Is computer science based on the wrong fundamental concept of "program"? An extended concept

John Backus

IBM Research Laboratory
5600 Cottle Road
San Jose, California 95193 USA

1. Introduction

1.1 The von Neumann concept of "program"

Ever since John von Neumann and others proposed the machine we call the von Neumann computer, programmers have been writing "von Neumann style" programs. Originally these programs were written in machine language, then in assembly language, then in Fortran, and then in a great variety of so-called higher level languages. The units of action specified by a program element grew as these languages evolved, but all of these programs were primarily concerned with two things: (1) the transmission of input and output between the "store" and the outside world, and (2) the transformation of the store from its state at input to some new state in which the desired output was available. (Of course, the concept of the "store" also evolved from a device to an abstract entity comprising a set of "cells" each with a "name" and "contents".) Additionally these languages allowed one to make various assertions or declarations about the contents of the store, for example, that a certain cell contains an integer.

The variety and dynamic nature of input-output operations in the great assortment of von Neumann languages make it difficult to include these operations in a uniform concept of "program" that is common to all these languages over the past thirty years. So, following the example of Algol 60, we shall not attempt to do so.

Having dismissed the question of input-output, we are left with what may be considered the most fundamental concept of "program": a "program" is a mapping of some domain of "stores" into itself. Each store in the

domain of a machine language is a set of pairs, each pair being the number of a cell and the contents of that cell. In assembly language the cell names of a store can be symbols; in higher level languages the cell names can be more complicated, and some languages permit stores whose "cells" can hold "contents" of more than one "word". But in all cases a "store" is an association between "names" and "objects", each "name" denoting a "cell" in the store having a certain "object" as its "contents".

Thus, neglecting input-output behavior, when any von Neumann "program" is given a store s in its domain, it will "execute" and either the execution will go on forever or it will stop and yield some new store s' in its domain (it may happen that s = s').

At this point we must confess that our simple notion of "program" as a mapping of "stores" into "stores" differs from various precise notions to be found in works on denotational semantics [Scott & Strachey 71], e.g., [Milne & Strachey 76], [Stoy 77], and [Tennent 76], even though our simple notion reflects the spirit of these precise ones. To explain the details of how a program achieves a store-to-store mapping, or to explain dynamic storage reallocation, scope rules for variables, GOTOs, side effects, error stops, or other such issues, "programs" are assigned mappings in denotational semantics that are more complex than store-to-store mappings. These mappings often involve "environments" and "continuations" as well as "stores".

In spite of the detailed explanations and language complications that force denotational semantics to represent "programs" as more complex mappings, I submit that the single concept that is closest to our intuitive understanding and to all the various detailed concepts embodied in different von Neumann languages is that "programs" represent mappings of "stores" into "stores". (For each particular notion of "program" we must suitably choose the "names" and "contents" needed to construct the domain of its "stores".)

1.2 A more general concept of "program"; a basic question about the conventional concept

To make it easier to consider a wider range of concepts of "program", some of which are non-von Neumann, let us define a more general property of "programs" and then define what we mean by "von Neumann programs":

(A) A "program" represents a mapping of some domain D into itself.

(B) The domain associated with "von Neumann programs" is some domain of "stores".

Thus for each notion of "program" there is associated a particular domain D; and for each kind of "von Neumann program" there is a particular domain of "stores" built from certain "names" and certain "contents". Of course such a domain includes each possible association of "names" with "contents" in one of its "stores".

If you accept these notions as fundamental elements of the concept of "program" and of "von Neumann program", then the question I should like to raise for your consideration is an important one, one which, oddly enough, seems to have received little attention. The question is this: is the choice of "stores" as the domain for "programs" the correct choice? That is, is there perhaps some other domain such that, if the notion of "program" were associated with mappings of this new domain into itself, then the resulting concept of "program" could be simpler, more powerful and elegant than the von Neumann concept?

1.3 Evolution of the von Neumann concept; psychological barriers to adoption of non-von Neumann concepts

Before we consider notions of "program" founded on other domains, let us review some of the reasons why "stores" were chosen for the domain of conventional programs. Of course, in the first place the von Neumann computer itself required programs based on the domain of

stores. And when "higher level" languages began to evolve it was important that their programs correspond closely with machine programs, therefore they naturally adopted an abstract notion of "store" quite close to that of the machine. Thus the evolution of programming languages from machine languages is a natural and basic reason for the choice of "stores" as the domain for "programs". But I believe there is another, deeper reason.

In natural language and in mathematics one of the most universal and deeply ingrained practices in thinking and writing is the use of names to stand in place of their referents. In every natural language sentence or mathematical expression most symbols denote something other than themselves. Thus in "a+b" it is universally understood that we are to add, not the letters "a" and "b", but some numbers to which they refer. The store that is implicit in every conventional program is the repository of this name-referent association that is so much a part of our traditional way of thinking.

If we choose any domain other than "stores" for our concept of "program", then it may no longer be possible for the programmer to use a "name" or "variable" to refer to an object. Thus non-von Neumann concepts of "program", those employing domains other than "stores", threaten to violate deep-rooted traditions of thought, therefore such concepts tend to confuse and disturb programmers.

Because the conventional concept of "program" is so closely linked by its use of "stores" to the traditional use of names in natural language and mathematics, the adoption of new concepts may cause many computer scientists a certain amount of anguish. But if we are to consider new concepts of "program" that may yield a more profound order in the realm of programs, it is important that we be aware of the psychological difficulties we may face in dealing with such new ideas.

Alternative concepts of "program" not based on the domain of "stores" have been evolving over a long period. Their evolution has been

confused because many developers of the new view have been unable to free themselves from the old one and have sought to find a viable mixture of the two. Thus there is as yet no consensus on the key elements of a new concept. Pure LISP was the first language whose program domain was not "stores". But most versions of LISP, other "applicative" languages such as GEDANKEN [Reynolds 70], as well as others emphasizing functional elements, such as APL [Iverson 62], all tend to incorporate the traditional notion that programs transform stores.

1.4 The object level and the function level viewpoints

One important consequence of the conventional concept of "program" is an emphasis on the "object level" view of programming. Since stores hold "objects" in their cells, programming becomes a process of describing how to combine objects to form other objects, a program being a description of how to combine the "input objects", which are found in given cells, to form the desired "output objects" and deposit them in the proper output cells.

Even the traditional applicative languages such as pure LISP and ISWIM [Landin 66] emphasize the object level view of programming. Though their programs do not map stores into stores, they do use names for objects and their basic semantics automatically builds a kind of store (during the execution of a program) in which the values of variables mentioned in the program are kept. And, like conventional programs, these programs are primarily concerned with combining objects. For example, consider the following definition of the object-to-object function f in the style of the lambda calculus [Church 41]:

$$f = \lambda x.h(x,g(x)) \ .$$

We are given the functions g and h and we wish to build the function f. We do this at the object level by introducing the object x, forming the object g(x) and then the "result object" h(x,g(x)); we then use the principal program-forming operation of the lambda calculus, lambda abstraction, to abstract the object variable x and convert the result

object into the function f. Thus we see that the object level definition of f does not directly combine the functions g and h from which f is to be built, but instead this definition descends from the function level of g and h to the object level of x, g(x), and h(x,g(x)), then ascends again to the function level of f. This down-then-up-again style is characteristic of the object level approach. (For further discussion of "functional" object level programs see [Backus 81b].)

As we shall see later, there is a "function level" approach to programming that defines a new function in terms of given ones without descending to the object level. For example, the function level definition of f as above would use two program-forming operations to build f from the functions g and h and the identity function; no object or object variable would appear in the definition. In general the function level approach uses program-forming operations (functionals) to combine given functions directly to form the desired new one. ("Constants" in an object level definition are "lifted" to constant-valued functions in the corresponding function level definition.) Object level definitions all correspond to some isomorphic function level one, but there are many function level definitions that have no isomorph at the object level, although of course there is some non-isomorphic object level definition for the defined function.

Of the various "functional" or non-von Neumann approaches to programming, we shall argue that it is this function level approach that offers the best possibility to have a universe of "programs" with a deeper mathematical order than can be found in the universe of von Neumann programs. It is also an approach that departs the farthest from the traditional use of names and variables for objects, hence it is also an approach that will be likely to cause the kind of deep unease I have tried to indicate above.

1.5 Goals of the function level approach to programming

In the area of data types, as in e.g., [Burstall & Goguen 80],

[Guttag & Horning 78], [Thatcher, Wagner & Wright 78], and [Zilles 79], we have already gone far in moving from the object level viewpoint to a function level one. We have moved from focussing on the objects of a data type and on their "structure" to an emphasis on (a) the operations used to build and manipulate those objects and their structure, and on (b) the algebraic properties of these operations as expressed in various "axioms".

The non-von Neumann, function level approach to "programs" seeks to shift our viewpoint similarly, to introduce the kind of order into the universe of "programs" that the abstract, algebraic approach to data types has introduced into the universe of objects, an order represented by axioms or laws about the operations over the given universe (of objects or of "programs"). With data types, we are concerned with objects (data) and with operations on them; with "programs" we are concerned with objects (data), operations on objects (programs), and operations on programs (program-forming operations or PFOs).

For a long time we regarded programs as a kind of yard-goods pieced together by semicolons and begin/ends, just as we used to regard data structures as pieced together by commas and brackets. More recently, in the era of "structured programs", we noticed that semicolons, if-then-elses, and while-do's were "control operations" that served to "structure" programs.

The goal of the function level approach to a concept of "programs" is to move now to an emphasis on the operations (PFOs) used to construct programs and on the algebraic properties of those operations. For example, in this approach if-then-else becomes a program-forming operation, not a "control" operation; it maps three given programs into one new one and it has important algebraic properties with respect to other PFOs, just as "addition" in a ring has important properties with respect to the other ring operation, "multiplication". It is these algebraic properties that make it possible to transform programs from one form to another and to solve equations for programs, just as

it is the properties of the ring operations that enable us to transform ordinary algebraic expressions and to solve equations.

1.6 Incompatibility of the von Neumann concept of "program" and the function level approach to programming

The choice of "stores" as the domain for von Neumann programs has two immediate, harmful consequences for our ability to write general, composable programs (we discuss these in the next section). But it is an indirect consequence of that choice that blocks the use of a function level style in von Neumann programming: the choice of "stores" as the domain of programs limits our choice of program-forming operations, apparently to PFOs lacking the required algebraic properties. (Again, we shall see later why this is so.)

1.7 Requirements and prospects for the function level view of "programs"

We shall show that by enlarging the domain for "programs" beyond that of "stores" that we can form domains for a new concept of "programs" and that, using this new concept, we have a wider choice of program-forming operations from which we can then choose a set of PFOs with a strong algebraic structure.

If we adopt both the new concept of "program" (with its new domain) and the new PFOs that become available with it, then we can move from an object-centered view of programming to a function-centered one (or perhaps a relation-centered one) -- despite the temporary trauma this may cause. Already there are signs that a more profound order can be found in the universe of the new "programs" than we have found in the universe of the old, but the concept and the form of its programs, being new, are very much open to change. Having incubated for twenty years, the non-von Neumann concepts of "program" are just beginning to develop a function level view and "programs" built by PFOs with algebraic structure.

Much work remains to be done before a definite function level or

other extended concept of "program" (with its accompanying methodology and implementations) can be developed and achieve some form of consensus. Finding the "best" concept of "program" and achieving such a consensus would be aided by a larger abstract theory concerning the effect of various representations of programs over various domains on the properties of such systems. Much will depend on our ability to exploit the algebraic structure of PFOs to optimize programs. It will also be important to develop new computer architectures that directly implement both the PFOs and the composite data objects of the new concept and that exploit the inherent parallelism of its programs.

Of course, as with any new approach, we may find difficulties with the function level view that make it unsuitable, but that too can only become clear after a lot of further work.

1.8 Organization of this paper

Section 2 discusses two difficulties associated directly with the choice of "stores" as the domain for programs. Section 3 indicates how the von Neumann concept of "program" can be extended to a non-von Neumann concept by enlarging the domain for programs. It discusses the advantages of this extension; it does so by describing a "typical" von Neumann language called L and its non-von Neumann extension, called L*, and comparing the two. The rest of the paper is best outlined by giving the section and subsection titles:

2. Two fundamental problems with the von Neumann concept of "program"
   - 2.1 Problem domains are not program domains
   - 2.2 The principal program-forming operation, composition, is ineffective for building von Neumann programs

3. The extension of the von Neumann concept of "program" to a simpler, non-von Neumann concept
   - 3.1 Introduction
   - 3.2 Program domains
   - 3.3 Abstract programs and PFOs versus concrete programs and PFOs

2. Two fundamental problems with the von Neumann concept of "program"

2.1 Problem domains are not program domains

Programmers are never approached with the following request: "I have a set of stores of this kind and I want you to write a program that will transform them into stores of this other kind." Instead they are asked, given a set of files and transactions, to write a program to transform a file-transaction pair into a new file and a response. Or they are asked to write a program that inverts matrices, and so on. But "programs" can only map stores into stores.

Thus the primary difficulty with the von Neumann idea of "program" is that the solution of a problem is never a program. For example, there is no program for finding square roots. We tend to think this last statement is wrong because we regard as insignificant the "explanation"

needed to connect a so-called "square root program" with the actual mapping that carries numbers into their square roots. Thus a "square root program", to be meaningful, actually consists of a program (that maps stores into stores) plus a storage plan, the "explanation", without which the program proper is useless. If the program takes its input from, say, cell a and deposits its output in cell b of the store, then the storage plan is a statement of these facts. To compute the square root of some number n then requires two other mappings in addition to that of the program itself: an input mapping that creates a store with n in cell a and an output mapping that maps a store into the contents of cell b; then the composition output∘program∘input is a function that maps numbers into their square roots, where "input" and "output" depend on the storage plan of "program". (In practice the use of a square root program as a subroutine employs mechanisms to conform the storage plan of the program to that of its context or vice versa, according as it is called by name or by value.)

The choice of "stores" as the domain of von Neumann programs means that a typical program has a "purpose", that is, to map some domain D into another domain E (possibly the same), which it can achieve only partially and indirectly; it can be interpreted as accomplishing its "purpose" only by a mental transformation that requires full knowledge of its storage plan. This storage plan constitutes a kind of artificial representation of the domains of the program's "purpose", D and E, in terms of stores.

The representation problem that is intrinsic in the von Neumann approach, representing problem data by stores, greatly complicates programming by interposing storage plans between the straightforward "purpose" of a program and the program itself. This makes it impossible for a von Neumann program to achieve its "purpose" directly.

2.2 The principal program-forming operation, composition, is ineffective for building von Neumann programs

The von Neumann requirement for representing data by stores is also

the main reason we have found it so difficult to effectively build new programs from existing ones. The principal program-forming operation for building programs is composition, thus from programs p and q we can form the new program p;q . But if p and q are von Neumann programs written independently, then p;q is almost certain to be meaningless. That is, the program p;q will not achieve the "purpose" of p followed by the "purpose" of q (even in the indirect sense in which programs achieve their "purpose") except in the unlikely event that q happens to take its inputs from just those cells in which p places its outputs.

Thus the von Neumann concept of "program" assures that composition is useful only for piecing together programs that are written together under a unified storage plan, whereas in mathematics and in a proper universe of "programs", composition is the primary, most powerful operation for building functions or programs.

Some readers will complain that the above remarks about composition fail to take into account the existence of procedure declarations, since these enable one to make specific programs into general ones whose storage plans can be varied. Suppose one wishes to compose two procedure calls so as to obtain the composition of their "purposes". If they are P(x,y,z,w) and Q(x,y,z,w), where in each case x and y are inputs and z and w are outputs, then P(a,b,c,d);Q(c,d,e,f) transforms a and b into e and f. Here we have merely simplified the storage planning problem: that of designing two programs to be composed so that the storage plan for the first is appropriate for that of the second. For ordinary programs the "store" they operate on is large and hence storage plans involve much detail; in the case of our procedure calls their "stores" have effectively only four "cells" that require planning (the other, local cells being isolated from all other cells in the basic store). Thus P(a,b,c,d) becomes an actual program only after a, b, c, and d are chosen; once chosen the chance that P(a,b,c,d) is meaningfully composable with another independent program is as slight as ever.

Thus by the use of procedure declarations we make it possible to plan storage at the time of use, rather than the time of writing a procedure. This means that storage plans can be more local and flexible when using procedure calls in place of "programs". But that still

means that p;q is almost always meaningless unless p and q have a common storage plan. Compare this situation with the composition of functions in mathematics. In that context a function and its "purpose" are the same: a function maps its intended domain directly into its intended range without any intervening representations. Thus the composition of two functions is always meaningful if the composition of their "purposes" is meaningful.

The contrast between composition of von Neumann programs and that of functions is perhaps best seen by an example. If feet-inch is a program to convert feet to inches and inch-yard is one to convert inches to yards, then

feet-inch ; inch-yard

is a program, but it will almost certainly not convert feet to yards. On the other hand if the same names denote two conversion functions, then

inch-yard ∘ feet-inch

is a function that will convert feet to yards. The difference follows from the fact that the program feet-inch maps stores into stores whereas the function feet-inch maps numbers (in feet) into numbers (in inches).

Of course von Neumann languages provide functions for use within expressions, and these can be composed meaningfully within individual expressions. Here we are addressing the problem of building programs from pre-existing ones by composition, a more central issue.

3. The extension of the von Neumann concept of "program" to a simpler, non-von Neumann concept

3.1 Introduction

We have considered two important defects in the von Neumann concept of "program" that are direct consequences of choosing "stores" as the domain for programs. These defects can be understood without examining

the structure of "programs" determined by the operations used to build them. But the most fundamental difficulties of von Neumann programs and the languages used to express them lie in their unnecessarily complex structure, in our inability to use a powerful set of program-forming operations to build programs, and in the fact that the PFOs that we do use lack the algebraic properties that would allow us to prove useful general theorems about large classes of programs.

In order to illustrate these difficulties we propose to describe the elements of a typical, but oversimplified von Neumann language, L. Since our purpose is to expose defects in the simplest conventional concepts of "program", it will not be necessary to include in L many of the complications that appear in real languages. That complexity is very well illustrated in the descriptions of real languages in the literature of denotational semantics [Stoy 77], [Tennent 76]. It will become apparent that the defects we discuss in L can only become worse with the addition of further features.

The complexity of real languages revealed in denotational semantics can be viewed in two different ways. The notion of "continuations", for example, can be viewed as an ingenious invention that makes it possible to cope formally with various features of conventional languages. On the other hand, continuations can be regarded as yet one more level of mathematical obfuscation and complexity that is essential to shore up a failing concept of "program", a concept tottering under its own weight.

In the following we shall briefly examine the structure of L-programs in terms of the operations used to build them and to build their components. We shall then describe a non-von Neumann language L* that is an extension of L in the sense that (a) the domain for L*-programs contains that for L-programs and (b) every L-program has an exact image within a small subset of the programs of L*.

Up to this point we have emphasized the association of the concept of "program" with the domain that programs operate on. In the following we shall give equal emphasis to the second, independent

element that is central to the concept: the program-forming operations used to construct "programs" and the properties of those PFOs.

We shall see that L*-programs can be built with a set of PFOs different from that used to build L-programs, that these PFOs have a strong algebraic structure where those of L do not, and that L*-programs have a simpler structure than L-programs. We shall see that L*-programs represent a function level approach to programming and L-programs an object level one. But first we must discuss a few issues that have been dealt with only vaguely, and then describe L and L*. We will then be able to compare the two approaches in more detail.

3.2 Program domains

By definition we have assumed that "programs" map some domain D into itself. We do so because, if "programs" mapped D into E, then, unless E is contained in D, it would not make sense to compose two such programs.

3.3 Abstract programs and PFOs versus concrete programs and PFOs

"Programs" can be regarded either as mappings (infinite entities) or as representations (finite expressions) for such mappings, that is, as "abstract" or "concrete" programs. In our discussions of "expressions" in L (store-to-object mappings) and of programs in L and L* we do not want to constantly distinguish between their abstract (mapping) and concrete (representation) aspects. Even more, we want to speak of expression-forming and program-forming operations without saying whether we mean operations that form abstract expressions or programs, or that form concrete ones. We sketch informally how this can be done without confusion.

Let us consider only programs, since the treatment of expressions is similar. Let us agree that all concrete programs that represent one abstract program are "equivalent" and that when we speak of a "program" we may be referring either to the abstract program (mapping)

or to one of the concrete programs that represent it or to the entire equivalence class of concrete programs. This gives us, at least, a clear understanding of the ambiguity we shall allow ourselves in the use of the term "program".

Now consider an abstract program-forming operation π that builds, for example, the abstract program $\pi(p,q)$ from the abstract programs p and q. Then there is a corresponding concrete program-forming operation $\tilde{\pi}$ such that, for any concrete program-representations $\tilde{p}$ and $\tilde{q}$ for p and q, then $\tilde{\pi}(\tilde{p},\tilde{q})$ builds a concrete program that represents $\pi(p,q)$. Of course, concrete program-forming operations must be blind to the particular choice of concrete program $\tilde{p}$ used to represent some abstract p; replacing an argument of a concrete PFO by an equivalent one must give equivalent results. Again, for each abstract PFO π there are many corresponding ("equivalent") concrete PFOs $\tilde{\pi}$ that differ in the representations they produce for an abstract program.

As with the term "program", we shall use the term "program-forming operation" or "PFO" to mean either some abstract PFO π or some corresponding concrete PFO $\tilde{\pi}$ or the entire equivalence class of concrete PFOs that correspond to π.

3.4 Algebraic structure of a set of operations

We have referred to a set of operations as having a "strong algebraic structure"; although we do not intend the phrase to have a precise meaning, it deserves some explanation. To the extent that pairs or tuples of operations in the set are related by algebraic laws, we think of the set as being structured by those laws. Thus the pair of operations, addition and multiplication, has a strong algebraic structure since the only pair in the set is related by the distributive law $(a+b)c = ac + bc$, which expresses a multiplication involving addition as an addition involving multiplications. It is this kind of law, with this kind of "symmetry" that has the greatest intuitive "strength" in our notion of algebraic stucture. Thus, for example, the recursive definition of while-do using condition and composition lacks this "symmetry" and hence is not as "strong" a law.

The intent of our notion of algebraic structure of a set of operations is that strong structure goes with strong theorems about the operations and their domains and weak structure with weak theorems. We do not pretend that the notion is precise or that it always achieves this intent. In the case of the set comprising addition and multiplication our informal notion does work: it has a strong algebraic structure and there are strong theorems about the operations and their domain of numbers, e.g., the general solution of quadratic equations.

A similar example, but with weak structure, is the set of operations of addition and square root. Without the introduction of multiplication there are no strong laws relating addition and square root, therefore there are no strong, general theorems about systems with just these two operations, only particular ones like sqrt(4) + sqrt(9) = 5.

I believe it will turn out that the reason we find so few strong, general theorems about conventional programs is that their program-forming operations have a weak algebraic structure. We shall see that the PFOs of L have a weak structure; but those of L* have a strong structure and, in accordance with the above notion, there are beginning to emerge some general theorems about programs of this new kind (see [Backus 78], [Backus 81a], [Kieburtz & Shultis 81], [Williams 80]) and I believe we shall see more theorems and stronger ones. We shall examine the algebraic strucure of the PFOs of both L and L* in more detail later on.

3.5 The von Neumann language L

Our typical but oversimplified von Neumann language L has the following elements:

(1) A set of L-programs that map L-stores into L-stores.

(2) A set of L-stores, each store being a set of cells, each cell having a "name" and "contents".

(3) A set of L-objects that includes all the "names" and "contents" found in any L-store. We assume that the set of L-objects is a rich one containing objects built from elementary objects such as "true", "false", numbers, symbols, etc., by constructions such as sequences, arrays, stacks, files, etc.

(4) A set of L-expressions that map L-stores into L-objects.

(5) A set of object-forming operations such as +, square root, as well as more complex operations on more complex objects.

(6) A set of program-forming operations that build "structured" L-programs from L-expressions and L-programs. These are: composition (semicolon), if-then-else, and while-do.

3.6 The structure of L-programs

L-programs are built on three planes. On the highest plane are L-programs; these are built from elementary L-programs by PFOs. On the middle plane are the elementary L-programs (assignments); these are built from "names" and L-expressions by the "assignment-forming operation". On the lowest plane are L-expressions; these are built by object-forming operations (actually, by operations isomorphic to these) from elementary L-expressions, which are L-objects and L-variables. L-programs map L-stores into L-stores. L-expressions map L-stores into L-objects.

3.6.1 Elementary L-expressions map stores into objects. An L-object, serving as an L-expression, maps any store into the object itself. Thus 3 maps any store s into 3 (we write 3:s = 3). An L-variable $\hat{n}$ (associated with the "name" n) maps a store s into the contents, in s, of cell n ($\hat{n}$:s = contents cell n in s). (We do not deal with the question of subscripted variables or other compound cell names, nor with the necessary distinction between "names" (objects n) and "variables" (expressions $\hat{n}$).)

3.6.2 Constructing L-expressions with object-forming operations. Actually, expressions are built with expression-forming operations that are derived from object-forming ones as follows. If $o(x_1,\ldots,x_n)$ is an object-forming operation of L, then $o_e(e_1,\ldots,e_n)$ is the corresponding expression-forming one, where the store-to-object mapping of the latter is

$$o_e(e_1,\ldots,e_n):s = o(e_1:s,\ldots,e_n:s)$$

for any store s. For example $+_e$ builds an expression from the expressions a and b, where $(a +_e b):s = a:s + b:s$ for any store s.

Having noted the isomorphic relationship between the object-forming and expression-forming operations of L, from here on we can pretend they are identical.

3.6.2 Properties of L-expressions. The points to notice about expressions are these:

(a) They are object level constructions describing the combination of objects to produce others.
(b) They represent the principal "work" of a program.
(c) Laws concerning object-forming operations are suitable for proving facts about objects, whereas we often want to prove facts about programs rather than objects.
(d) Expressions cannot be built by composition since their domain (stores) and range (objects) differ.

3.6.3 Elementary L-programs are assignments; these are built by an assignment-forming operation, := , from a name n and an expression e, yielding the assignment n:=e . The store-to-store mapping of this assignment satisfies the following two equations:

(1) $\hat{n}:s' = e:s$

(2) $\hat{n}':s' = \hat{n}':s$ for all $n' \neq n$,
where $s' = (n:=e):s$

for all stores s. Thus (1) asserts that the contents of cell n in s' (obtained from s by "executing" the assignment) is the value of e with respect to s, and (2) asserts that the contents of other cells in s' are the same as in s.

3.6.4 Constructing L-programs with PFOs. The PFOs of L are composition (;), if-then-else and while-do; these are used to build programs from elementary ones. Thus if p and q are L-programs and b an L-expression, then

p;q
if b then p else q
while b do p

are L-programs. Note that here we are emphasizing the operational character of PFOs as operations on program-mappings. For example, if-then-else is not to be seen as a kind of punctuation used to divide up the text of if b then p else q into the subtexts of b, p and q. Instead, we see it as an operation with three operands: b, a mapping of stores into objects, and p and q, mappings of stores into stores. The result of applying if-then-else to these operands is a store-to-store mapping we denote by if b then p else q.

Of course, as discussed earlier, there are many equivalent concrete-program-forming operations for if-then-else that map three concrete representations for the mappings b, p, q into some concrete representation for the mapping if b then p else q; as agreed earlier we have lumped the one abstract PFO and the many concrete PFOs for if-then-else into the one term "if-then-else". For this "lumping" to be valid one must check that, if one equivalent concrete program is substituted for another in a program built by a PFO, then the resulting program will be equivalent to the original. This will be true for any PFO if every program built by it only applies the constituent programs in obtaining its result, since equivalent programs are indistinguishable in that case. Every PFO we shall use has this property.

The mappings associated with each of the PFOs of L are given by the following definitions, for all stores s:

composition (p;q):s = q:(p:s)

if-then-else (if b then p else q):s = p:s if b:s = "true"
= q:s if b:s = "false"
= undefined otherwise

while-do (while b do p):s = s if b:s = "false"
= (while b do p):(p:s) if b:s = "true"
= undefined otherwise

3.6.5 Properties of L-programs. The main discussion of properties of L-programs is best left until we have described L*-programs and can compare the two. For the present the main thing to note about L-programs is their three-plane construction that divides them into expressions (that do most of the work and whose object-forming operations may have good algebraic properties), assignments (that are built from expressions and change one cell of a store), and programs (that are built from assignments). Thus the power and possible algebraic elegance to be found in L-programs is confined to expressions whose individual effect in a program is restricted to changing one cell. (We shall examine the algebraic structure of the PFOs of L later on.)

3.7 An extension of L: the non-von Neumann language L*

The language L* has the following elements:

(1) A set of L*-programs that map L*-objects into L*-objects.

(2) A set of L*-objects (that contain all L-stores and L-objects as described below).

(3) A set of program-forming operations.

The set of L*-objects contains all L-stores and all L-objects and hence every "name" and "contents" found in any L-store. Furthermore, the set of L*-objects is closed under sequence-formation; thus if $x_1,\ldots,x_n$ are L*-objects then so is the sequence $\langle x_1,\ldots,x_n\rangle$. We shall see that enlarging the domain for "programs" from "stores" as in L to the domain of L*-objects results in a surprising simplification in the concept of "program" (already it is evident that L* has fewer elements than L).

3.8 The strucure of L*-programs

L*-programs are built on one plane; they are built from elementary L*-programs by PFOs.

3.8.1 Elementary L*-programs. These are the given, primitive programs of L*. They, and all L*-programs, have a single argument; they are functions from L*-objects into L*-objects. They include the object-forming operations of L, such as + and square root, and various functions and predicates for accessing, rearranging and testing sequences, as well as functions for dealing with whatever special data types are included in the set of L*-objects. For example, +:<3,4> = 7, null:<> = "true", equal:<A,B> = "false", length:<A,B,C> = 3 . Note that A and B are simply objects, they do not name other objects.

3.8.2 Constructing L*-programs with PFOs. Having chosen a domain for programs does not determine the program-forming operations that can be used to build them. In L we have used the traditional PFOs for "structured" programs. However, since L*-programs have a larger domain and a simpler structure than L-programs, we shall use a somewhat different set of PFOs to build L*programs. We want this new set to have a strong algebraic structure; as it turns out the following three PFOs for L* have such a structure (we describe for each one the program it constructs in terms of its argument-programs):

composition builds the program p∘q from programs p and q, where

p∘q:x = p:(q:x) for all L*-objects x.

condition builds the program $p \rightarrow q;r$ from the programs p, q and r, where

$$(p \rightarrow q;r):x = q:x \quad \text{if } p:x = \text{"true"}$$
$$= r:x \quad \text{if } P:x = \text{"false"} \qquad \text{for all L*-objects x.}$$
$$= \text{undefined otherwise}$$

construction builds the program $[p_1,\ldots,p_n]$ from programs $p_1,\ldots,p_n$, where

$$[p_1,\ldots,p_n]:x = \langle p_1:x,\ldots,p_n:x\rangle \qquad \text{for all L*-objects x.}$$

Composition is essentially the same as the PFO used in L except for its domain, notation, and the order of its arguments. Condition is slightly different from the if-then-else of L in that all three arguments are L*-programs; this difference improves its algebraic relationship to composition. Construction is entirely new; in fact, it cannot be used in L since the mapping it builds from L-programs would map a store into a sequence of stores, and a sequence of stores is never a store, hence this mapping cannot be an L-program and construction cannot be used to build L-programs.

A fourth program-forming operation that is essential in L* is "constant", one that is different in that it builds a program from an object.

constant builds the program $\bar{x}$ from the object x, where

$$\bar{x}:y = x \qquad \text{for all L*-objects y.}$$

The reader may wonder at this point why there is no PFO in L* analogous to while-do. Of course we could introduce one without difficulty. But the three principal PFOs above have a strong algebraic structure whereas there are no strong algebraic laws relating while-do to these (other than the function level defining equation for while-do, which relates it to composition and condition, a "weak" law). Furthermore, the algebraic structure of the principal PFOs allows us to formally solve and reason about a much larger class of recursive equations than the class of tail recursive ones for which while-do represents solutions (see [Backus 81a], [Williams 80] for a discussion of such formal solutions).

Since recursive equations comprise a more powerful and expressive way of defining programs than while-do, we shall allow them as program definitions in L*; thus we have no need for while-do and can retain the strong algebraic structure of the PFOs of L*.

One of the major benefits of the L* approach is that one can use a great many operations for building programs in L*. Their analogues can be used in L only to build object-forming operations, operations that are then used to build L-expressions. Adopting this approach introduces a fourth plane into the structure of L-programs: (1) build object-forming operations from elementary ones using "operation-forming operations", (2) build expressions from object-forming operations, (3) build assignments from variables and expressions, and (4) build programs from assignments with PFOs. This structure is found in APL [Iverson 62].

A few other PFOs we might use in L* are the following.

insert builds /p from p, where

$$/p:\langle x_1\rangle = x_1$$
$$/p:\langle x_1,x_2,\ldots,x_n\rangle = p:\langle x_1, /p:\langle x_2,\ldots,x_n\rangle\rangle$$

apply-to-all builds αp from p, where

$$\alpha p:\langle x_1,\ldots,x_n\rangle = \langle p:x_1,\ldots,p:x_n\rangle$$

fetch builds $\uparrow x$ from the object x where

$\uparrow x:s$ = contents of cell x in store s for any store s

store builds $\downarrow x$ from object x where

$\downarrow x:\langle y,s\rangle = s'$
where s' is s with contents of cell x now equal y.

3.9 L*-images of L-expressions and L-programs

By the "image" $\tilde{p}$ in L* of an L-program or an L-expression p we mean the L*-program $\tilde{p}$ such that $\tilde{p}:s = p:s$ for all L-stores s; we shall not be concerned whether $\tilde{p}$ is minimal, i.e., undefined for all non-stores. Because of the defects of L-programs, their images in L* are perhaps the least interesting and least useful programs in L*. But it may be of interest to some readers, as an exercise in programming in L*, to see how expressions and programs of L can be built (as programs) in L* using its different set of PFOs. (This section is not essential to understanding later ones and may be skipped.)

3.9.1 Elementary L-expressions. We illustrate images by examples and consider the L*-images of the two kinds of elementary L-expressions, that of the L-object 3 and that of the L-variable $\hat{v}$. The L*-image of 3 is $\bar{3}$; if we take v to be an L*-object, then the L*-image of $\hat{v}$ is $\uparrow v$. To back up this claim we must show that each entity and its image is, one in L and the other in L*, the same mapping of stores into objects:

$3:s = 3$ in L

$\bar{3}:s = 3$ in L*

$\hat{v}:s$ = contents of cell v of s, in L

$\uparrow v:s$ = contents of cell v of s, in L* (see the PFO fetch) .

3.9.2 Composite L-expressions. If a and b are L-expressions and $\tilde{a}$ and $\tilde{b}$ are their L*-images, then $+\circ[\tilde{a},\tilde{b}]$ is the L*-image of the L-expression $a +_e b$; $sqrt\circ\tilde{a}$ is the image of $sqrt_e(a)$, since, for all stores s:

$(a +_e b):s = a:s + b:s$ in L

$+\circ[\tilde{a},\tilde{b}]:s = +:\langle\tilde{a}:s,\tilde{b}:s\rangle = a:s + b:s$ in L*

$sqrt_e(a):s = sqrt(a:s)$ in L

$sqrt\circ\tilde{a}:s = sqrt:(\tilde{a}:s) = sqrt(a:s)$ in L*

3.9.3 Elementary L-programs (assignments). The L*-image of v:=e is $\downarrow v \circ [\tilde{e}, id]$, where $\tilde{e}$ is the image of e and id is the identity function:

(v:=e):s = s' in L, where contents of cell v in s' is e:s.

$\downarrow v \circ [\tilde{e}, id]:s = \downarrow v:\langle \tilde{e}:s, s \rangle = s'$ in L*, where s' is the same as above, since $\tilde{e}:s = e:s$. (see PFO "store")

3.9.4 Composite L-programs. The L*-image of p;q in L is $\tilde{q} \circ \tilde{p}$. The L*-image of if b then p else q is $\tilde{b} \rightarrow \tilde{p}; \tilde{q}$. The L*-image of while b do p is the solution f of the equation

$$f = \tilde{b} \rightarrow f \circ \tilde{p};\ id \quad .$$

(To apply f as defined above to any object, apply the right side.) I leave it to the interested reader to verify that these L*-images represent the same store-to-store mappings as the original L-programs.

3.10 The algebraic structure of the PFOs of L* and L

The principal three PFOs of L* have a strong algebraic structure as shown by the following "strong" laws that relate each pair (with two laws for one of the pairs). For all programs p, f, g, h, $f_1, \ldots, f_n$ the following function level identities hold:

Composition and condition

(1) $f \circ (p \rightarrow g;\ h) = p \rightarrow f \circ g;\ f \circ h$

(2) $(p \rightarrow g;\ h) \circ f = p \circ f \rightarrow g \circ f;\ h \circ f$

Composition and construction

(3) $[f_1, \ldots, f_n] \circ h = [f_1 \circ h, \ldots, f_n \circ h]$

Construction and condition

(4) $[\ldots(p \rightarrow g; h)\ldots] = p \rightarrow [\ldots g \ldots];\ [\ldots h \ldots]$

(4) holds either in the domain for which p is boolean-valued or always if the sequence constructor is strict.

Each of the above laws relates two PFOs, call them A and B, where each law expresses a program built by A (involving a program built by B) as an equivalent program built by B (involving programs built by A).

Many other algebraic laws hold in L*, some relating other PFOs to the principal ones and each other, and others that involve primitive L*-programs. But most of these laws are less symmetric, "weaker" than those above. For example, the law relating the PFOs insert, composition and construction,

$$/f\circ[g_1,g_2,\ldots,g_n] = f\circ[g_1,/f\circ[g_2,\ldots,g_n]]$$

is a "function level" version of the second part of the object level description of /f given earlier under the PFO insert.

Since any abstract L*-program can be represented by programs built from suitable primitive programs by the three principal PFOs and recursive equations, the strength of their algebraic structure indicates that there should be a lot of strong general theorems involving these PFOs whose universally quantified variables range over L*-programs.

Now consider the PFOs of L. They satisfy only one symmetric law, which relates composition and if-then-else:

$$(\underline{\text{if}}\ b\ \underline{\text{then}}\ p\ \underline{\text{else}}\ q);r = \underline{\text{if}}\ b\ \underline{\text{then}}\ (p;r)\ \underline{\text{else}}\ (q;r)\ .$$

This is the analogue of (1); the analogue of (2) fails because one cannot compose a program and an expression to form an expression (even if this is allowed there are other complications). There are no symmetric laws relating while-do with either composition or if-then-else (the function level definition of while-do is a "weak" law that relates all three PFOs of L).

The weak algebraic structure of the PFOs of L is consistent with the existence of few general theorems about the programs of L; instead we tend to find theorems about particular programs ("my program is correct") or small classes of programs ("this program, where o is any associative operation, is equivalent to that one"). One can (and should) ask whether there are better sets of PFOs for building L-programs. The answer is unclear; at this point all we can say is that construction, which relates well to composition and condition (which is close to if-then-else) to form a strongly algebraic triad, cannot be used as a PFO for any programs that have "stores" as their domain. Since it is hard to do without the PFOs for composition and condition, or something similar, this does not bode well for finding PFOs for L with a good algebraic structure.

Not only does L* have a strong algebraic structure in its major PFOs but also it has, as noted above, weaker laws relating these to lesser PFOs such as insert and apply-to-all, and to primitive L*-programs, such as those that select the nth element of a sequence or rearrange various data structures. These lesser PFOs and primitive L*-programs could only be used within expressions of L, therefore theorems relating all of these elements would be blocked in L by the barrier between programs and expressions. In L*, on the other hand, we can expect to obtain theorems that interrelate both major elements of a program-scheme (corresponding to the program plane in L) and minor elements (corresponding to the expression plane in L), since all these are programs in L* and are all put together by PFOs that are related by algebraic laws, either weak or strong.

4. Comparison of L and L*

4.1 Problem domains and program domains

We noted earlier that L-programs never solve problems directly since they are store-to-store mappings and real problems require other kinds of mappings. Provided the data types of a problem are

in the set of L*-objects, there is an L*-program that is a rather direct solution. Thus there are L*-programs for square root, matrix inversion and file updating that do not require storage plans to be useable, programs that map numbers into their square roots, matrices into their inverses, and so on.

4.2 Composition of Programs

Again we observed earlier that in L the composition p;q of two independent programs has little chance of achieving a meaningful program that represents the composite purpose of p and q. Thus if the purpose of p is to transform A's into B's, and that of q is to transform B's into C's, then p;q will almost certainly not transform A's into C's unless p and q have a common storage plan.

On the other hand, if p and q are L*-programs for the same purposes, then the results of p will be B's, as will the arguments of q, and $q \circ p$ is an L*-program to map A's into C's.

4.3 Relationship between a composite program and its subprograms

We have observed that composition cannot assemble independent programs in L into a meaningful composite program. That observation applies equally well to the other two PFOs of L, if-then-else and while-do. If we wish to use if-then-else to build a program from expression b and programs p and q, then p and q must, in most cases, use corresponding input and output cells, otherwise the composite program is likely to be meaningless. Thus again p and q must have a common storage plan.

In contrast, in L* the PFO condition easily assembles appropriate independent programs into a meaningful composite. For example, if b tests whether its argument is an A or a B, and p maps A's into C's and q maps B's into C's, then $b \rightarrow p;q$ is a program that maps $A \cup B$ into C in a way that is evident from its structure.

Again in the case of the program while b do p the expression b and the program p must have a common storage plan for the composite to be meaningful. The recursive definition in L* corresponding to while-do can, like the other PFOs of L*, easily combine its constituent programs into a meaningful composite.

The PFOs of L* that are not in L also have this important ability to create meaningful programs from existing, independent programs. Thus, for example, from programs p, q and r, all defined on the domain A, the PFO construction builds the program [p,q,r] that maps A into $B \times C \times D$, where B, C, D are the ranges of p, q, r.

It is important to note that the inability of the PFOs of L to assemble independent programs into meaningful new ones comes from the choice of "stores" as the domain for its programs and the barrier this poses between the purpose of a program and what it actually does: map stores into stores. In L*, on the other hand, there is no such barrier; if the purpose of a program is to map A's into B's, then that is what it does. (Within L* there are "von Neumann programs" that map stores into stores, but we do not have to use them.)

The most important property of any system of programming is its ability to build new programs from existing ones at any level. The lack of this ability is the primary weakness of L and the von Neumann concept of "program"; having this ability is one of the primary strengths of the L* concept of "program".

4.4 Complexity of program structure and of language structure

It is obvious that L*-programs have a much simpler structure than L-programs. The latter have a three-plane structure (expressions, assignments -- the interface between expressions and programs, and programs), each plane having its own entity-forming operation(s). (The structure of real von Neumann programs is generally more complex than those of L and their domain is usually more complex than "stores".)

L*-programs have a one-plane structure; they are built from primitive ones by PFOs.

The term "structured programming language" is often applied to languages with PFOs like those of L in which the use of GOTOs is restricted (if it is not, it is hard to define the effect of PFOs). This use of the term is trivial and misleading: if programs are built by any PFOs at all, then they are "structured" by those PFOs, the only "non-structured" programs being those that are not built exclusively by well-defined PFOs. But the fact that a language uses PFOs to build programs does not mean that the language is structured, only its programs are.

If programs are "structured" by the way they are built by PFOs, then what is a reasonable notion of "structure" for a language? Since the PFOs of a language are one of its principal elements and since PFOs may be "structured" by their interrelating algebraic laws, I propose that a programming language should be considered "structured" to the extent that its PFOs have an algebraic structure. In this sense L and other von Neumann languages are very weakly structured, whereas the language L* is strongly structured.

4.5 The relationship between languages and machines, serial versus parallel

As outlined earlier, languages like L evolved from the von Neumann computer and its machine language; this is the basic reason behind the choice of "stores" as the domain for its programs. Therefore there is a relatively small "distance" between L-programs and machine programs. Hence it is relatively easy to (a) convert one into the other, (b) project a notion of efficiency from one to the other, and (c) retain the efficiency of one in converting it into the other. But, like their machine counterparts, L-programs are hard to transform, to optimize, and to reason about.

The "distance" between L*-programs and von Neumann machine programs is relatively large and therefore it is more difficult to (a) convert one to the other, and (b) project notions of efficiency onto L*-programs.

The symbiosis between conventional programs and von Neumann machine architecture, in which each needs the other, has kept the concepts of "program" and "computer" all too static over the last thirty years. Originally the von Neumann concept of machine design was an elegant and beautiful one that matched, in a design of great economy, the economics of circuitry and the object level ideas about programming that were then current. Those basic circuit economics persisted over several generations of new circuitry, so there was little pressure for radical changes in machine design. Now, however, VLSI circuitry has changed those economics and the von Neumann design does not seem able to exploit the new economics nearly as well as it did the old.

At the same time, VLSI is making computers so cheap that programming costs are becoming far greater than equipment costs. Ever larger and more complicated von Neumann languages have been produced in response to the resulting pressure to reduce programming costs, but they have not succeeded in making a satisfactory reduction.

Thus there are twin pressures to find radical new designs that exploit VLSI better and to find radical new languages that reduce the cost of programming so that cheap machines can be cheaply and conveniently programmed. It may be that non-von Neumann languages such as L* and others can offer some help in both areas.

There is now a race underway to find new architectures to exploit VLSI to the full. One of the main concerns is to achieve a high degree of parallel operation and to do so without paying the penalty (paid by some earlier parallel designs) of greatly increasing the already soaring costs of programming. On the contrary, it is vital to reduce programming costs drastically while at the same time exploiting VLSI. This will require finding parallel designs that are relatively "close" to new non-von Neumann concepts of "program" that appear to offer greater programming power.

Therefore many machine designers are studying non-von Neumann languages and attempting to find designs that correspond, that minimize the "distance" between their design and their chosen language model.

Some are looking at "object level" functional languages, such as LISP, others at "function level" ones, such as FP [Backus 78] or L* (plus other language models that are hard to classify). (Here is a small sample of papers on representative machine designs: [Arvind, Gostelow & Plouffe 78], [Darlington & Reeve 81], [Dennis, Leung & Misunas 79], [Holloway, Steele, Sussman & Bell 80], [Keller, Lindstrom & Patil 78], and [Mago 79].)

The resulting machine designs vary widely and, until their economics are better understood, we can only wait to see what their cost-performance turns out to be and what the "distance" is between each design and the various concepts of "program". (Of course that "distance" is short for some designs built around specific languages, e.g., the "Scheme-79" machine and the Scheme variant of LISP [Holloway et al], Mago's machine and FP.)

One of the principal challenges in designing machines based on any of the non-von Neumann languages is to find economical machine techniques to store, manage and operate on data having a hierarchical structure (such as lists or sequences). Another challenge in designing machines based on function level languages like L* is to implement a variety of PFOs in hardware, thereby making the machine language "higher level" in some sense than today's "higher level languages", and thereby helping to make programs for such machines easier and cheaper to produce.

In the effort to find machine designs for parallel operation, languages like L are poor guides. Basically this is because all of its PFOs combine programs in a serial fashion, and having "stores" as the domain for programs greatly complicates the problems of introducing parallel PFOs. In contrast, the one PFO of L* that cannot be used in L, construction, is the one that combines L*-programs in parallel. Thus [p,q,r] is a program all of whose subprograms, p, q, and r, can be applied in parallel to its argument. Other PFOs can be used in L* that introduce parallel operations, such as "tree", which serves the same purpose as "insert" [Williams 81]. Thus L* allows a

programmer to naturally express parallel operation where that is called for.

4.6 Object level and function level programs

The essence of an object level definition of a function or program is the description, for every possible set of input objects, of how to build a succession of objects (by applying given object-forming operations or given programs) until the desired result-objects have been constructed. But what are the "ingredients" from which we build some desired program? They are not the objects we construct in an object level definition. They are in fact the object-forming operations and programs we use in laboriously building the succession of objects that culminates in the "results".

In contrast, the function level approach to defining a program is to build it directly from the given "ingredients", the given operations and programs that must be used in either an object level or a function level definition. Instead of applying the given operations to objects, a function level definition applies functionals (PFOs) to the given operations; instead of building a succession of objects to obtain the "result-objects", it builds a succession of programs to produce the desired program.

For example, consider the problem of defining a program that transforms x into sqrt(x) + square(x), given the object-forming operations (in L) or programs (in L*) for +, sqrt, and square. A program in L is

(1) y := sqrt(x) + square(x) .

If this object level program is applied to a store s in which the input number is in cell x, then the result is a store s' in which the result is in cell y. The expression on the right causes sqrt and square to be applied to the number x:s, giving two intermediate objects that are added to form the result-object.

The corresponding L*-program is

(2) $+\circ[sqrt, square]$.

This program maps numbers directly into the desired result. When it is used it will construct the same intermediate and final results that (1) does:

$$+\circ[sqrt, square]:x = +:<sqrt:x, square:x>$$.

But (1) and (2) differ in that (1) applies sqrt and square to the ("abstract") object x, giving two objects, whereas (2) applies the functional "construction" to the programs sqrt and square, giving a program, [sqrt, square], and so on.

While L-programs focus on combining objects, L*-programs focus on combining programs. Thus L-programs draw attention to object-forming operations whereas L*-programs draw attention to PFOs. The algebraic properties of object-forming operations are the basis for general theorems about objects, whereas the algebraic properties of PFOs are the basis for general theorems about programs.

Thus if we want to have a concept of "program" in which the set of programs themselves, together with some set of PFOs, form an elegant mathematical space for which there are strong, interesting theorems, then it behooves us to pursue a function level approach to the concept of "program", the approach that is founded on just this viewpoint.

5. Conclusions

Powerful forces now threaten the von Neumann concept of "program". The ever greater need to reduce programming costs and the continuing failure of ever more gigantic von Neumann languages to bring about significant reductions. The drive to find non-von Neumann architectures that better exploit VLSI. These forces will continue to grow.

It is unclear what new concept of "program" will emerge in response to these forces, but I think it will become increasingly clear that the answer to the question in the title of this paper is "yes", that we, computer scientists, need to work hard to develop a new concept of "program". And if we are to succeed we need to be aware of and free ourselves from the psychological barriers, the ancient traditions of language that keep us trying to modify the von Neumann concept based on "stores" rather than develop something new.

The new concept of "program" that finally is developed may not turn out to be of the function level kind that I have tried to sketch by describing L*, but I believe the L* approach does at least bring out several properties that will be important in any new concept of "program". First, and of the most practical and immediate importance, is the ability to combine independent programs to form new, meaningful programs at all levels, and to use a rich set of operations in doing so. This essential element of programming power is just the property that the von Neumann concept of "program" lacks, and it is the one that the L* approach is designed to provide.

But perhaps the most important property for the success of the new concept is that it should make possible an elegant and powerful mathematics of "programs". Just as numbers, under the operations of addition and multiplication, form a mathematical system called a ring, so should "programs", under their program-forming operations, form a mathematical system of a similar kind. Just as there are hundreds of important theorems about numbers and about the solutions of equations in the ring of numbers, so there should be hundreds of important theorems about "programs" and about the solutions of equations in the mathematical system of "programs". Just as theorems about numbers and their equations represent a great body of deep understanding and knowledge that saves an immense amount of work for mathematicians, so should theorems about the mathematical system of "programs" contain a deep understanding going far beyond what has already been achieved, and save programmers an immense amount of work.

The von Neumann concept of "program" has not given us a powerful

mathematics of its programs with a large body of useful general theorems. The best mathematicians and logicians in computer science still struggle hard to prove (often from first principles) that one single program does what is claimed for it. Often they find it necessary to use elaborate programs to help them. One has only to look at the voluminous formal description of any von Neumann language to realize that its programs do not form a system one can regard as "mathematical" any more than one can regard a system with many pages of axioms as a useful mathematical one.

I believe that the fundamental reason behind the scarcity of general theorems about conventional programs is the lack of algebraic structure of their program-forming operations. The mutual properties of their PFOs probably guarantee the non-existence of such theorems, just as the mutual properties of addition and square root probably guarantee the lack of interesting general theorems about a system of numbers having only these two operations. But whatever the underlying reasons, it seems time to abandon hope that the von Neumann concept of "program" can be the basis of a mathematical system of programs, since twenty years of effort by the best computer scientists have failed to produce the body of general theorems that one expects of such a system.

The work to discover general theorems about function level "programs" of the sort belonging to L* has just begun, thus it is too early to predict whether it can produce a large number of theorems, some of which are of general practical importance and others, perhaps, which provide new fundamental insights. But at least the outlook is brighter, since we begin with PFOs having a strong algebraic structure and have the possibility of discovering others that may strengthen it further. The laws relating the principal PFOs of L* are themselves already useful and very general identities and from them a small number of general theorems have been obtained [Backus 78], [Backus 81a],[Kieburtz & Shultis 81], [Williams 80].

The kind of mathematical system represented by L* is somewhat new in having a relatively large number of operations (PFOs) on a

single domain (of L*-programs) and an even larger number of laws relating these operations. I believe this may turn out to be an exciting new area for study, a relatively unexplored one that invites adjustment, exploration and classification by mathematicians. I believe it is such studies that give the best hope for a mathematics of "programs", one that can guide us toward the rapid development of a program by using its accumulated knowledge and one that will provide the general tools for concisely proving a program correct and for optimizing it.

Acknowledgments

I am grateful to Steven S. Muchnick for his careful and thoughtful review of an earlier draft of this paper. I owe thanks also to John H. Williams for many helpful discussions on various topics covered in the paper.

References

Arvind, Gostelow, K. P., and Plouffe, W. (1978) An asynchronous programming language and computing machine. Univ. of Calif. (Irvine) Report, Dept. of Information & Computer Science.

Backus, J. (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. CACM 21, 8.

Backus, J. (1981a) The algebra of functional programs: function level reasoning, linear equations, and extended definitions. Proc. International Colloquium on Formalization of Programming Concepts, Peniscola, Spain (April), Lecture Notes in Computer Science, No. 107, Springer-Verlag, Heidelberg.

Backus, J. (1981b) Function level programs as mathematical objects. Proc. Conf. on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire (October)

Burstall, R. and Goguen, J. A. (1980) The semantics of CLEAR, a specification language. Lecture Notes in Computer Science, No. 86, Springer Verlag, Heidelberg.

Church, A. (1941) The calculi of lambda-conversion. Princeton Univ. Press, Princeton.

Darlington, J. and Reeve, M. (1981) ALICE: A multi-processor reduction machine for parallel evaluation of applicative languages. Proc. Conf. on Functional Languages and Computer Architecture, Portsmouth, New Hampshire (October).

Dennis, J. B., Leung, C. K. C., and Misunas, D. P. (1979) A highly parallel processor using a data flow machine language. CSG Memo 134-1, Laboratory for Computer Science, MIT, Cambridge Mass.

Guttag, J. V. and Horning, J. J. (1978) The algebraic specification of abstract data types. Acta Informatica 10.

Holloway, J., Steele, G. L., Sussman, G. J., and Bell, A. (1980) The SCHEME-79 chip. AI Memo No. 559, Artificial Intelligence Laboratory, MIT, Cambridge, Mass.

Iverson, K. E. (1962) A programming language. Wiley, NY.

Keller, R. M., Lindstrom, G. and Patil, S. (1978) An architecture for a loosely-coupled parallel processor. TR UUCS-78-105, Dept. of Computer Science, Univ. of Utah, Salt Lake City.

Kieburtz, R. and Shultis, J. (1981) Transformation of FP program schemes. Proc. Conf. on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire (October).

Landin, P. J. (1966) The next 700 programming languages. CACM 9, 3.

Mago, G. A. (1979) A network of microprocessors to execute reduction languages. International Jour. of Computer & Information Sciences 8,5 and 8,6.

Milne, R. E. and Strachey, C. (1976) A theory of programming language semantics. Chapman & Hall, London and Wiley, New York.

Reynolds, J. C. (1970) GEDANKEN -- A simple typeless language based on the principle of completeness and the reference concept. CACM 13, 5.

Scott, D. S. and Strachey, C. (1971) Toward a mathematical semantics for computer languages. Tech. Monograph PRG-6, Univ. of Oxford.

Stoy, J. E. (1977) Denotational semantics: The Scott-Strachey approach to programming language theory. MIT Press, Cambridge, Mass.

Tennent, R. D. (1976) The denotational semantics of programming languages. CACM 19, 8.

Thatcher, J. W., Wagner, E. G. and Wright, J. B. (1978) Data type specification: parameterization and the power of specification techniques. Proc. Tenth Annual ACM Symposium on Theory of Computing.

New York (May).

Williams, J. H. (1980) On the development of the algebra of functional programs. Tech Report RJ2983, IBM Research Laboratory, San Jose.

Williams, J. H. (1981) Notes on the FP style of functional programming. Lecture notes for the course "Functional Programming and its Applications", Univ. of Newcastle upon Tyne (July).

Zilles, S. N. (1979) An introduction to data algebras. Lecture Notes in Computer Science, No. 86, Springer Verlag, Heidelberg.