
rotl:$x \equiv x=\phi \rightarrow \phi$; $x=\langle x_1\rangle \rightarrow \langle x_1\rangle$;
$x=\langle x_1, \ldots, x_n\rangle$ & $n \geq 2 \rightarrow \langle x_2, \ldots, x_n,x_1\rangle$; $\perp$
etc.

11.2.4 Functional forms, F. A functional form is an expression denoting a function; that function depends on the functions or objects which are the *parameters* of the expression. Thus, for example, if f and g are any functions, then $f \circ g$ is a functional form, the *composition* of f and g, f and g are its parameters, and it denotes the function such that, for any object x,

$(f \circ g):x = f:(g:x)$.

Some functional forms may have objects as parameters. For example, for any object x, $\bar{x}$ is a functional form, the *constant* function of x, so that for any object y

$\bar{x}:y \equiv y=\perp \rightarrow \perp$; x.

In particular, $\bar{\perp}$ is the everywhere-$\perp$ function.

Below we give some functional forms, many of which are used later in this paper. We use p, f, and g with and without subscripts to denote arbitrary functions; and x, $x_1, \ldots, x_n$, y as arbitrary objects. Square brackets [...] are used to indicate the functional form for *construction*, which denotes a function, whereas pointed brackets <...> denote sequences, which are objects. Parentheses are used both in particular functional forms (e.g., in *condition*) and generally to indicate grouping.

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

Construction
$[f_1, \ldots, f_n]:x \equiv \langle f_1:x, \ldots, f_n:x\rangle$ (Recall that since $\langle \ldots, \perp, \ldots\rangle = \perp$ and all functions are $\perp$-preserving, so is $[f_1, \ldots, f_n]$.)

Condition
$(p \rightarrow f; g):x \equiv (p:x)=T \rightarrow f:x; \quad (p:x)=F \rightarrow g:x; \perp$

Conditional *expressions* (used outside of FP systems to describe their functions) and the *functional form* condition are both identified by "$\rightarrow$". They are quite different although closely related, as shown in the above definitions. But no confusion should arise, since the elements of a conditional expression all denote values, whereas the elements of the functional form condition all denote functions, never values. When no ambiguity arises we omit right-associated parentheses; we write, for example, $p_1 \rightarrow f_1; p_2 \rightarrow f_2; g$ for $(p_1 \rightarrow f_1; (p_2 \rightarrow f_2; g))$.

Constant (Here x is an object parameter.)
$\bar{x}:y \equiv y=\perp \rightarrow \perp; x$

Insert
$/f:x \equiv x=\langle x_1\rangle \rightarrow x_1$; $x=\langle x_1, \ldots, x_n\rangle$ & $n \geq 2$
$\rightarrow f:\langle x_1, /f:\langle x_2, \ldots, x_n\rangle\rangle$; $\perp$

If f has a unique right unit $u_f \neq \perp$, where $f:\langle x,u_f\rangle \in \{x, \perp\}$ for all objects x, then the above definition is extended: $/f:\phi = u_f$. Thus

$/+:\langle 4,5,6\rangle = +:\langle 4, +:\langle 5, /+:\langle 6\rangle\rangle\rangle$
$= +:\langle 4, +:\langle 5,6\rangle\rangle = 15$
$/+:\phi = 0$

Apply to all
$\alpha f:x \equiv x=\phi \rightarrow \phi$;
$x=\langle x_1, \ldots, x_n\rangle \rightarrow \langle f:x_1, \ldots, f:x_n\rangle$; $\perp$

Binary to unary (x is an object parameter)
$(\text{bu}\, f\, x):y \equiv f:\langle x,y\rangle$

Thus

$(\text{bu} + 1):x = 1+x$

While
$(\text{while}\, p\, f):x \equiv p:x=T \rightarrow (\text{while}\, p\, f):(f:x);\ p:x=F \rightarrow x;\ \bot$

The above functional forms provide an effective method for computing the values of the functions they denote (if they terminate) provided one can effectively apply their function parameters.

11.2.5 Definitions. A *definition* in an FP system is an expression of the form

Def $l \equiv r$

where the left side l is an unused function symbol and the right side r is a functional form (which may depend on l). It expresses the fact that the symbol l is to denote the function given by r. Thus the definition **Def** last1 $\equiv$ 1∘reverse defines the function last1 that produces the last element of a sequence (or $\bot$). Similarly,

Def last $\equiv$ null∘tl $\rightarrow$ 1; last∘tl

defines the function last, which is the same as last1. Here in detail is how the definition would be used to compute last:$\langle 1,2\rangle$:

last:$\langle 1,2\rangle$ =	
definition of last	$\Rightarrow$ (null∘tl $\rightarrow$ 1; last∘tl):$\langle 1,2\rangle$
action of the form $(p \rightarrow f; g)$	$\Rightarrow$ last∘tl:$\langle 1,2\rangle$
	since null∘tl:$\langle 1,2\rangle$ = null:$\langle 2\rangle$ = F
action of the form $f \circ g$	$\Rightarrow$ last:(tl:$\langle 1,2\rangle$)
definition of primitive tail	$\Rightarrow$ last:$\langle 2\rangle$
definition of last	$\Rightarrow$ (null∘tl $\rightarrow$ 1; last∘tl):$\langle 2\rangle$
action of the form $(p \rightarrow f; g)$	$\Rightarrow$ 1:$\langle 2\rangle$
	since null∘tl:$\langle 2\rangle$ = null:ϕ = T
definition of selector 1	$\Rightarrow$ 2

The above illustrates the simple rule: to apply a defined symbol, replace it by the right side of its definition. Of course, some definitions may define nonterminating functions. A set D of definitions is *well formed* if no two left sides are the same.

11.2.6 Semantics. It can be seen from the above that an FP system is determined by choice of the following sets: (a) The set of atoms A (which determines the set of objects). (b) The set of primitive functions P. (c) The set of functional forms **F**. (d) A well formed set of definitions D. To understand the semantics of such a system one needs to know how to compute $f:x$ for any function f and any object x of the system. There are exactly four possibilities for f:
(1) f is a primitive function;
(2) f is a functional form;
(3) there is one definition in D, **Def** $f \equiv r$; and
(4) none of the above.
If f is a primitive function, then one has its description and knows how to apply it. If f is a functional form, then the description of the form tells how to compute $f:x$ in terms of the parameters of the form, which can be done by further use of these rules. If f is defined, **Def** $f \equiv r$, as in (3), then to find $f:x$ one computes $r:x$, which can be done by further use of these rules. If none of these, then $f:x \equiv \bot$. Of course, the use of these rules may not terminate for some f and some x, in which case we assign the value $f:x \equiv \bot$.

11.3 Examples of Functional Programs

The following examples illustrate the functional programming style. Since this style is unfamiliar to most readers, it may cause confusion at first; the important point to remember is that no part of a function definition is a result itself. Instead, each part is a *function* that must be applied to an argument to obtain a result.

11.3.1 Factorial.

Def ! $\equiv$ eq0 $\rightarrow \bar{1}$; $\times$∘[id, !∘sub1]

where

Def eq0 $\equiv$ eq∘[id, $\bar{0}$]
Def sub1 $\equiv -\circ$[id, $\bar{1}$]

Here are some of the intermediate expressions an FP system would obtain in evaluating !:2:

$!:2 \Rightarrow (\text{eq0} \rightarrow \bar{1};\ \times\circ[\text{id}, !\circ\text{sub1}]):2$
$\Rightarrow \times\circ[\text{id}, !\circ\text{sub1}]:2$
$\Rightarrow \times:\langle \text{id}:2, !\circ\text{sub1}:2\rangle \Rightarrow \times:\langle 2, !:1\rangle$
$\Rightarrow \times:\langle 2, \times:\langle 1, !:0\rangle\rangle$
$\Rightarrow \times:\langle 2, \times:\langle 1,\bar{1}:0\rangle\rangle \Rightarrow \times:\langle 2, \times:\langle 1,1\rangle\rangle$
$\Rightarrow \times:\langle 2,1\rangle \Rightarrow 2.$

In Section 12 we shall see how theorems of the algebra of FP programs can be used to prove that ! is the factorial function.

11.3.2 Inner product. We have seen earlier how this definition works.

Def IP $\equiv (/+)\circ(\alpha\times)\circ$trans

11.3.3 Matrix multiply. This matrix multiplication program yields the product of any pair $\langle m,n\rangle$ of conformable matrices, where each matrix m is represented as the sequence of its rows:

$m = \langle m_1, \ldots, m_r\rangle$
where $m_i = \langle m_{i1}, \ldots, m_{is}\rangle$ for i = 1, ... , r.
Def MM $\equiv (\alpha\alpha\text{IP})\circ(\alpha\text{distl})\circ\text{distr}\circ[1, \text{trans}\circ 2]$

The program MM has four steps, reading from right to left; each is applied in turn, beginning with [1, trans∘2], to the result of its predecessor. If the argument is $\langle m,n\rangle$, then the first step yields $\langle m,n'\rangle$ where n' = trans:n. The second step yields $\langle\langle m_1,n'\rangle, \ldots, \langle m_r,n'\rangle\rangle$, where the m_i are the rows of m. The third step, αdistl, yields

$\langle \text{distl}:\langle m_1,n'\rangle, \ldots, \text{distl}:\langle m_r,n'\rangle\rangle = \langle p_1, \ldots, p_r\rangle$

where

p_i = distl:<m_i,n'> = <<m_i,n_1'>, ... , <m_i,n_s'>>
for i = 1, ... , r

and n_j' is the jth column of n (the jth row of n'). Thus p_i, a sequence of row and column pairs, corresponds to the i-th product row. The operator $\alpha\alpha$IP, or $\alpha(\alpha$IP), causes αIP to be applied to each p_i, which in turn causes IP to be applied to each row and column pair in each p_i. The result of the last step is therefore the sequence of rows comprising the product matrix. If either matrix is not rectangular, or if the length of a row of m differs from that of a column of n, or if any element of m or n is not a number, the result is $\perp$.

This program MM does not name its arguments or any intermediate results; contains no variables, no loops, no control statements nor procedure declarations; has no initialization instructions; is not word-at-a-time in nature; is hierarchically constructed from simpler components; uses generally applicable housekeeping forms and operators (e.g., αf, distl, distr, trans); is perfectly general; yields $\perp$ whenever its argument is inappropriate in any way; does not constrain the order of evaluation unnecessarily (all applications of IP to row and column pairs can be done in parallel or in any order); and, using algebraic laws (see below), can be transformed into more "efficient" or into more "explanatory" programs (e.g., one that is recursively defined). None of these properties hold for the typical von Neumann matrix multiplication program.

Although it has an unfamiliar and hence puzzling form, the program MM describes the essential operations of matrix multiplication without overdetermining the process or obscuring parts of it, as most programs do; hence many straightforward programs for the operation can be obtained from it by formal transformations. It is an inherently inefficient program for von Neumann computers (with regard to the use of space), but efficient ones can be derived from it and realizations of FP systems can be imagined that could execute MM without the prodigal use of space it implies. Efficiency questions are beyond the scope of this paper; let me suggest only that since the language is so simple and does not dictate any binding of lambda-type variables to data, there may be better opportunities for the system to do some kind of "lazy" evaluation [9, 10] and to control data management more efficiently than is possible in lambda-calculus based systems.

11.4 Remarks About FP Systems

11.4.1 FP systems as programming languages. FP systems are so minimal that some readers may find it difficult to view them as programming languages. Viewed as such, a function f is a program, an object x is the contents of the store, and $f:x$ is the contents of the store after program f is activated with x in the store. The set of definitions is the program library. The primitive functions and the functional forms provided by the system are the basic statements of a particular programming language. Thus, depending on the choice of primitive functions and functional forms, the FP framework provides for a large class of languages with various styles and capabilities. The algebra of programs associated with each of these depends on its particular set of functional forms. The primitive functions, functional forms, and programs given in this paper comprise an effort to develop just one of these possible styles.

11.4.2 Limitations of FP systems. FP systems have a number of limitations. For example, a given FP system is a fixed language; it is not history sensitive: no program can alter the library of programs. It can treat input and output only in the sense that x is an input and $f:x$ is the output. If the set of primitive functions and functional forms is weak, it may not be able to express every computable function.

An FP system cannot compute a program since function expressions are not objects. Nor can one define new functional forms within an FP system. (Both of these limitations are removed in formal functional programming (FFP) systems in which objects "represent" functions.) Thus no FP system can have a function, apply, such that

apply:<x,y> $\equiv$ x:y

because, on the left, x is an object, and, on the right, x is a function. (Note that we have been careful to keep the set of function symbols and the set of objects distinct: thus 1 is a function symbol, and 1 is an object.)

The primary limitation of FP systems is that they are not history sensitive. Therefore they must be extended somehow before they can become practically useful. For discussion of such extensions, see the sections on FFP and AST systems (Sections 13 and 14).

11.4.3 Expressive power of FP systems. Suppose two FP systems, FP_1 and FP_2, both have the same set of objects and the same set of primitive functions, but the set of functional forms of FP_1 properly includes that of FP_2. Suppose also that both systems can express all computable functions on objects. Nevertheless, we can say that FP_1 is more expressive than FP_2, since every function expression in FP_2 can be duplicated in FP_1, but by using a functional form not belonging to FP_2, FP_1 can express some functions more directly and easily than FP_2.

I believe the above observation could be developed into a theory of the expressive power of languages in which a language A would be *more expressive* than language B under the following roughly stated conditions. First, form all possible functions of all types in A by applying all existing functions to objects and to each other in all possible ways until no new function of any type can be formed. (The set of objects is a type; the set of continuous functions [T→U] from type T to type U is a type. If $f\in$[T→U] and $t\in$T, then ft in U can be formed by applying f to t.) Do the same in language B. Next, compare each type in A to the corresponding type in B. If, for every type, A's type includes B's corresponding

type, then A is more expressive than B (or equally expressive). If some type of A's functions is incomparable to B's, then A and B are not comparable in expressive power.

11.4.4 Advantages of FP systems. The main reason FP systems are considerably simpler than either conventional languages or lambda-calculus-based languages is that they use only the most elementary fixed naming system (naming a function in a definition) with a simple fixed rule of substituting a function for its name. Thus they avoid the complexities both of the naming systems of conventional languages and of the substitution rules of the lambda calculus. FP systems permit the definition of different naming systems (see Sections 13.3.4 and 14.7) for various purposes. These need not be complex, since many programs can do without them completely. Most importantly, they treat names as functions that can be combined with other functions without special treatment.

FP systems offer an escape from conventional word-at-a-time programming to a degree greater even than APL [12] (the most successful attack on the problem to date within the von Neumann framework) because they provide a more powerful set of functional forms within a unified world of expressions. They offer the opportunity to develop higher level techniques for thinking about, manipulating, and writing programs.

12. The Algebra of Programs for FP Systems

12.1 Introduction

The algebra of the programs described below is the work of an amateur in algebra, and I want to show that it is a game amateurs can profitably play and enjoy, a game that does not require a deep understanding of logic and mathematics. In spite of its simplicity, it can help one to understand and prove things about programs in a systematic, rather mechanical way.

So far, proving a program correct requires knowledge of some moderately heavy topics in mathematics and logic: properties of complete partially ordered sets, continuous functions, least fixed points of functionals, the first-order predicate calculus, predicate transformers, weakest preconditions, to mention a few topics in a few approaches to proving programs correct. These topics have been very useful for professionals who make it their business to devise proof techniques; they have published a lot of beautiful work on this subject, starting with the work of McCarthy and Floyd, and, more recently, that of Burstall, Dijkstra, Manna and his associates, Milner, Morris, Reynolds, and many others. Much of this work is based on the foundations laid down by Dana Scott (denotational semantics) and C. A. R. Hoare (axiomatic semantics). But its theoretical level places it beyond the scope of most amateurs who work outside of this specialized field.

If the average programmer is to prove his programs correct, he will need much simpler techniques than those the professionals have so far put forward. The algebra of programs below may be one starting point for such a proof discipline and, coupled with current work on algebraic manipulation, it may also help provide a basis for automating some of that discipline.

One advantage of this algebra over other proof techniques is that the programmer can use his programming language as the language for deriving proofs, rather than having to state proofs in a separate logical system that merely talks *about* his programs.

At the heart of the algebra of programs are laws and theorems that state that one function expression is the same as another. Thus the law $[f,g]\circ h \equiv [f\circ h, g\circ h]$ says that the construction of f and g (composed with h) is the same function as the construction of (f composed with h) and (g composed with h) no matter what the functions f, g, and h are. Such laws are easy to understand, easy to justify, and easy and powerful to use. However, we also wish to use such laws to solve equations in which an "unknown" function appears on both sides of the equation. The problem is that if f satisfies some such equation, it will often happen that some extension f' of f will also satisfy the same equation. Thus, to give a unique meaning to solutions of such equations, we shall require a foundation for the algebra of programs (which uses Scott's notion of least fixed points of continuous functionals) to assure us that solutions obtained by algebraic manipulation are indeed least, and hence unique, solutions.

Our goal is to develop a foundation for the algebra of programs that disposes of the theoretical issues, so that a programmer can use simple algebraic laws and one or two theorems from the foundations to solve problems and create proofs in the same mechanical style we use to solve high-school algebra problems, and so that he can do so without knowing anything about least fixed points or predicate transformers.

One particular foundational problem arises: given equations of the form

$$f \equiv p_0 \rightarrow q_0; \ldots ; p_i \rightarrow q_i; E_i(f), \quad (1)$$

where the p_i's and q_i's are functions not involving f and $E_i(f)$ is a function expression involving f, the laws of the algebra will often permit the formal "extension" of this equation by one more "clause" by deriving

$$E_i(f) \equiv p_{i+1} \rightarrow q_{i+1}; E_{i+1}(f) \quad (2)$$

which, by replacing $E_i(f)$ in (1) by the right side of (2), yields

$$f \equiv p_0 \rightarrow q_0; \ldots ; p_{i+1} \rightarrow q_{i+1}; E_{i+1}(f). \quad (3)$$

This formal extension may go on without limit. One question the foundations must then answer is: when can the least f satisfying (1) be represented by the infinite expansion

$$f \equiv p_0 \rightarrow q_0; \ldots ; p_n \rightarrow q_n; \ldots \quad (4)$$

in which the final clause involving f has been dropped,

so that we now have a solution whose right side is free of f's? Such solutions are helpful in two ways: first, they give proofs of "termination" in the sense that (4) means that $f:x$ is defined if and only if there is an n such that, for every i less than n, $p_i:x = F$ and $p_n:x = T$ and $q_n:x$ is defined. Second, (4) gives a case-by-case description of f that can often clarify its behavior.

The foundations for the algebra given in a subsequent section are a modest start toward the goal stated above. For a limited class of equations its "linear expansion theorem" gives a useful answer as to when one can go from indefinitely extendable equations like (1) to infinite expansions like (4). For a larger class of equations, a more general "expansion theorem" gives a less helpful answer to similar questions. Hopefully, more powerful theorems covering additional classes of equations can be found. But for the present, one need only know the conclusions of these two simple foundational theorems in order to follow the theorems and examples appearing in this section.

The results of the foundations subsection are summarized in a separate, earlier subsection titled "expansion theorems," without reference to fixed point concepts. The foundations subsection itself is placed later where it can be skipped by readers who do not want to go into that subject.

12.2 Some Laws of the Algebra of Programs

In the algebra of programs for an FP system variables range over the set of functions of the system. The "operations" of the algebra are the functional forms of the system. Thus, for example, $[f,g]\circ h$ is an expression of the algebra for the FP system described above, in which f, g, and h are variables denoting arbitrary functions of that system. And

$$[f,g]\circ h \equiv [f\circ h, g\circ h]$$

is a law of the algebra which says that, whatever functions one chooses for f, g, and h, the function on the left is the same as that on the right. Thus this algebraic law is merely a restatement of the following proposition about any FP system that includes the functional forms $[f,g]$ and $f\circ g$:

PROPOSITION: For all functions f, g, and h and all objects x, $([f,g]\circ h):x \equiv [f\circ h, g\circ h]:x$.

PROOF:

$$\begin{aligned}([f,g]\circ h):x &= [f,g]:(h:x) \\ &\quad \text{by definition of composition} \\ &= \langle f:(h:x), g:(h:x)\rangle \\ &\quad \text{by definition of construction} \\ &= \langle (f\circ h):x, (g\circ h):x\rangle \\ &\quad \text{by definition of composition} \\ &= [f\circ h, g\circ h]:x \\ &\quad \text{by definition of construction} \quad \square\end{aligned}$$

Some laws have a domain smaller than the domain of all objects. Thus $1\circ[f,g] \equiv f$ does not hold for objects x such that $g:x = \bot$. We write

$$\text{defined}\circ g \longrightarrow 1\circ[f,g] \equiv f$$

to indicate that the law (or theorem) on the right holds within the domain of objects x for which $\text{defined}\circ g:x = T$. Where

Def $\text{defined} \equiv \bar{T}$

i.e. $\text{defined}:x \equiv x=\bot \rightarrow \bot;\ T$. In general we shall write a *qualified functional equation*:

$$p \longrightarrow f \equiv g$$

to mean that, for any object x, whenever $p:x = T$, then $f:x = g:x$.

Ordinary algebra concerns itself with two operations, addition and multiplication; it needs few laws. The algebra of programs is concerned with more operations (functional forms) and therefore needs more laws.

Each of the following laws requires a corresponding proposition to validate it. The interested reader will find most proofs of such propositions easy (two are given below). We first define the usual ordering on functions and equivalence in terms of this ordering:

DEFINITION $f\leq g$ iff for all objects x, either $f:x = \bot$, or $f:x = g:x$.

DEFINITION $f \equiv g$ iff $f\leq g$ and $g\leq f$.

It is easy to verify that $\leq$ is a partial ordering, that $f\leq g$ means g is an extension of f, and that $f \equiv g$ iff $f:x = g:x$ for all objects x. We now give a list of algebraic laws organized by the two principal functional forms involved.

I Composition and construction

I.1 $[f_1, \ldots, f_n]\circ g \equiv [f_1\circ g, \ldots, f_n\circ g]$

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

I.3 $/f\circ[g_1, \ldots, g_n]$
$\equiv f\circ[g_1, /f\circ[g_2, \ldots, g_n]]$ when $n\geq 2$
$\equiv f\circ[g_1, f\circ[g_2, \ldots, f\circ[g_{n-1}, g_n]\ldots]]$
$/f\circ[g] \equiv g$

I.4 $f\circ[\bar{x},g] \equiv (\text{bu}\, f\, x)\circ g$

I.5 $1\circ[f_1, \ldots, f_n] \leq f_1$
$s\circ[f_1, \ldots, f_s, \ldots, f_n] \leq f_s$ for any selector s, $s\leq n$
$\text{defined}\circ f_i$ (for all $i\neq s$, $1\leq i\leq n$) $\longrightarrow$
$s\circ[f_1, \ldots, f_n] \equiv f_s$

I.5.1 $[f_1\circ 1, \ldots, f_n\circ n]\circ[g_1, \ldots, g_n] \equiv [f_1\circ g_1, \ldots, f_n\circ g_n]$

I.6 $\text{tl}\circ[f_1] \leq \bar{\phi}$ and
$\text{tl}\circ[f_1, \ldots, f_n] \leq [f_2, \ldots, f_n]$ for $n\geq 2$
$\text{defined}\circ f_1 \longrightarrow \text{tl}\circ[f_1] \equiv \bar{\phi}$
and $\text{tl}\circ[f_1, \ldots, f_n] \equiv [f_2, \ldots, f_n]$ for $n\geq 2$

I.7 $\text{distl}\circ[f, [g_1, \ldots, g_n]] \equiv [[f,g_1], \ldots, [f,g_n]]$
$\text{defined}\circ f \longrightarrow \text{distl}\circ[f,\bar{\phi}] \equiv \bar{\phi}$
The analogous law holds for distr.

I.8 $\text{apndl}\circ[f, [g_1, \ldots, g_n]] \equiv [f,g_1, \ldots, g_n]$
$\text{null}\circ g \longrightarrow \text{apndl}\circ[f,g] \equiv [f]$

And so on for apndr, reverse, rotl, etc.

I.9 $[\ldots, \bar{\bot}, \ldots] \equiv \bar{\bot}$

I.10 $\text{apndl}\circ[f\circ g, \alpha f\circ h] \equiv \alpha f\circ\text{apndl}\circ[g,h]$

I.11 $\text{pair} \;\&\; \text{not}\circ\text{null}\circ 1 \longrightarrow$
$\text{apndl}\circ[[1\circ 1,2], \text{distr}\circ[\text{tl}\circ 1,2]] \equiv \text{distr}$

Where $f\&g \equiv \text{and}\circ[f,g]$;
pair $\equiv$ atom $\rightarrow \bar{F}$; eq$\circ$[length,$\bar{2}$]

II Composition and condition (right associated parentheses omitted) (Law II.2 is noted in Manna et al. [16], p. 493.)

II.1 $(p \rightarrow f; g)\circ h \equiv p\circ h \rightarrow f\circ h; g\circ h$

II.2 $h\circ(p \rightarrow f; g) \equiv p \rightarrow h\circ f; h\circ g$

II.3 $\text{or}\circ[q,\text{not}\circ q] \rightarrow\rightarrow \text{and}\circ[p,q] \rightarrow f;$
$\text{and}\circ[p,\text{not}\circ q] \rightarrow g;\ h \equiv p \rightarrow (q \rightarrow f; g); h$

II.3.1 $p \rightarrow (p \rightarrow f; g); h \equiv p \rightarrow f; h$

III Composition and miscellaneous

III.1 $\bar{x}\circ f \leq \bar{x}$
$\text{defined}\circ f \rightarrow\rightarrow \bar{x}\circ f \equiv \bar{x}$

III.1.1 $\bar{\perp}\circ f \equiv f\circ\bar{\perp} \equiv \bar{\perp}$

III.2 $f\circ\text{id} \equiv \text{id}\circ f \equiv f$

III.3 pair $\rightarrow\rightarrow$ 1$\circ$distr $\equiv$ [1$\circ$1,2] also:
pair $\rightarrow\rightarrow$ 1$\circ$tl $\equiv$ 2 etc.

III.4 $\alpha(f\circ g) \equiv \alpha f \circ \alpha g$

III.5 $\text{null}\circ g \rightarrow\rightarrow \alpha f\circ g \equiv \bar{\phi}$

IV Condition and construction

IV.1 $[f_1, \ldots, (p \rightarrow g; h), \ldots, f_n]$
$\equiv p \rightarrow [f_1, \ldots, g, \ldots, f_n]; [f_1, \ldots, h, \ldots, f_n]$

IV.1.1 $[f_1, \ldots, (p_1 \rightarrow g_1; \ldots; p_n \rightarrow g_n; h), \ldots, f_m]$
$\equiv p_1 \rightarrow [f_1, \ldots, g_1, \ldots, f_m];$
$\ldots; p_n \rightarrow [f_1, \ldots, g_n, \ldots, f_m]; [f_1, \ldots, h, \ldots, f_m]$

This concludes the present list of algebraic laws; it is by no means exhaustive, there are many others.

Proof of two laws

We give the proofs of validating propositions for laws I.10 and I.11, which are slightly more involved than most of the others.

PROPOSITION 1

apndl $\circ$ $[f\circ g, \alpha f\circ h] \equiv \alpha f \circ$ apndl $\circ$ $[g,h]$

PROOF. We show that, for every object x, both of the above functions yield the same result.

CASE 1. $h{:}x$ is neither a sequence nor ϕ.
Then both sides yield $\perp$ when applied to x.

CASE 2. $h{:}x = \phi$. Then
apndl$\circ$$[f\circ g, \alpha f\circ h]$: x
$=$ apndl: $\langle f\circ g{:}x, \phi\rangle = \langle f{:}(g{:}x)\rangle$
$\alpha f\circ$apndl$\circ[g,h]$: x
$= \alpha f\circ$apndl: $\langle g{:}x, \phi\rangle = \alpha f{:}\langle g{:}x\rangle$
$= \langle f{:}(g{:}x)\rangle$

CASE 3. $h{:}x = \langle y_1, \ldots, y_n\rangle$. Then

apndl$\circ$$[f\circ g, \alpha f\circ h]$: x
$=$ apndl: $\langle f\circ g{:}x, \alpha f{:}\langle y_1, \ldots, y_n\rangle\rangle$
$= \langle f{:}(g{:}x), f{:}y_1, \ldots, f{:}y_n\rangle$
$\alpha f\circ$apndl$\circ[g,h]$: x
$= \alpha f\circ$apndl: $\langle g{:}x, \langle y_1, \ldots, y_n\rangle\rangle$
$= \alpha f{:}\langle g{:}x, y_1, \ldots, y_n\rangle$
$= \langle f{:}(g{:}x), f{:}y_1, \ldots, f{:}y_n\rangle$ □

PROPOSITION 2

Pair & not$\circ$null$\circ$1 $\rightarrow\rightarrow$
apndl$\circ$[[1^2, 2], distr$\circ$[tl$\circ$1, 2]] $\equiv$ distr

where f & g is the function: and$\circ[f, g]$, and $f^2 \equiv f\circ f$.

PROOF. We show that both sides produce the same result when applied to any pair $\langle x,y\rangle$, where $x \neq \phi$, as per the stated qualification.

CASE 1. x is an atom or $\perp$. Then distr: $\langle x,y\rangle = \perp$, since $x \neq \phi$. The left side also yields $\perp$ when applied to $\langle x,y\rangle$, since tl$\circ$1:$\langle$x,y$\rangle$ $= \perp$ and all functions are $\perp$-preserving.

CASE 2. $x = \langle x_1, \ldots, x_n\rangle$. Then

apndl$\circ$[[1^2, 2], distr$\circ$[tl$\circ$1, 2]]:$\langle x, y\rangle$
$=$ apndl: $\langle\langle$1:x, $y\rangle$, distr: $\langle$tl:x, $y\rangle\rangle$
$=$ apndl: $\langle\langle x_1,y\rangle, \phi\rangle = \langle\langle x_1,y\rangle\rangle$ if tl:$x = \phi$
$=$ apndl: $\langle\langle x_1,y\rangle, \langle\langle x_2,y\rangle, \ldots, \langle x_n,y\rangle\rangle\rangle$
if tl:$x \neq \phi$
$= \langle\langle x_1,y\rangle, \ldots, \langle x_n,y\rangle\rangle$
$=$ distr: $\langle x,y\rangle$ □

12.3 Example: Equivalence of Two Matrix Multiplication Programs

We have seen earlier the matrix multiplication program:

Def MM $\equiv \alpha\alpha$IP $\circ$ αdistl $\circ$ distr $\circ$ [1, trans$\circ$2].

We shall now show that its initial segment, MM′, where

Def MM′ $\equiv \alpha\alpha$IP $\circ$ αdistl $\circ$ distr,

can be defined recursively. (MM′ "multiplies" a pair of matrices after the second matrix has been transposed. Note that MM′, unlike MM, gives $\perp$ for all arguments that are not pairs.) That is, we shall show that MM′ satisfies the following equation which recursively defines the same function (on pairs):

$f \equiv \text{null}\circ 1 \rightarrow \bar{\phi}$; apndl$\circ[\alpha$IP$\circ$distl$\circ$[1$\circ$1, 2], $f\circ$[tl$\circ$1, 2]].

Our proof will take the form of showing that the following function, R,

Def R $\equiv$ null$\circ$1 $\rightarrow \bar{\phi}$;
apndl$\circ[\alpha$IP$\circ$distl$\circ$[1$\circ$1, 2], MM′$\circ$[tl$\circ$1, 2]]

is, for all pairs $\langle x,y\rangle$, the same function as MM′. R "multiplies" two matrices, when the first has more than zero rows, by computing the first row of the "product" (with αIP$\circ$distl$\circ$[1$\circ$1, 2]) and adjoining it to the "product" of the tail of the first matrix and the second matrix. Thus the theorem we want is

pair $\rightarrow\rightarrow$ MM′ $\equiv$ R,

from which the following is immediate:

MM $\equiv$ MM′ $\circ$ [1, trans$\circ$2] $\equiv$ R $\circ$ [1, trans$\circ$2];

where

Def pair $\equiv$ atom $\rightarrow \bar{F}$; eq$\circ$[length, $\bar{2}$].

THEOREM: pair $\rightarrow\rightarrow$ MM′ $\equiv$ R
where

Def MM′ ≡ ααIP ∘ αdistl ∘ distr
Def R ≡ null∘1 → $\bar{\phi}$;
apndl∘[αIP∘distl∘[1^2, 2], MM′∘[tl∘1, 2]]

PROOF.
CASE 1. pair & null∘1 →→ MM′ ≡ R.

pair & null∘1 →→ R ≡ $\bar{\phi}$ by def of R
pair & null∘1 →→ MM′ ≡ $\bar{\phi}$
since distr: <ϕ,x> = ϕ by def of distr
and αf:ϕ = ϕ by def of Apply to all.
And so: ααIP ∘ αdistl ∘ distr: <ϕ,x> = ϕ.
Thus pair & null∘1 →→ MM′ ≡ R.
CASE 2. pair & not∘null∘1 →→ MM′ ≡ R.

pair & not∘null∘1 →→ R ≡ R′, (1)

by def of R and R′, where

Def R′ ≡ apndl∘[αIP∘distl∘[1^2, 2], MM′∘[tl∘1, 2]].

We note that

R′ ≡ apndl∘[f∘g, αf∘h]

where

f ≡ αIP∘distl
g ≡ [1^2, 2]
h ≡ distr∘[tl∘1, 2]
αf ≡ α(αIP∘distl) ≡ ααIP∘αdistl (by III.4). (2)

Thus, by I.10,

R′ ≡ αf∘apndl∘[g,h]. (3)

Now apndl∘[g,h] ≡ apndl∘[[1^2, 2], distr∘[tl∘1, 2]],
thus, by I.11,

pair & not∘null∘1 →→ apndl∘[g,h] ≡ distr. (4)

And so we have, by (1), (2), (3) and (4),

pair & not∘null∘1 →→ R ≡ R′
≡ αf∘distr ≡ ααIP∘αdistl∘distr ≡ MM′.

Case 1 and Case 2 together prove the theorem. □

12.4 Expansion Theorems

In the following subsections we shall be "solving" some simple equations (where by a "solution" we shall mean the "least" function which satisfies an equation). To do so we shall need the following notions and results drawn from the later subsection on foundations of the algebra, where their proofs appear.

12.4.1 Expansion. Suppose we have an equation of the form

f ≡ E(f) (E1)

where E(f) is an expression involving f. Suppose further that there is an infinite sequence of functions f_i for i = 0, 1, 2, ... , each having the following form:

f_0 ≡ $\bar{\perp}$
f_{i+1} ≡ $p_0 \to q_0$; ... ; $p_i \to q_i$; $\bar{\perp}$ (E2)

where the p_i's and q_i's are particular functions, so that E has the property:

E(f_i) ≡ f_{i+1} for i = 0, 1, 2, ... (E3)

Then we say that E is *expansive* and has the f_i's as *approximating functions.*

If E is expansive and has approximating functions as in (E2), and if f is the solution of (E1), then f can be written as the infinite expansion

f ≡ $p_0 \to q_0$; ... ; $p_n \to q_n$; ... (E4)

meaning that, for any x, f:$x \neq \perp$ iff there is an $n \geq 0$ such that (a) p_i:x = F for all i < n, and (b) p_n:x = T, and (c) q_n:$x \neq \perp$. When f:$x \neq \perp$, then f:x = q_n:x for this n. (The foregoing is a consequence of the "expansion theorem".)

12.4.2 Linear expansion. A more helpful tool for solving some equations applies when, for any function h,

E(h) ≡ $p_0 \to q_0$; E$_1$(h) (LE1)

and there exist p_i and q_i such that

E$_1$($p_i \to q_i$; h) ≡ $p_{i+1} \to q_{i+1}$; E$_1$(h)
for i = 0, 1, 2, ... (LE2)

and

E$_1$($\bar{\perp}$) ≡ $\bar{\perp}$. (LE3)

Under the above conditions E is said to be *linearly expansive.* If so, and f is the solution of

f ≡ E(f) (LE4)

then E is expansive and f can again be written as the infinite expansion

f ≡ $p_0 \to q_0$; ... ; $p_n \to q_n$; ... (LE5)

using the p_i's and q_i's generated by (LE1) and (LE2).

Although the p_i's and q_i's of (E4) or (LE5) are not unique for a given function, it may be possible to find additional constraints which would make them so, in which case the expansion (LE5) would comprise a canonical form for a function. Even without uniqueness these expansions often permit one to prove the equivalence of two different function expressions, and they often clarify a function's behavior.

12.5 A Recursion Theorem

Using three of the above laws and linear expansion, one can prove the following theorem of moderate generality that gives a clarifying expansion for many recursively defined functions.

RECURSION THEOREM: Let f be a solution of

f ≡ $p \to g$; $Q(f)$ (1)

where

Q(k) ≡ h∘[i, k∘j] for any function k (2)

and p, g, h, i, j are any given functions, then

$$f \equiv p \to g; p\circ j \to Q(g); \ldots ; p\circ j^n \to Q^n(g); \ldots \quad (3)$$

(where $Q^n(g)$ is $h\circ[i, Q^{n-1}(g)\circ j]$, and j^n is $j\circ j^{n-1}$ for $n \geq 2$) and

$$Q^n(g) \equiv /h \circ [i, i\circ j, \ldots , i\circ j^{n-1}, g\circ j^n]. \quad (4)$$

PROOF. We verify that $p \to g; Q(f)$ is linearly expansive. Let p_n, q_n and k be any functions. Then

$$\begin{aligned} &Q(p_n \to q_n; k) \\ &\quad \equiv h\circ[i, (p_n \to q_n; k)\circ j] \quad \text{by (2)} \\ &\quad \equiv h\circ[i, (p_n\circ j \to q_n\circ j; k\circ j)] \quad \text{by II.1} \\ &\quad \equiv h\circ(p_n\circ j \to [i, q_n\circ j]; [i, k\circ j]) \quad \text{by IV.1} \\ &\quad \equiv p_n\circ j \to h\circ[i, q_n\circ j]; h\circ[i, k\circ j] \quad \text{by II.2} \\ &\quad \equiv p_n\circ j \to Q(q_n); Q(k) \quad \text{by (2)} \end{aligned} \quad (5)$$

Thus if $p_0 \equiv p$ and $q_0 \equiv g$, then (5) gives $p_1 \equiv p\circ j$ and $q_1 = Q(g)$ and in general gives the following functions satisfying (LE2)

$$p_n \equiv p\circ j^n \quad \text{and} \quad q_n \equiv Q^n(g). \quad (6)$$

Finally,

$$\begin{aligned} Q(\bar{\perp}) &\equiv h\circ[i, \bar{\perp}\circ j] \\ &\equiv h\circ[i, \bar{\perp}] \quad \text{by III.1.1} \\ &\equiv h\circ\bar{\perp} \quad \text{by I.9} \\ &\equiv \bar{\perp} \quad \text{by III.1.1.} \end{aligned} \quad (7)$$

Thus (5) and (6) verify (LE2) and (7) verifies (LE3), with $E_1 \equiv Q$. If we let $E(f) \equiv p \to g; Q(f)$, then we have (LE1); thus E is linearly expansive. Since f is a solution of $f \equiv E(f)$, conclusion (3) follows from (6) and (LE5). Now

$$\begin{aligned} Q^n(g) &\equiv h\circ[i, Q^{n-1}(g)\circ j] \\ &\equiv h\circ[i, h\circ[i\circ j, \ldots , h\circ[i\circ j^{n-1}, g\circ j^n] \ldots]] \\ &\qquad \text{by I.1, repeatedly} \\ &\equiv /h\circ[i, i\circ j, \ldots , i\circ j^{n-1}, g\circ j^n] \quad \text{by I.3} \end{aligned} \quad (8)$$

Result (8) is the second conclusion (4). □

12.5.1 Example: correctness proof of a recursive factorial function. Let f be a solution of

$$f \equiv \text{eq0} \to \bar{1}; \times\circ[\text{id}, f\circ s]$$

where

Def $s \equiv -\circ[\text{id}, \bar{1}]$ (subtract 1).

Then f satisfies the hypothesis of the recursion theorem with $p \equiv \text{eq0}$, $g \equiv \bar{1}$, $h \equiv \times$, $i \equiv \text{id}$, and $j \equiv s$. Therefore

$$f \equiv \text{eq0} \to \bar{1}; \ldots ; \text{eq0}\circ s^n \to Q^n(\bar{1}); \ldots$$

and

$$Q^n(\bar{1}) \equiv /\times \circ [\text{id}, \text{id}\circ s, \ldots , \text{id}\circ s^{n-1}, \bar{1}\circ s^n].$$

Now $\text{id}\circ s^k \equiv s^k$ by III.2 and $\text{eq0}\circ s^n \rightarrow\!\!\!\!\rightarrow \bar{1}\circ s^n \equiv \bar{1}$ by III.1, since $\text{eq0}\circ s^n{:}x$ implies $\text{defined}\circ s^n{:}x$; and also $\text{eq0}\circ s^n{:}x \equiv \text{eq0}{:}(x - \text{n}) \equiv x{=}\text{n}$. Thus if $\text{eq0}\circ s^n{:}\ x = T$, then $x = \text{n}$ and

$$Q^n(\bar{1}){:}\ \text{n} = \text{n} \times (\text{n} - 1) \times \ldots \times (\text{n} - (\text{n} - 1)) \times (\bar{1}{:}\ (\text{n} - \text{n})) = \text{n}!.$$

Using these results for $\bar{1}\circ s^n$, $\text{eq0}\circ s^n$, and $Q^n(\bar{1})$ in the previous expansion for f, we obtain

$$f{:}x \equiv x{=}0 \to 1; \ldots ; x{=}\text{n} \to \text{n} \times (\text{n} - 1) \times \ldots \times 1 \times 1; \ldots$$

Thus we have proved that f terminates on precisely the set of nonnegative integers and that it is the factorial function thereon.

12.6 An Iteration Theorem

This is really a corollary of the recursion theorem. It gives a simple expansion for many iterative programs.

ITERATION THEOREM: Let f be the solution (i.e., the least solution) of

$$f \equiv p \to g; h\circ f\circ k$$

then

$$f \equiv p \to g; p\circ k \to h\circ g\circ k; \ldots ; p\circ k^n \to h^n\circ g\circ k^n; \ldots$$

PROOF. Let $h' \equiv h\circ 2$, $i' \equiv id$, $j' \equiv k$, then

$$f \equiv p \to g; h'\circ[i', f\circ j']$$

since $h\circ 2\circ[\text{id}, f\circ k] \equiv h\circ f\circ k$ by I.5 (id is defined except for $\perp$, and the equation holds for $\perp$). Thus the recursion theorem gives

$$f \equiv p \to g; \ldots ; p\circ k^n \to Q^n(g); \ldots$$

where

$$\begin{aligned} Q^n(g) &\equiv h\circ 2\circ[\text{id}, Q^{n-1}(g)\circ k] \\ &\equiv h\circ Q^{n-1}(g)\circ k \equiv h^n\circ g\circ k^n \end{aligned}$$

by I.5 □

12.6.1 Example: Correctness proof for an iterative factorial function. Let f be the solution of

$$f \equiv \text{eq0}\circ 1 \to 2; f\circ[s\circ 1, \times]$$

where **Def** $s \equiv -\circ[\text{id}, \bar{1}]$ (substract 1). We want to prove that $f{:}\langle x,1\rangle = x!$ iff x is a nonnegative integer. Let $p \equiv \text{eq0}\circ 1$, $g \equiv 2$, $h \equiv \text{id}$, $k \equiv [s\circ 1, \times]$. Then

$$f \equiv p \to g; h\circ f\circ k$$

and so

$$f \equiv p \to g; \ldots ; p\circ k^n \to g\circ k^n; \ldots \quad (1)$$

by the iteration theorem, since $h^n \equiv \text{id}$. We want to show that

$$\text{pair} \rightarrow\!\!\!\!\rightarrow k^n \equiv [a_n, b_n] \quad (2)$$

holds for every $n \geq 1$, where

$$a_n \equiv s^n\circ 1 \quad (3)$$

$$b_n \equiv /\times \circ [s^{n-1}\circ 1, \ldots , s\circ 1, 1, 2] \quad (4)$$

Now (2) holds for $n = 1$ by definition of k. We assume it holds for some $n \geq 1$ and prove it then holds for $n + 1$. Now

$$\text{pair} \rightarrow\!\!\!\!\rightarrow k^{n+1} \equiv k\circ k^n \equiv [s\circ 1, \times]\circ[a_n, b_n] \quad (5)$$

since (2) holds for n. And so

$$\text{pair} \rightarrow\rightarrow k^{n+1} \equiv [s\circ a_n,\ \times\circ[a_n, b_n]] \text{ by I.1 and I.5} \quad (6)$$

To pass from (5) to (6) we must check that whenever a_n or b_n yield $\perp$ in (5), so will the right side of (6). Now

$$s\circ a_n \equiv s^{n+1}\circ 1 \equiv a_{n+1} \quad (7)$$

$$\times\circ[a_n, b_n] \equiv /\times \circ [s^n\circ 1, s^{n-1}\circ 1, \ldots, s\circ 1, 1, 2] \equiv b_{n+1} \text{ by I.3.} \quad (8)$$

Combining (6), (7), and (8) gives

$$\text{pair} \rightarrow\rightarrow k^{n+1} \equiv [a_{n+1}, b_{n+1}]. \quad (9)$$

Thus (2) holds for n = 1 and holds for n + 1 whenever it holds for n, therefore, by induction, it holds for every $n \geq 1$. Now (2) gives, for pairs:

$$\text{defined}\circ k^n \rightarrow\rightarrow p\circ k^n \equiv \text{eq0}\circ 1\circ[a_n, b_n] \equiv \text{eq0}\circ a_n \equiv \text{eq0}\circ s^n\circ 1 \quad (10)$$

$$\text{defined}\circ k^n \rightarrow\rightarrow g\circ k^n \equiv 2\circ[a_n, b_n] \equiv /\times \circ [s^{n-1}\circ 1, \ldots, s\circ 1, 1, 2] \quad (11)$$

(both use I.5). Now (1) tells us that $f{:}\langle x,1\rangle$ is defined iff there is an n such that $p\circ k^i{:}\langle x,1\rangle = F$ for all $i < n$, and $p\circ k^n{:}\langle x,1\rangle = T$, that is, by (10), $\text{eq0}\circ s^n{:}x = T$, i.e., x=n; and $g\circ k^n{:}\langle x,1\rangle$ is defined, in which case, by (11),

$$f{:}\langle x,1\rangle = /\times{:}\langle 1, 2, \ldots, x-1, x, 1\rangle = \text{n!},$$

which is what we set out to prove.

12.6.2 Example: proof of equivalence of two iterative programs. In this example we want to prove that two iteratively defined programs, f and g, are the same function. Let f be the solution of

$$f \equiv p\circ 1 \rightarrow 2;\ h\circ f\circ[k\circ 1, 2]. \quad (1)$$

Let g be the solution of

$$g \equiv p\circ 1 \rightarrow 2;\ g\circ[k\circ 1, h\circ 2]. \quad (2)$$

Then, by the iteration theorem:

$$f \equiv p_0 \rightarrow q_0; \ldots; p_n \rightarrow q_n; \ldots \quad (3)$$

$$g \equiv p'_0 \rightarrow q'_0; \ldots; p'_n \rightarrow q'_n; \ldots \quad (4)$$

where (letting $r^0 \equiv \text{id}$ for any r), for n = 0, 1, ...

$$p_n \equiv p\circ 1\circ[k\circ 1, 2]^n \equiv p\circ 1\circ[k^n\circ 1, 2] \text{ by I.5.1} \quad (5)$$

$$q_n \equiv h^n\circ 2\circ[k\circ 1, 2]^n \equiv h^n\circ 2\circ[k^n\circ 1, 2] \text{ by I.5.1} \quad (6)$$

$$p'_n \equiv p\circ 1\circ[k\circ 1, h\circ 2]^n \equiv p\circ 1\circ[k^n\circ 1, h^n\circ 2] \text{ by I.5.1} \quad (7)$$

$$q'_n \equiv 2\circ[k\circ 1, h\circ 2]^n \equiv 2\circ[k^n\circ 1, h^n\circ 2] \text{ by I.5.1.} \quad (8)$$

Now, from the above, using I.5,

$$\text{defined}\circ 2 \rightarrow\rightarrow p_n \equiv p\circ k^n\circ 1 \quad (9)$$

$$\text{defined}\circ h^n\circ 2 \rightarrow\rightarrow p'_n \equiv p\circ k^n\circ 1 \quad (10)$$

$$\text{defined}\circ k^n\circ 1 \rightarrow\rightarrow q_n \equiv q'_n \equiv h^n\circ 2 \quad (11)$$

Thus

$$\text{defined}\circ h^n\circ 2 \rightarrow\rightarrow \text{defined}\circ 2 \equiv \bar{T} \quad (12)$$

$$\text{defined}\circ h^n\circ 2, \rightarrow\rightarrow p_n \equiv p'_n \quad (13)$$

and

$$f \equiv p_0 \rightarrow q_0; \ldots; p_n \rightarrow h^n\circ 2; \ldots \quad (14)$$

$$g \equiv p'_0 \rightarrow q'_0; \ldots; p'_n \rightarrow h^n\circ 2; \ldots \quad (15)$$

since p_n and p'_n provide the qualification needed for $q_n \equiv q'_n \equiv h^n\circ 2$.

Now suppose there is an x such that $f{:}x \not\equiv g{:}x$. Then there is an n such that $p_i{:}x = p'_i{:}x = F$ for $i < n$, and $p_n{:}x \not\equiv p'_n{:}x$. From (12) and (13) this can only happen when $h^n\circ 2{:}x = \perp$. But since h is $\perp$-preserving, $h^m\circ 2{:}x = \perp$ for all $m \geq n$. Hence $f{:}x = g{:}x = \perp$ by (14) and (15). This contradicts the assumption that there is an x for which $f{:}x \not\equiv g{:}x$. Hence $f \equiv g$.

This example (by J. H. Morris, Jr.) is treated more elegantly in [16] on p. 498. However, some may find that the above treatment is more constructive, leads one more mechanically to the key questions, and provides more insight into the behavior of the two functions.

12.7 Nonlinear Equations

The preceding examples have concerned "linear" equations (in which the "unknown" function does not have an argument involving itself). The question of the existence of simple expansions that "solve" "quadratic" and higher order equations remains open.

The earlier examples concerned solutions of $f \equiv \text{E}(f)$, where E is linearly expansive. The following example involves an $\text{E}(f)$ that is quadratic and expansive (but not linearly expansive).

12.7.1 Example: proof of idempotency ([16] p. 497). Let f be the solution of

$$f \equiv \text{E}(f) \equiv p \rightarrow \text{id};\ f^2\circ h. \quad (1)$$

We wish to prove that $f \equiv f^2$. We verify that E is expansive (Section 12.4.1) with the following approximating functions:

$$f_0 \equiv \bar{\perp} \quad (2a)$$

$$f_n \equiv p \rightarrow \text{id}; \ldots; \text{p}\circ\text{h}^{n-1} \rightarrow \text{h}^{n-1}; \bar{\perp} \quad \text{for } n > 0 \quad (2b)$$

First we note that $p \rightarrow\rightarrow f_n \equiv \text{id}$ and so

$$p\circ h^i \rightarrow\rightarrow f_n\circ h^i \equiv h^i. \quad (3)$$

$$\text{Now E}(f_0) \equiv p \rightarrow \text{id};\ \bar{\perp}^2\circ h \equiv f_1, \quad (4)$$

and

$$\begin{aligned}&\text{E}(f_n)\\ &\equiv p \rightarrow \text{id}; f_n\circ(p \rightarrow \text{id}; \ldots; p\circ h^{n-1} \rightarrow h^{n-1}; \bar{\perp})\circ h\\ &\equiv p \rightarrow \text{id}; f_n\circ(p\circ h \rightarrow h; \ldots; p\circ h^n \rightarrow h^n; \bar{\perp}\circ h)\\ &\equiv p \rightarrow \text{id}; p\circ h \rightarrow f_n\circ h; \ldots; p\circ h^n \rightarrow f_n\circ h^n; f_n\circ\bar{\perp}\\ &\equiv p \rightarrow \text{id}; p\circ h \rightarrow h; \ldots; p\circ h^n \rightarrow h^n; \bar{\perp} \text{ by (3)}\\ &\equiv f_{n+1}. \end{aligned} \quad (5)$$

Thus E is expansive by (4) and (5); so by (2) and Section 12.4.1 (E4)

$$f \equiv p \rightarrow \text{id}; \ldots; \text{p}\circ\text{h}^n \rightarrow \text{h}^n; \ldots. \quad (6)$$

But (6), by the iteration theorem, gives

$$f \equiv p \rightarrow \text{id}; f\circ h. \quad (7)$$

Now, if $p{:}x = T$, then $f{:}x = x = f^2{:}x$, by (1). If $p{:}x = F$, then

$$f{:}x = f^2\circ h{:}x \quad \text{by (1)}$$

$= f:(f \circ h:x) = f:(f:x)$ by (7)
$= f^2:x.$

If $p:x$ is neither T nor F, then $f:x = \bot = f^2:x$. Thus $f \equiv f^2$.

12.8 Foundations for the Algebra of Programs

Our purpose in this section is to establish the validity of the results stated in Section 12.4. Subsequent sections do not depend on this one, hence it can be skipped by readers who wish to do so. We use the standard concepts and results from [16], but the notation used for objects and functions, etc., will be that of this paper.

We take as the domain (and range) for all functions the set O of objects (which includes $\bot$) of a given FP system. We take F to be the set of functions, and **F** to be the set of functional forms of that FP system. We write $E(f)$ for any function expression involving functional forms, primitive and defined functions, and the function symbol f; and we regard E as a functional that maps a function f into the corresponding function $E(f)$. We assume that all $f \in F$ are $\bot$-preserving and that all functional forms in **F** correspond to continuous functionals in every variable (e.g., $[f, g]$ is continuous in both f and g). (All primitive functions of the FP system given earlier are $\bot$-preserving, and all its functional forms are continuous.)

DEFINITIONS. Let $E(f)$ be a function expression. Let

$f_0 \equiv \bar{\bot}$
$f_{i+1} \equiv p_0 \to q_0; \ldots; p_i \to q_i; \bar{\bot}$ for $i = 0, 1, \ldots$

where $p_i, q_i \in F$. Let E have the property that

$E(f_i) \equiv f_{i+1}$ for $i = 0, 1, \ldots$.

Then E is said to be *expansive* with the *approximating functions* f_i. We write

$f \equiv p_0 \to q_0; \ldots; p_n \to q_n; \ldots$

to mean that $f \equiv \lim_i\{f_i\}$, where the f_i have the form above. We call the right side an *infinite expansion* of f. We take $f:x$ to be defined iff there is an $n \geq 0$ such that (a) $p_i:x = F$ for all $i < n$, and (b) $p_n:x = T$, and (c) $q_n:x$ is defined, in which case $f:x = q_n:x$.

EXPANSION THEOREM: Let $E(f)$ be expansive with approximating functions as above. Let f be the least function satisfying

$f \equiv E(f)$.

Then

$f \equiv p_0 \to q_0; \ldots; p_n \to q_n; \ldots$

PROOF. Since E is the composition of continuous functionals (from **F**) involving only monotonic functions ($\bot$-preserving functions from F) as constant terms, E is continuous ([16] p. 493). Therefore its least fixed point f is $\lim_i\{E^i(\bar{\bot})\} \equiv \lim_i\{f_i\}$ ([16] p. 494), which by definition is the above infinite expansion for f. □

DEFINITION. Let $E(f)$ be a function expression satisfying the following:

$E(h) \equiv p_0 \to q_0; E_1(h)$ for all $h \in F$ (LE1)

where $p_i \in F$ and $q_i \in F$ exist such that

$E_1(p_i \to q_i; h) \equiv p_{i+1} \to q_{i+1}; E_1(h)$
for all $h \in F$ and $i = 0, 1, \ldots$ (LE2)

and

$E_1(\bar{\bot}) \equiv \bar{\bot}$. (LE3)

Then E is said to be *linearly expansive* with respect to these p_i's and q_i's.

LINEAR EXPANSION THEOREM: Let E be linearly expansive with respect to p_i and q_i, $i = 0, 1, \ldots$. Then E is expansive with approximating functions

$f_0 \equiv \bar{\bot}$ (1)
$f_{i+1} \equiv p_0 \to q_0; \ldots; p_i \to q_i; \bar{\bot}$. (2)

PROOF. We want to show that $E(f_i) \equiv f_{i+1}$ for any $i \geq 0$. Now

$E(f_0) \equiv p_0 \to q_0; E_1(\bar{\bot}) \equiv p_0 \to q_0; \bar{\bot} \equiv f_1$ (3)
by (LE1) (LE3) (1).

Let $i > 0$ be fixed and let

$f_i \equiv p_0 \to q_0; w_1$ (4a)
$w_1 \equiv p_1 \to q_1; w_2$ (4b)
etc.
$w_{i-1} \equiv p_{i-1} \to q_{i-1}; \bar{\bot}$. (4-)

Then, for this $i > 0$

$E(f_i) \equiv p_0 \to q_0; E_1(f_i)$ by (LE1)
$E_1(f_i) \equiv p_1 \to q_1; E_1(w_1)$ by (LE2) and (4a)
$E_1(w_1) \equiv p_2 \to q_2; E_1(w_2)$ by (LE2) and (4b)

etc.

$E_1(w_{i-1}) \equiv p_i \to q_i; E_1(\bar{\bot})$ by (LE2) and (4-)
$\equiv p_i \to q_i; \bar{\bot}$ by (LE3)

Combining the above gives

$E(f_i) \equiv f_{i+1}$ for arbitrary $i > 0$, by (2). (5)

By (3), (5) also holds for $i = 0$; thus it holds for all $i \geq 0$. Therefore E is expansive and has the required approximating functions. □

COROLLARY. If E is linearly expansive with respect to p_i and q_i, $i = 0, 1, \ldots$, and f is the least function satisfying

$f \equiv E(f)$ (LE4)

then

$f \equiv p_0 \to q_0; \ldots; p_n \to q_n; \ldots$. (LE5)

12.9 The Algebra of Programs for the Lambda Calculus and for Combinators

Because Church's lambda calculus [5] and the system of combinators developed by Schönfinkel and Curry [6]

are the primary mathematical systems for representing the notion of application of functions, and because they are more powerful than FP systems, it is natural to enquire what an algebra of programs based on those systems would look like.

The lambda calculus and combinator equivalents of FP composition, $f \circ g$, are

$$\lambda fgx.(f(gx)) \equiv \mathrm{B}$$

where B is a simple combinator defined by Curry. There is no direct equivalent for the FP object $<x,y>$ in the Church or Curry systems proper; however, following Landin [14] and Burge [4], one can use the primitive functions prefix, head, tail, null, and atomic to introduce the notion of list structures that correspond to FP sequences. Then, using FP notation for lists, the lambda calculus equivalent for construction is $\lambda fgx.<fx,gx>$. A combinatory equivalent is an expression involving prefix, the null list, and two or more basic combinators. It is so complex that I shall not attempt to give it.

If one uses the lambda calculus or combinatory expressions for the functional forms $f \circ g$ and $[f,g]$ to express the law I.1 in the FP algebra, $[f,g] \circ h \equiv [f \circ h, g \circ h]$, the result is an expression so complex that the sense of the law is obscured. The only way to make that sense clear in either system is to name the two functionals: composition $\equiv$ B, and construction $\equiv$ A, so that $\mathrm{B}fg \equiv f \circ g$, and $\mathrm{A}fg \equiv [f,g]$. Then I.1 becomes

$$\mathrm{B}(\mathrm{A}fg)h \equiv \mathrm{A}(\mathrm{B}fh)(\mathrm{B}gh),$$

which is still not as perspicuous as the FP law.

The point of the above is that if one wishes to state clear laws like those of the FP algebra in either Church's or Curry's system, one finds it necessary to select certain functionals (e.g., composition and construction) as the basic operations of the algebra and to either give them short names or, preferably, represent them by some special notation as in FP. If one does this and provides primitives, objects, lists, etc., the result is an FP-like system in which the usual lambda expressions or combinators do not appear. Even then these Church or Curry versions of FP systems, being less restricted, have some problems that FP systems do not have:

a) The Church and Curry versions accommodate functions of many types and can define functions that do not exist in FP systems. Thus, $\mathrm{B}f$ is a function that has no counterpart in FP systems. This added power carries with it problems of type compatibility. For example, in $f \circ g$, is the range of g included in the domain of f? In FP systems all functions have the same domain and range.

b) The semantics of Church's lambda calculus depends on substitution rules that are simply stated but whose implications are very difficult to fully comprehend. The true complexity of these rules is not widely recognized but is evidenced by the succession of able logicians who have published "proofs" of the Church-Rosser theorem that failed to account for one or another of these complexities. (The Church-Rosser theorem, or Scott's proof of the existence of a model [22], is required to show that the lambda calculus has a consistent semantics.) The definition of pure Lisp contained a related error for a considerable period (the "funarg" problem). Analogous problems attach to Curry's system as well.

In contrast, the formal (FFP) version of FP systems (described in the next section) has no variables and only an elementary substitution rule (a function for its name), and it can be shown to have a consistent semantics by a relatively simple fixed-point argument along the lines developed by Dana Scott and by Manna et al [16]. For such a proof see McJones [18].

12.10 Remarks

The algebra of programs outlined above needs much work to provide expansions for larger classes of equations and to extend its laws and theorems beyond the elementary ones given here. It would be interesting to explore the algebra for an FP-like system whose sequence constructor is not $\perp$-preserving (law I.5 is strengthened, but IV.1 is lost). Other interesting problems are: (a) Find rules that make expansions unique, giving canonical forms for functions; (b) find algorithms for expanding and analyzing the behavior of functions for various classes of arguments; and (c) explore ways of using the laws and theorems of the algebra as the basic rules either of a formal, preexecution "lazy evaluation" scheme [9, 10], or of one which operates during execution. Such schemes would, for example, make use of the law $1 \circ [f,g] \leq f$ to avoid evaluating $g{:}x$.

13. Formal Systems for Functional Programming (FFP Systems)

13.1 Introduction

As we have seen, an FP system has a set of functions that depends on its set of primitive functions, its set of functional forms, and its set of definitions. In particular, its set of functional forms is fixed once and for all, and this set determines the power of the system in a major way. For example, if its set of functional forms is empty, then its entire set of functions is just the set of primitive functions. In FFP systems one can create new functional forms. Functional forms are represented by object sequences; the first element of a sequence determines which form it represents, while the remaining elements are the parameters of the form.

The ability to define new functional forms in FFP systems is one consequence of the principal difference between them and FP systems: in FFP systems objects are used to "represent" functions in a systematic way. Otherwise FFP systems mirror FP systems closely. They are similar to, but simpler than, the Reduction (Red) languages of an earlier paper [2].

We shall first give the simple syntax of FFP systems, then discuss their semantics informally, giving examples, and finally give their formal semantics.

13.2 Syntax

We describe the set O of objects and the set E of expressions of an FFP system. These depend on the choice of some set A of *atoms*, which we take as given. We assume that T (true), F (false), ϕ (the empty sequence), and # (default) belong to A, as well as "numbers" of various kinds, etc.

1) Bottom, $\perp$, is an *object* but not an atom.
2) Every atom is an *object.*
3) Every object is an *expression.*
4) If $x_1, \ldots, x_n$ are objects [expressions], then $\langle x_1, \ldots, x_n \rangle$ is an *object* [resp., *expression*] called a *sequence* (of *length* n) for $n \geq 1$. The object [expression] x_i for $1 \leq i \leq n$, is the ith *element* of the sequence $\langle x_1, \ldots, x_i, \ldots, x_n \rangle$. ($\phi$ is both a sequence and an atom; its length is 0.)
5) If x and y are expressions, then $(x{:}y)$ is an *expression* called an *application.* x is its *operator* and y is its *operand.* Both are *elements* of the expression.
6) If $x = \langle x_1, \ldots, x_n \rangle$ and if one of the elements of x is $\perp$, then $x = \perp$. That is, $\langle \ldots, \perp, \ldots \rangle = \perp$.
7) All objects and expressions are formed by finite use of the above rules.

A *subexpression* of an expression x is either x itself or a subexpression of an element of x. An FFP object is an expression that has no application as a subexpression. Given the same set of atoms, FFP and FP objects are the same.

13.3 Informal Remarks About FFP Semantics

13.3.1 The meaning of expressions; the semantic function μ. Every FFP expression e has a *meaning*, μe, which is always an object; μe is found by repeatedly replacing each innermost application in e by its meaning. If this process is nonterminating, the meaning of e is $\perp$. The meaning of an innermost application $(x{:}y)$ (since it is innermost, x and y must be objects) is the result of applying the function *represented* by x to y, just as in FP systems, except that in FFP systems functions are represented by objects, rather than by function expressions, with atoms (instead of function symbols) representing primitive and defined functions, and with sequences representing the FP functions denoted by functional forms.

The association between objects and the functions they represent is given by the *representation function*, ρ, of the FFP system. (Both ρ and μ belong to the description of the system, not the system itself.) Thus if the atom $NULL$ represents the FP function null, then $\rho NULL$ = null and the meaning of $(NULL{:}A)$ is $\mu(NULL{:}A) = (\rho NULL){:}A = \text{null}{:}A = F$.

From here on, as above, we use the colon in two senses. When it is between two objects, as in $(NULL{:}A)$, it identifies an FFP application that denotes only itself; when it comes between a *function* and an object, as in $(\rho NULL){:}A$ or null:A, it identifies an FP-like application that denotes the *result* of applying the function to the object.

The fact that FFP operators are objects makes possible a function, apply, which is meaningless in FP systems:

$$\text{apply}{:}\langle x,y \rangle = (x{:}y).$$

The result of apply:$\langle x,y \rangle$, namely $(x{:}y)$, is meaningless in FP systems on two levels. First, $(x{:}y)$ is not itself an object; it illustrates another difference between FP and FFP systems: some FFP functions, like apply, map objects into expressions, not directly into objects as FP functions do. However, the *meaning* of apply:$\langle x,y \rangle$ *is* an object (see below). Second, $(x{:}y)$ could not be even an intermediate result in an FP system; it is meaningless in FP systems since x is an object, not a function and FP systems do not associate functions with objects. Now if $APPLY$ represents apply, then the meaning of $(APPLY{:}\langle NULL,A \rangle)$ is

$$\begin{aligned}\mu(APPLY{:}\langle NULL,A \rangle) &= \mu((\rho APPLY){:}\langle NULL,A \rangle) \\ &= \mu(\text{apply}{:}\langle NULL,A \rangle) \\ &= \mu(NULL{:}A) = \mu((\rho NULL){:}A) \\ &= \mu(\text{null}{:}A) = \mu F = F.\end{aligned}$$

The last step follows from the fact that every object is its own meaning. Since the meaning function μ eventually evaluates all applications, one can think of apply:$\langle NULL,A \rangle$ as yielding F even though the actual result is $(NULL{:}A)$.

13.3.2 How objects represent functions; the representation function ρ. As we have seen, some atoms (*primitive* atoms) will represent the primitive functions of the system. Other atoms can represent defined functions just as symbols can in FP systems. If an atom is neither primitive nor defined, it represents $\bar{\perp}$, the function which is $\perp$ everywhere.

Sequences also represent functions and are analogous to the functional forms of FP. The function represented by a sequence is given (recursively) by the following rule.

Metacomposition rule

$$(\rho\langle x_1, \ldots, x_n \rangle){:}y = (\rho x_1){:}\langle\langle x_1, \ldots, x_n \rangle, y \rangle,$$

where the x_i's and y are objects. Here ρx_1 determines what functional form $\langle x_1, \ldots, x_n \rangle$ represents, and $x_2, \ldots, x_n$ are the parameters of the form (in FFP, x_1 itself can also serve as a parameter). Thus, for example, let **Def** $\rho CONST \equiv 2 \circ 1$; then $\langle CONST,x \rangle$ in FFP represents the FP functional form $\bar{x}$, since, by the metacomposition rule, if $y \neq \perp$,

$$\begin{aligned}(\rho\langle CONST,x \rangle){:}y &= (\rho CONST){:}\langle\langle CONST,x \rangle,y \rangle \\ &= 2 \circ 1{:}\langle\langle CONST,x \rangle,y \rangle = x.\end{aligned}$$

Here we can see that the first, controlling, operator of a sequence or form, $CONST$ in this case, always has as its operand, after metacomposition, a pair whose first element is the sequence itself and whose second element is the original operand of the sequence, y in this case. The controlling operator can then rearrange and reapply the elements of the sequence and original operand in a great variety of ways. The significant point about metacom-

position is that it permits the definition of new functional forms, in effect, merely by defining new functions. It also permits one to write recursive functions without a definition.

We give one more example of a controlling function for a functional form: **Def** $\rho CONS \equiv \alpha$apply$\circ$tl$\circ$distr. This definition results in $<CONS,f_1, \ldots ,f_n>$—where the f_i are objects—representing the same function as $[\rho f_1, \ldots , \rho f_n]$. The following shows this.

$(\rho<CONS,f_1, \ldots ,f_n>):x$
$= (\rho CONS):<<CONS,f_1, \ldots ,f_n>,x>$
by metacomposition
$= \alpha$apply$\circ$tl$\circ$distr:$<<CONS,f_1, \ldots ,f_n>,x>$
by def of $\rho CONS$
$= \alpha$apply:$<<f_1,x>, \ldots , <f_n,x>>$
by def of tl and distr and $\circ$
$= <$apply:$<f_1,x>, \ldots ,$ apply:$<f_n,x>>$
by def of α
$= <(f_1:x), \ldots , (f_n:x)>$ by def of apply.

In evaluating the last expression, the meaning function μ will produce the meaning of each application, giving $\rho f_i:x$ as the ith element.

Usually, in describing the function represented by a sequence, we shall give its overall effect rather than show how its controlling operator achieves that effect. Thus we would simply write

$(\rho<CONS,f_1, \ldots ,f_n>):x = <(f_1:x), \ldots , (f_n:x)>$

instead of the more detailed account above.

We need a controlling operator, *COMP*, to give us sequences representing the functional form composition. We take $\rho COMP$ to be a primitive function such that, for all objects x,

$(\rho<COMP,f_1, \ldots ,f_n>):x$
$= (f_1:(f_2:(\ldots :(f_n:x)\ldots)))$ for $n \geq 1$.

(I am indebted to Paul McJones for his observation that ordinary composition could be achieved by this primitive function rather than by using two composition rules in the basic semantics, as was done in an earlier paper [2].)

Although FFP systems permit the definition and investigation of new functional forms, it is to be expected that most programming would use a fixed set of forms (whose controlling operators are primitives), as in FP, so that the algebraic laws for those forms could be employed, and so that a structured programming style could be used based on those forms.

In addition to its use in defining functional forms, metacomposition can be used to create recursive functions directly without the use of recursive definitions of the form **Def** $f \equiv E(f)$. For example, if $\rho MLAST \equiv$ null$\circ$tl$\circ$2 $\rightarrow$ 1$\circ$2; apply$\circ$[1, tl$\circ$2], then $\rho<MLAST> \equiv$ last, where last:$x \equiv x = <x_1, \ldots, x_n> \rightarrow x_n; \perp$. Thus the operator $<MLAST>$ works as follows:

$\mu(<MLAST>:<A,B>)$
$= \mu(\rho MLAST:<<MLAST>,<A,B>>)$
by metacomposition
$= \mu($apply$\circ$[1, tl$\circ$2]:$<<MLAST>,<A,B>>)$
$= \mu($apply:$<<MLAST>,<B>>)$
$= \mu(<MLAST>:<B>)$
$= \mu(\rho MLAST:<<MLAST>,<B>>)$
$= \mu(1\circ 2:<<MLAST>,<B>>)$
$= B$.

13.3.3 Summary of the properties of ρ and μ. So far we have shown how ρ maps atoms and sequences into functions and how those functions map objects into expressions. Actually, ρ and all FFP functions can be extended so that they are defined for all expressions. With such extensions the properties of ρ and μ can be summarized as follows:

1) $\mu \in$ [expressions $\rightarrow$ objects].
2) If x is an object, $\mu x = x$.
3) If e is an expression and $e = <e_1, \ldots , e_n>$, then $\mu e = <\mu e_1, \ldots , \mu e_n>$.
4) $\rho \in$ [expressions $\rightarrow$ [expressions $\rightarrow$ expressions]].
5) For any expression e, $\rho e = \rho(\mu e)$.
6) If x is an object and e an expression, then $\rho x:e = \rho x:(\mu e)$.
7) If x and y are objects, then $\mu(x:y) = \mu(\rho x:y)$.

In words: the meaning of an FFP application $(x:y)$ is found by applying ρx, the function represented by x, to y and then finding the meaning of the resulting expression (which is *usually* an object and is then its own meaning).

13.3.4 Cells, fetching, and storing. For a number of reasons it is convenient to create functions which serve as names. In particular, we shall need this facility in describing the semantics of definitions in FFP systems. To introduce naming functions, that is, the ability to *fetch* the contents of a cell with a given name from a store (a sequence of cells) and to *store* a cell with given name and contents in such a sequence, we introduce objects called *cells* and two new functional forms, *fetch* and *store*.

Cells

A *cell* is a triple $<CELL,name,contents>$. We use this form instead of the pair $<name,contents>$ so that cells can be distinguished from ordinary pairs.

Fetch

The functional form *fetch* takes an object n as its parameter (n is customarily an atom serving as a name); it is written $\uparrow n$ (read "fetch n"). Its definition for objects n and x is

$\uparrow n:x \equiv x = \phi \rightarrow \#;$ atom:$x \rightarrow \perp;$
$(1:x) = <CELL,n,c> \rightarrow c; \uparrow n\circ$tl:$x,$

where $\#$ is the atom "default." Thus $\uparrow n$ (fetch n) applied to a sequence gives the contents of the first cell in the sequence whose name is n; If there is no cell named n, the result is default, $\#$. Thus $\uparrow n$ is the name function for the name n. (We assume that $\rho FETCH$ is the primitive function such that $\rho<FETCH,n> \equiv \uparrow n$. Note that $\uparrow n$ simply passes over elements in its operand that are not cells.)

Store and push, pop, purge

Like fetch, *store* takes an object n as its parameter; it is written $\downarrow n$ ("store n"). When applied to a pair $<x,y>$, where y is a sequence, $\downarrow n$ removes the first cell named n from y, if any, then creates a new cell named n with contents x and appends it to y. Before defining $\downarrow n$ (store n) we shall specify four auxiliary functional forms. (These can be used in combination with fetch n and store n to obtain multiple, named, LIFO stacks within a storage sequence.) Two of these auxiliary forms are specified by recursive functional equations; each takes an object n as its parameter.

$$(\text{cellname } n) \equiv \text{atom} \rightarrow \bar{F};$$
$$\text{eq}\circ[\text{length}, \bar{3}] \rightarrow \text{eq}\circ[[\overline{CELL}, \bar{n}], [1, 2]]; \bar{F}$$
$$(\text{push } n) \equiv \text{pair} \rightarrow \text{apndl}\circ[[\overline{CELL}, \bar{n}, 1], 2]; \perp$$
$$(\text{pop } n) \equiv \text{null} \rightarrow \bar{\phi};$$
$$(\text{cellname } n)\circ 1 \rightarrow \text{tl}; \text{apndl}\circ[1, (\text{pop } n)\circ\text{tl}]$$
$$(\text{purge } n) \equiv \text{null} \rightarrow \bar{\phi}; (\text{cellname } n)\circ 1 \rightarrow (\text{purge } n)\circ\text{tl};$$
$$\text{apndl}\circ[1, (\text{purge } n)\circ\text{tl}]$$
$$\downarrow n \equiv \text{pair} \rightarrow (\text{push } n)\circ[1, (\text{pop } n)\circ 2]; \perp$$

The above functional forms work as follows. For $x \neq \perp$, (cellname n):x is T if x is a cell named n, otherwise it is F. (pop n):y removes the first cell named n from a sequence y; (purge n):y removes all cells named n from y. (push n):$<x,y>$ puts a cell named n with contents x at the head of sequence y; $\downarrow n$:$<x,y>$ is (push n):$<x$, (pop n):$y>$.

(Thus (push n):$<x,y> = y'$ pushes x onto the top of a "stack" named n in y'; x can be read by $\uparrow n$:$y' = x$ and can be removed by (pop n):y'; thus $\uparrow n\circ$(pop n):y' is the element below x in the stack n, provided there is more than one cell named n in y'.)

13.3.5 Definitions in FFP systems. The semantics of an FFP system depends on a fixed set of definitions D (a sequence of cells), just as an FP system depends on its informally given set of definitions. Thus the semantic function μ depends on D; altering D gives a new μ' that reflects the altered definitions. We have represented D as an *object* because in AST systems (Section 14) we shall want to transform D by applying functions to it and to fetch data from it—in addition to using it as the source of function definitions in FFP semantics.

If $<CELL,n,c>$ is the first cell named n in the sequence D (and n is an atom) then it has the same effect as the FP definition **Def** $n \equiv \rho c$, that is, the meaning of $(n:x)$ will be the same as that of ρc:x. Thus for example, if $<CELL,CONST,<COMP,2,1>>$ is the first cell in D named *CONST*, then it has the same effect as **Def** $CONST \equiv 2\circ 1$, and the FFP system with that D would find

$$\mu(CONST:<<x,y>,z>) = y$$

and consequently

$$\mu(<CONST,A>:B) = A.$$

In general, in an FFP system with definitions D, the meaning of an application of the form (*atom*:x) is dependent on D; if $\uparrow atom$:D $\neq \#\!\!\#$ (that is, *atom* is defined in D) then its meaning is $\mu(c:x)$, where $c = \uparrow atom$:D, the contents of the first cell in D named *atom*. If $\uparrow atom$:D $= \#\!\!\#$, then *atom* is not defined in D and either *atom* is primitive, i.e. the system knows how to compute $\rho atom$:x, and $\mu(atom:x) = \mu(\rho atom:x)$, otherwise $\mu(atom:x) = \perp$.

13.4 Formal Semantics for FFP Systems

We assume that a set A of atoms, a set D of definitions, a set P $\subset$ A of primitive atoms and the primitive functions they represent have all been chosen. We assume that ρa is the primitive function represented by a if a belongs to P, and that $\rho a = \perp$ if a belongs to Q, the set of atoms in A-P that are not defined in D. Although ρ is defined for all expressions (see 13.3.3), the formal semantics uses its definition only on P and Q. The functions that ρ assigns to other expressions x are implicitly determined and applied in the following semantic rules for evaluating $\mu(x:y)$. The above choices of A and D, and of P and the associated primitive functions determine the objects, expressions, and the semantic function μ_D for an FFP system. (We regard D as fixed and write μ for μ_D.) We assume D is a sequence and that $\uparrow y$:D can be computed (by the function $\uparrow y$ as given in Section 13.3.4) for any atom y. With these assumptions we define μ as the least fixed point of the functional τ, where the function $\tau\mu$ is defined as follows for any function μ (for all expressions x, x_i, y, y_i, z, and w):

$$(\tau\mu)x \equiv x \in \text{A} \rightarrow x;$$
$$x = <x_1, \ldots, x_n> \rightarrow <\mu x_1, \ldots, \mu x_n>;$$
$$x = (y:z) \rightarrow$$
$$(y \in \text{A} \;\&\; (\uparrow y\text{:D}) = \#\!\!\# \rightarrow \mu((\rho y)(\mu z));$$
$$y \in \text{A} \;\&\; (\uparrow y\text{:D}) = w \rightarrow \mu(w:z);$$
$$y = <y_1, \ldots, y_n> \rightarrow \mu(y_1:<y,z>); \mu(\mu y:z)); \perp$$

The above description of μ expands the operator of an application by definitions and by metacomposition before evaluating the operand. It is assumed that predicates like "$x \in$ A" in the above definition of $\tau\mu$ are $\perp$-preserving (e.g., "$\perp \in$ A" has the value $\perp$) and that the conditional expression itself is also $\perp$-preserving. Thus $(\tau\mu)\perp \equiv \perp$ and $(\tau\mu)(\perp:z) \equiv \perp$. This concludes the semantics of FFP systems.

14. Applicative State Transition Systems (AST Systems)

14.1 Introduction

This section sketches a class of systems mentioned earlier as alternatives to von Neumann systems. It must be emphasized again that these applicative state transition systems are put forward not as practical programming systems in their present form, but as examples of a class in which applicative style programming is made available in a history sensitive, but non-von Neumann system. These systems are loosely coupled to states and depend on an underlying applicative system for both

their programming language and the description of their state transitions. The underlying applicative system of the AST system described below is an FFP system, but other applicative systems could also be used.

To understand the reasons for the structure of AST systems, it is helpful first to review the basic structure of a von Neumann system, Algol, observe its limitations, and compare it with the structure of AST systems. After that review a minimal AST system is described; a small, top-down, self-protecting system program for file maintenance and running user programs is given, with directions for installing it in the AST system and for running an example user program. The system program uses "name functions" instead of conventional names and the user may do so too. The section concludes with subsections discussing variants of AST systems, their general properties, and naming systems.

14.2 The Structure of Algol Compared to That of AST Systems

An Algol program is a sequence of statements, each representing a transformation of the Algol state, which is a complex repository of information about the status of various stacks, pointers, and variable mappings of identifiers onto values, etc. Each statement communicates with this constantly changing state by means of complicated protocols peculiar to itself and even to its different parts (e.g., the protocol associated with the variable x depends on its occurrence on the left or right of an assignment, in a declaration, as a parameter, etc.).

It is as if the Algol state were a complex "store" that communicates with the Algol program through an enormous "cable" of many specialized wires. The complex communications protocols of this cable are fixed and include those for every statement type. The "meaning" of an Algol program must be given in terms of the total effect of a vast number of communications with the state via the cable and its protocols (plus a means for identifying the output and inserting the input into the state). By comparison with this massive cable to the Algol state/store, the cable that is the von Neumann bottleneck of a computer is a simple, elegant concept.

Thus Algol statements are not expressions representing state-to-state functions that are built up by the use of orderly combining forms from simpler state-to-state functions. Instead they are complex *messages* with context-dependent parts that nibble away at the state. Each part transmits information to and from the state over the cable by its own protocols. There is no provision for applying general functions to the *whole* state and thereby making large changes in it. The possibility of large, powerful transformations of the state S by function application, $S \rightarrow f{:}S$, is in fact inconceivable in the von Neumann—cable and protocol—context: there could be no assurance that the new state $f{:}S$ would match the cable and its fixed protocols unless f is restricted to the tiny changes allowed by the cable in the first place.

We want a computing system whose semantics does not depend on a host of baroque protocols for communicating with the state, and we want to be able to make large transformations in the state by the application of general functions. AST systems provide one way of achieving these goals. Their semantics has two protocols for getting information from the state: (1) get from it the definition of a function to be applied, and (2) get the whole state itself. There is one protocol for changing the state: compute the new state by function application. Besides these communications with the state, AST semantics is applicative (i.e. FFP). It does not depend on state changes because the state does not change at all during a computation. Instead, the result of a computation is output *and* a new state. The structure of an AST state is slightly restricted by one of its protocols: It must be possible to identify a definition (i.e. cell) in it. Its structure—it is a sequence—is far simpler than that of the Algol state.

Thus the structure of AST systems avoids the complexity and restrictions of the von Neumann state (with its communications protocols) while achieving greater power and freedom in a radically different and simpler framework.

14.3 Structure of an AST System

An AST system is made up of three elements:

1) An *applicative subsystem* (such as an FFP system).

2) A *state* D that is the set of definitions of the applicative subsystem.

3) A set of *transition rules* that describe how inputs are transformed into outputs and how the state D is changed.

The programming language of an AST system is just that of its applicative subsystem. (From here on we shall assume that the latter is an FFP system.) Thus AST systems can use the FP programming style we have discussed. The applicative subsystem cannot change the state D and it does not change during the evaluation of an expression. A new state is computed along with output and replaces the old state when output is issued. (Recall that a set of definitions D is a sequence of cells; a cell name is the name of a defined function and its contents is the defining expression. Here, however, some cells may name data rather than functions; a data name n will be used in $\uparrow n$ (fetch n) whereas a function name will be used as an operator itself.)

We give below the transition rules for the elementary AST system we shall use for examples of programs. These are perhaps the simplest of many possible transition rules that could determine the behavior of a great variety of AST systems.

14.3.1 Transition rules for an elementary AST system. When the system receives an input x, it forms the application $(SYSTEM{:}x)$ and then proceeds to obtain its meaning in the FFP subsystem, using the current state D as the set of definitions. *SYSTEM* is the distinguished name of a function defined in D (i.e. it is the "system program"). Normally the result is a pair

$\mu(SYSTEM{:}x) = \langle o,d \rangle$

where o is the system output that results from input x and d becomes the new state D for the system's next input. Usually d will be a copy or partly changed copy of the old state. If $\mu(SYSTEM{:}x)$ is not a pair, the output is an error message and the state remains unchanged.

14.3.2 Transition rules: exception conditions and startup. Once an input has been accepted, our system will not accept another (except $\langle RESET,x \rangle$, see below) until an output has been issued and the new state, if any, installed. The system will accept the input $\langle RESET,x \rangle$ at any time. There are two cases: (a) If *SYSTEM* is defined in the current state D, then the system aborts its current computation without altering D and treats x as a new normal input; (b) if *SYSTEM* is not defined in D, then x is appended to D as its first element. (This ends the complete description of the transition rules for our elementary AST system.)

If *SYSTEM* is defined in D it can always prevent any change in its own definition. If it is not defined, an ordinary input x will produce $\mu(SYSTEM{:}x) = \perp$ and the transition rules yield an error message and an unchanged state; on the other hand, the input $\langle RESET, \langle CELL,SYSTEM,s \rangle \rangle$ will define *SYSTEM* to be s.

14.3.3 Program access to the state; the function *ρ DEFS*. Our FFP subsystem is required to have one new primitive function, defs, named *DEFS* such that for any object $x \neq \perp$,

$\text{defs}{:}x = \rho DEFS{:}x = \text{D}$

where D is the current state and set of definitions of the AST system. This function allows programs access to the whole state for any purpose, including the essential one of computing the successor state.

14.4 An Example of a System Program

The above description of our elementary AST system, plus the FFP subsystem and the FP primitives and functional forms of earlier sections, specify a complete history-sensitive computing system. Its input and output behavior is limited by its simple transition rules, but otherwise it is a powerful system once it is equipped with a suitable set of definitions. As an example of its use we shall describe a small system program, its installation, and operation.

Our example system program will handle queries and updates for a file it maintains, evaluate FFP expressions, run general user programs that do not damage the file or the state, and allow authorized users to change the set of definitions and the system program itself. All inputs it accepts will be of the form $\langle key,input \rangle$ where *key* is a code that determines both the input class (*system-change, expression, program, query, update*) and also the identity of the user and his authority to use the system for the given input class. We shall not specify a format for *key*. *Input* is the input itself, of the class given by *key*.

14.4.1 General plan of the system program. The state D of our AST system will contain the definitions of all nonprimitive functions needed for the system program and for users' programs. (Each definition is in a cell of the sequence D.) In addition, there will be a cell in D named *FILE* with contents *file*, which the system maintains. We shall give FP definitions of functions and later show how to get them into the system in their FFP form. The transition rules make the input the operand of *SYSTEM*, but our plan is to use name-functions to refer to data, so the first thing we shall do with the input is to create two cells named *KEY* and *INPUT* with contents *key* and *input* and append these to D. This sequence of cells has one each for *key*, *input*, and *file*; it will be the operand of our main function called subsystem. Subsystem can then obtain *key* by applying $\uparrow KEY$ to its operand, etc. Thus the definition

Def system ≡ pair → subsystem∘f; [$\overline{NONPAIR}$, defs]

where

$f \equiv \downarrow INPUT \circ [2, \downarrow KEY \circ [1, \text{defs}]]$

causes the system to output *NONPAIR* and leave the state unchanged if the input is not a pair. Otherwise, if it is $\langle key,input \rangle$, then

$$f{:}\langle key,input \rangle = \langle \langle CELL,INPUT,input \rangle, \langle CELL,KEY,key \rangle, d_1, \ldots, d_n \rangle$$

where $\text{D} = \langle d_1, \ldots, d_n \rangle$. (We might have constructed a different operand than the one above, one with just three cells, for *key*, *input*, and *file*. We did not do so because real programs, unlike subsystem, would contain many name functions referring to data in the state, and this "standard" construction of the operand would suffice then as well.)

14.4.2 The "subsystem" function. We now give the FP definition of the function subsystem, followed by brief explanations of its six cases and auxiliary functions.

```
Def subsystem ≡
  is-system-change∘↑KEY → [report-change, apply]∘[↑INPUT, defs];
  is-expression∘↑KEY → [↑INPUT, defs];
  is-program∘↑KEY → system-check∘apply∘[↑INPUT, defs];
  is-query∘↑KEY → [query-response∘[↑INPUT, ↑FILE], defs];
  is-update∘↑KEY →
            [report-update, ↓FILE∘[update, defs]]
                                  ∘[↑INPUT, ↑FILE];
                  [report-error∘[↑KEY,↑INPUT], defs].
```

This subsystem has five "$p \rightarrow f$;" clauses and a final default function, for a total of six classes of inputs; the treatment of each class is given below. Recall that the *operand* of subsystem is a sequence of cells containing *key*, *input*, and *file* as well as all the defined functions of D, and that subsystem:*operand* = $\langle output,newstate \rangle$.

Default inputs. In this case the result is given by the last (default) function of the definition when *key* does not satisfy any of the preceding clauses. The output is report-error: $\langle key,input \rangle$. The state is unchanged since it is given by defs:*operand* = D. (We leave to the reader's imagination what the function report-error will generate from its operand.)

System-change inputs. When

is-system-change∘↑*KEY*:*operand* =
is-system-change:*key* = *T*,

key specifies that the user is authorized to make a system change and that *input* = ↑*INPUT*:*operand* represents a function *f* that is to be applied to D to produce the new state *f*:D. (Of course *f*:D can be a useless new state; no constraints are placed on it.) The output is a report, namely report-change:<*input*,D>.

Expression inputs. When is-expression:*key* = *T*, the system understands that the output is to be the meaning of the FFP expression *input*; ↑*INPUT*:*operand* produces it and it is evaluated, as are all expressions. The state is unchanged.

Program inputs and system self-protection. When is-program:*key* = *T*, both the output and new state are given by (ρ*input*):D = <*output*,*newstate*>. If *newstate* contains *file* in suitable condition and the definitions of system and other protected functions, then
system-check: <*output*,*newstate*> = <*output*,*newstate*>.
Otherwise, system-check:<*output*,*newstate*>
= <*error-report*,D>.
Although *program* inputs can make major, possibly disastrous changes in the state when it produces *newstate*, system-check can use any criteria to either allow it to become the actual new state or to keep the old. A more sophisticated system-check might correct only prohibited changes in the state. Functions of this sort are possible because they can always access the old state for comparison with the new state-to-be and control what state transition will finally be allowed.

File query inputs. If is-query:*key* = *T*, the function query-response is designed to produce the output = answer to the query *input* from its operand <*input*,*file*>.

File update inputs. If is-update:*key* = *T*, *input* specifies a file transaction understood by the function update, which computes *updated-file* = update:<*input*,*file*>. Thus ↓*FILE* has <*updated-file*,D> as its operand and thus stores the updated file in the cell *FILE* in the new state. The rest of the state is unchanged. The function report-update generates the output from its operand <*input*,*file*>.

14.4.3 Installing the system program. We have described the function called system by some FP definitions (using auxiliary functions whose behavior is only indicated). Let us suppose that we have FP definitions for all the nonprimitive functions required. Then each definition can be converted to give the name and contents of a cell in D (of course this conversion itself would be done by a better system). The conversion is accomplished by changing each FP function name to its equivalent atom (e.g., update becomes *UPDATE*) and by replacing functional forms by sequences whose first member is the controlling function for the particular form. Thus ↓*FILE*∘[update, defs] is converted to

<*COMP*,<*STORE*,*FILE*>,
<*CONS*,*UPDATE*,*DEFS*>>,

and the FP function is the same as that represented by the FFP object, provided that update ≡ ρ*UPDATE* and *COMP*, *STORE*, and *CONS* represent the controlling functions for composition, store, and construction.

All FP definitions needed for our system can be converted to cells as indicated above, giving a sequence D_0. We assume that the AST system has an empty state to start with, hence *SYSTEM* is not defined. We want to define *SYSTEM* initially so that it will install its next input as the state; having done so we can then input D_0 and all our definitions will be installed, including our program—system—itself. To accomplish this we enter our first input
<*RESET*, <*CELL*,*SYSTEM*,*loader*>>
where *loader* ≡ <*CONS*, <*CONST*,*DONE*>,*ID*>.
Then, by the transition rule for *RESET* when *SYSTEM* is undefined in D, the cell in our input is put at the head of D = φ, thus defining ρ*SYSTEM* ≡ ρ*loader* ≡ [$\overline{DONE}$, id]. Our second input is D_0, the set of definitions we wish to become the state. The regular transition rule causes the AST system to evaluate
μ(*SYSTEM*:D_0) = [$\overline{DONE}$, id]:D_0 = <*DONE*,D_0>. Thus the output from our second input is *DONE*, the new state is D_0, and ρ*SYSTEM* is now our system program (which only accepts inputs of the form <*key*,*input*>).

Our next task is to load the file (we are given an initial value *file*). To load it we input a *program* into the newly installed system that contains *file* as a constant and stores it in the state; the input is
<*program-key*, [$\overline{DONE}$,*store-file*]> where

ρ*store-file* ≡ ↓*FILE*∘[$\overline{file}$, id].

Program-key identifies [$\overline{DONE}$, *store-file*] as a program to be applied to the state D_0 to give the output and new state D_1, which is:

ρ*store-file*:D_0 = ↓*FILE*∘[$\overline{file}$, id]:D_0,

or D_0 with a cell containing *file* at its head. The output is $\overline{DONE}$:D_0 = *DONE*. We assume that system-check will pass <*DONE*,D_1> unchanged. FP expressions have been used in the above in place of the FFP objects they denote, e.g. $\overline{DONE}$ for <*CONST*,*DONE*>.

14.4.4 Using the system. We have not said how the system's file, queries or updates are structured, so we cannot give a detailed example of file operations. However, the structure of subsystem shows clearly how the system's response to queries and updates depends on the functions query-response, update, and report-update.

Let us suppose that matrices *m*, *n* named *M*, and *N* are stored in D and that the function MM described earlier is defined in D. Then the input

<*expression-key*, (*MM*∘[↑*M*, ↑*N*]∘*DEFS*:#)>

would give the product of the two matrices as output and an unchanged state. *Expression-key* identifies the application as an expression to be evaluated and since defs:# = D and [↑*M*, ↑*N*]:D = <*m*,*n*>, the value of the expression is the result MM:<*m*,*n*>, which is the output.

Our miniature system program has no provision for giving control to a user's program to process many inputs, but it would not be difficult to give it that capability while still monitoring the user's program with the option of taking control back.

14.5 Variants of AST Systems

A major extension of the AST systems suggested above would provide combining forms, "system forms," for building a new AST system from simpler, component AST systems. That is, a system form would take AST systems as parameters and generate a new AST system, just as a functional form takes functions as parameters and generates new functions. These system forms would have properties like those of functional forms and would become the "operations" of a useful "algebra of systems" in much the same way that functional forms are the "operations" of the algebra of programs. However, the problem of finding useful system forms is much more difficult, since they must handle *RESETS*, match inputs and outputs, and combine history-sensitive systems rather than fixed functions.

Moreover, the usefulness or need for system forms is less clear than that for functional forms. The latter are essential for building a great variety of functions from an initial primitive set, whereas, even without system forms, the facilities for building AST systems are already so rich that one could build virtually any system (with the general input and output properties allowed by the given AST scheme). Perhaps system forms would be useful for building systems with complex input and output arrangements.

14.6 Remarks About AST Systems

As I have tried to indicate above, there can be innumerable variations in the ingredients of an AST system—how it operates, how it deals with input and output, how and when it produces new states, and so on. In any case, a number of remarks apply to any reasonable AST system:

a) A state transition occurs once per major computation and can have useful mathematical properties. State transitions are not involved in the tiniest details of a computation as in conventional languages; thus the linguistic von Neumann bottleneck has been eliminated. No complex "cable" or protocols are needed to communicate with the state.

b) Programs are written in an applicative language that can accommodate a great range of changeable parts, parts whose power and flexibility exceed that of any von Neumann language so far. The word-at-a-time style is replaced by an applicative style; there is no division of programming into a world of expressions and a world of statements. Programs can be analyzed and optimized by an algebra of programs.

c) Since the state cannot change during the computation of system:x, there are no side effects. Thus independent applications can be evaluated in parallel.

d) By defining appropriate functions one can, I believe, introduce major new features at any time, using the same framework. Such features must be built into the framework of a von Neumann language. I have in mind such features as: "stores" with a great variety of naming systems, types and type checking, communicating parallel processes, nondeterminacy and Dijkstra's "guarded command" constructs [8], and improved methods for structured programming.

e) The framework of an AST system comprises the syntax and semantics of the underlying applicative system plus the system framework sketched above. By current standards, this is a tiny framework for a language and is the only fixed part of the system.

14.7 Naming Systems in AST and von Neumann Models

In an AST system, naming is accomplished by functions as indicated in Section 13.3.3. Many useful functions for altering and accessing a store can be defined (e.g. push, pop, purge, typed fetch, etc.). All these definitions and their associated naming systems can be introduced without altering the AST framework. Different kinds of "stores" (e.g., with "typed cells") with individual naming systems can be used in one program. A cell in one store may contain another entire store.

The important point about AST naming systems is that they utilize the functional nature of names (Reynolds' GEDANKEN [19] also does so to some extent within a von Neumann framework). Thus name functions can be composed and combined with other functions by functional forms. In contrast, functions and names in von Neumann languages are usually disjoint concepts and the function-like nature of names is almost totally concealed and useless, because a) names cannot be applied as functions; b) there are no general means to combine names with other names and functions; c) the objects to which name functions apply (stores) are not accessible as objects.

The failure of von Neumann languages to treat names as functions may be one of their more important weaknesses. In any case, the ability to use names as functions and stores as objects may turn out to be a useful and important programming concept, one which should be thoroughly explored.

15. Remarks About Computer Design

The dominance of von Neumann languages has left designers with few intellectual models for practical computer designs beyond variations of the von Neumann computer. Data flow models [1] [7] [13] are one alternative class of history-sensitive models. The substitution rules of lambda-calculus based languages present serious problems for the machine designer. Berkling [3] has developed a modified lambda calculus that has three kinds of applications and that makes renaming of vari-

ables unnecessary. He has developed a machine to evaluate expressions of this language. Further experience is needed to show how sound a basis this language is for an effective programming style and how efficient his machine can be.

Magó [15] has developed a novel applicative machine built from identical components (of two kinds). It evaluates, directly, FP-like and other applicative expressions from the bottom up. It has no von Neumann store and no address register, hence no bottleneck; it is capable of evaluating many applications in parallel; its built-in operations resemble FP operators more than von Neumann computer operations. It is the farthest departure from the von Neumann computer that I have seen.

There are numerous indications that the applicative style of programming can become more powerful than the von Neumann style. Therefore it is important for programmers to develop a new class of history-sensitive models of computing systems that embody such a style and avoid the inherent efficiency problems that seem to attach to lambda-calculus based systems. Only when these models and their applicative languages have proved their superiority over conventional languages will we have the economic basis to develop the new kind of computer that can best implement them. Only then, perhaps, will we be able to fully utilize large-scale integrated circuits in a computer design not limited by the von Neumann bottleneck.

16. Summary

The fifteen preceding sections of this paper can be summarized as follows.

Section 1. Conventional programming languages are large, complex, and inflexible. Their limited expressive power is inadequate to justify their size and cost.

Section 2. The models of computing systems that underlie programming languages fall roughly into three classes: (a) simple operational models (e.g., Turing machines), (b) applicative models (e.g., the lambda calculus), and (c) von Neumann models (e.g., conventional computers and programming languages). Each class of models has an important difficulty: The programs of class (a) are inscrutable; class (b) models cannot save information from one program to the next; class (c) models have unusable foundations and programs that are conceptually unhelpful.

Section 3. Von Neumann computers are built around a bottleneck: the word-at-a-time tube connecting the CPU and the store. Since a program must make its overall change in the store by pumping vast numbers of words back and forth through the von Neumann bottleneck, we have grown up with a style of programming that concerns itself with this word-at-a-time traffic through the bottleneck rather than with the larger conceptual units of our problems.

Section 4. Conventional languages are based on the programming style of the von Neumann computer. Thus variables = storage cells; assignment statements = fetching, storing, and arithmetic; control statements = jump and test instructions. The symbol ":=" is the linguistic von Neumann bottleneck. Programming in a conventional—von Neumann—language still concerns itself with the word-at-a-time traffic through this slightly more sophisticated bottleneck. Von Neumann languages also split programming into a world of expressions and a world of statements; the first of these is an orderly world, the second is a disorderly one, a world that structured programming has simplified somewhat, but without attacking the basic problems of the split itself and of the word-at-a-time style of conventional languages.

Section 5. This section compares a von Neumann program and a functional program for inner product. It illustrates a number of problems of the former and advantages of the latter: e.g., the von Neumann program is repetitive and word-at-a-time, works only for two vectors named a and b of a given length n, and can only be made general by use of a procedure declaration, which has complex semantics. The functional program is nonrepetitive, deals with vectors as units, is more hierarchically constructed, is completely general, and creates "housekeeping" operations by composing high-level housekeeping operators. It does not name its arguments, hence it requires no procedure declaration.

Section 6. A programming language comprises a framework plus some changeable parts. The framework of a von Neumann language requires that most features must be built into it; it can accommodate only limited changeable parts (e.g., user-defined procedures) because there must be detailed provisions in the "state" and its transition rules for all the needs of the changeable parts, as well as for all the features built into the framework. The reason the von Neumann framework is so inflexible is that its semantics is too closely coupled to the state: every detail of a computation changes the state.

Section 7. The changeable parts of von Neumann languages have little expressive power; this is why most of the language must be built into the framework. The lack of expressive power results from the inability of von Neumann languages to effectively use combining forms for building programs, which in turn results from the split between expressions and statements. Combining forms are at their best in expressions, but in von Neumann languages an expression can only produce a single word; hence expressive power in the world of expressions is mostly lost. A further obstacle to the use of combining forms is the elaborate use of naming conventions.

Section 8. APL is the first language not based on the lambda calculus that is not word-at-a-time and uses functional combining forms. But it still retains many of the problems of von Neumann languages.

Section 9. Von Neumann languages do not have useful properties for reasoning about programs. Axiomatic and denotational semantics are precise tools for describing and understanding conventional programs,

but they only talk about them and cannot alter their ungainly properties. Unlike von Neumann languages, the language of ordinary algebra is suitable both for stating its laws and for transforming an equation into its solution, all within the "language."

Section 10. In a history-sensitive language, a program can affect the behavior of a subsequent one by changing some store which is saved by the system. Any such language requires some kind of state transition semantics. But it does not need semantics closely coupled to states in which the state changes with every detail of the computation. "Applicative state transition" (AST) systems are proposed as history-sensitive alternatives to von Neumann systems. These have: (a) loosely coupled state-transition semantics in which a transition occurs once per major computation; (b) simple states and transition rules; (c) an underlying applicative system with simple "reduction" semantics; and (d) a programming language and state transition rules both based on the underlying applicative system and its semantics. The next four sections describe the elements of this approach to non-von Neumann language and system design.

Section 11. A class of informal functional programming (FP) systems is described which use no variables. Each system is built from objects, functions, functional forms, and definitions. Functions map objects into objects. Functional forms combine existing functions to form new ones. This section lists examples of primitive functions and functional forms and gives sample programs. It discusses the limitations and advantages of FP systems.

Section 12. An "algebra of programs" is described whose variables range over the functions of an FP system and whose "operations" are the functional forms of the system. A list of some twenty-four laws of the algebra is followed by an example proving the equivalence of a nonrepetitive matrix multiplication program and a recursive one. The next subsection states the results of two "expansion theorems" that "solve" two classes of equations. These solutions express the "unknown" function in such equations as an infinite conditional expansion that constitutes a case-by-case description of its behavior and immediately gives the necessary and sufficient conditions for termination. These results are used to derive a "recursion theorem" and an "iteration theorem," which provide ready-made expansions for some moderately general and useful classes of "linear" equations. Examples of the use of these theorems treat: (a) correctness proofs for recursive and iterative factorial functions, and (b) a proof of equivalence of two iterative programs. A final example deals with a "quadratic" equation and proves that its solution is an idempotent function. The next subsection gives the proofs of the two expansion theorems.

The algebra associated with FP systems is compared with the corresponding algebras for the lambda calculus and other applicative systems. The comparison shows some advantages to be drawn from the severely restricted FP systems, as compared with the much more powerful classical systems. Questions are suggested about algorithmic reduction of functions to infinite expansions and about the use of the algebra in various "lazy evaluation" schemes.

Section 13. This section describes formal functional programming (FFP) systems that extend and make precise the behavior of FP systems. Their semantics are simpler than that of classical systems and can be shown to be consistent by a simple fixed-point argument.

Section 14. This section compares the structure of Algol with that of applicative state transition (AST) systems. It describes an AST system using an FFP system as its applicative subsystem. It describes the simple state and the transition rules for the system. A small self-protecting system program for the AST system is described, and how it can be installed and used for file maintenance and for running user programs. The section briefly discusses variants of AST systems and functional naming systems that can be defined and used within an AST system.

Section 15. This section briefly discusses work on applicative computer designs and the need to develop and test more practical models of applicative systems as the future basis for such designs.

Acknowledgments. In earlier work relating to this paper I have received much valuable help and many suggestions from Paul R. McJones and Barry K. Rosen. I have had a great deal of valuable help and feedback in preparing this paper. James N. Gray was exceedingly generous with his time and knowledge in reviewing the first draft. Stephen N. Zilles also gave it a careful reading. Both made many valuable suggestions and criticisms at this difficult stage. It is a pleasure to acknowledge my debt to them. I also had helpful discussions about the first draft with Ronald Fagin, Paul R. McJones, and James H. Morris, Jr. Fagin suggested a number of improvements in the proofs of theorems.

Since a large portion of the paper contains technical material, I asked two distinguished computer scientists to referee the third draft. David J. Gries and John C. Reynolds were kind enough to accept this burdensome task. Both gave me large, detailed sets of corrections and overall comments that resulted in many improvements, large and small, in this final version (which they have not had an opportunity to review). I am truly grateful for the generous time and care they devoted to reviewing this paper.

Finally, I also sent copies of the third draft to Gyula A. Magó, Peter Naur, and John H. Williams. They were kind enough to respond with a number of extremely helpful comments and corrections. Geoffrey A. Frank and Dave Tolle at the University of North Carolina reviewed Magó's copy and pointed out an important error in the definition of the semantic function of FFP systems. My grateful thanks go to all these kind people for their help.

References

1. Arvind, and Gostelow, K.P. A new interpreter for data flow schemas and its implications for computer architecture. Tech. Rep. No. 72, Dept. Comptr. Sci., U. of California, Irvine, Oct. 1975.

2. Backus, J. Programming language semantics and closed applicative languages. Conf. Record ACM Symp. on Principles of Programming Languages, Boston, Oct. 1973, 71–86.

3. Berkling, K.J. Reduction languages for reduction machines. Interner Bericht ISF-76-8, Gesellschaft für Mathematik und Datenverarbeitung MBH, Bonn, Sept. 1976.

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

5. Church, A. *The Calculi of Lambda-Conversion.* Princeton U. Press, Princeton, N.J., 1941.

6. Curry, H.B., and Feys, R. *Combinatory Logic, Vol. 1.* North-Holland Pub. Co., Amsterdam, 1958.

7. Dennis, J.B. First version of a data flow procedure language. Tech. Mem. No. 61, Lab. for Comptr. Sci., M.I.T., Cambridge, Mass., May 1973.

8. Dijkstra, E.W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

9. Friedman, D.P., and Wise, D.S. CONS should not evaluate its arguments. In *Automata, Languages and Programming,* S. Michaelson and R. Milner, Eds., Edinburgh U. Press, Edinburgh, 1976, pp. 257–284.

10. Henderson, P., and Morris, J.H. Jr. A lazy evaluator. Conf. Record Third ACM Symp. on Principles of Programming Languages, Atlanta, Ga., Jan. 1976, pp. 95–103.

11. Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM 12,* 10 (Oct. 1969), 576–583.

12. Iverson, K. *A Programming Language.* Wiley, New York, 1962.

13. Kosinski, P. A data flow programming language. Rep. RC 4264, IBM T.J. Watson Research Ctr., Yorktown Heights, N.Y., March 1973.

14. Landin, P.J. The mechanical evaluation of expressions. *Computer J. 6,* 4 (1964), 308–320.

15. Magó, G.A. A network of microprocessors to execute reduction languages. To appear in *Int. J. Comptr. and Inform. Sci.*

16. Manna, Z., Ness, S., and Vuillemin, J. Inductive methods for proving properties of programs. *Comm. ACM 16,* 8 (Aug. 1973) 491–502.

17. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Pt. 1. *Comm. ACM 3,* 4 (April 1960), 184–195.

18. McJones, P. A Church-Rosser property of closed applicative languages. Rep. RJ 1589, IBM Res. Lab., San Jose, Calif., May 1975.

19. Reynolds, J.C. GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept. *Comm. ACM 13,* 5 (May 1970), 308–318.

20. Reynolds, J.C. Notes on a lattice-theoretic approach to the theory of computation. Dept. Syst. and Inform. Sci., Syracuse U., Syracuse, N.Y., 1972.

21. Scott, D. Outline of a mathematical theory of computation. Proc. 4th Princeton Conf. on Inform. Sci. and Syst., 1970.

22. Scott, D. Lattice-theoretic models for various type-free calculi. Proc. Fourth Int. Congress for Logic, Methodology, and the Philosophy of Science, Bucharest, 1972.

23. Scott, D., and Strachey, C. Towards a mathematical semantics for computer languages. Proc. Symp. on Comptrs. and Automata, Polytechnic Inst. of Brooklyn, 1971.