CORRECTION COPY

DRAFT

p 17
p 20
p 23

On Extending the Concept of "Program" and Solving Linear Functional Equations

John Backus
IBM Research Laboratory, San Jose, California
August 10, 1979

1. Introduction

This paper identifies a fundamental difficulty with the practice of programming whose source lies at the deepest levels of our thought processes and in the foundations of the languages we use to express programs. Why does planning a program at a high level involve so many inappropriate details? Why are programming languages, and the concept of "program" itself, so complex, so lacking in mathematical elegance and power? Why is it so difficult to analyze programs to determine their effect? Why are there almost no general theorems that relate the structure of a program and its behavior, no theorems to guide us in choosing the proper structure of a program and to help us prove its correctness? Why, instead, do we find only correctness proofs that apply to individual programs?

We present evidence to show that the answer to many of these questions is closely connected with our concept of what a "program" is. That concept is largely determined by the choice of "program domain" that "programs" map into itself. Thus conventional, von Neumann programs are those that map stores into stores, a "von Neumann store" being a set of name-object "cells"; thus the concept "von Neumann program" is determined by choosing the program domain to be the set of "von Neumann stores".

Once the concept of "program" is essentially determined by the choice of program domain, the remaining basic question for a programming language is how programs are to be constructed. The

modern answer to this question, suggested by the work of Scott and Strachey [ ], is that programs are built from primitive entities by "operations". For example, identifiers, numbers, and other entities are common primitives; the expression "a+b" is built from the expressions "a" and "b" by the operation "+", and the program "while b do p" is built from the expression "b" and the program "p" by the operation iterate.

Thus the most important components of a programming language are, first, the "programs" it represents (and these are determined by the choice of "program domain"), and second, the entity- and program-forming "operations" used to build its programs. These two components are relatively independent; once a program domain is chosen and hence the class of programs is determined, there is still a wide (and relatively unexplored) choice of operations that can be used to build programs from the primitives.

This paper has two main theses. The first depends on the existence of program-forming operations that are both powerful program-builders and at the same time obey interrelating algebraic laws. A set of such pair-wise "strongly algebraic" program-forming operations was exhibited in an earlier paper [ ]; I refer primarily to the basic set of operations, called "functional forms": composition, condition, and construction. The first thesis is that, if strong theorems are to exist about large classes of programs, then programs must be built by a strongly algebraic set of operations. The fact that strong theorems do then exist is illustrated by the general solution of "linear" functional equations and its dependence on the algebraic laws satisfied by the program-forming operations involved (the treatment of linear equations comprises the last section of this paper).

The second thesis is that the choice of "program domain" and hence of the class of "programs" can limit the choice of program-forming operations so that it becomes difficult, perhaps impossible, to find a sufficiently algebraic set of operations (for building the chosen class of programs) to make possible the proof of useful

theorems about large classes of programs.

The evidence for these two theses that we present in the paper comprises a number of results of interest in themselves:

(1) Any von Neumann language L can be extended to a non-von Neumann, functional language L* so that the programs of L form a tiny subset of those of L*.

(2) The extension of the notion of "program" from that of L to that of L* is effected simply by extending the domain of L (stores) to form the program domain of L* (by forming the set of all stores, storeable objects, and names of L and then forming the closure of this set under cartesian products).

(3) The non-von Neumann programs of L* have a more direct interpretation than the programs of L.

(4) The entities of L (constants, variables, expressions, assignments, and programs) all have images in L* that are programs of L*.

(5) The entities and, at the highest level, programs of L traditionally require a considerable variety of types and of entity-forming operations.

(6) The larger, but simpler class of programs in L* can be built by four program-forming operations plus recursive equations.

(7) This set of program-forming operations in L* form a strongly algebraic set; those of L do not.

(8) The operation "construction" plays a fundamental role in L* but cannot be used in L because it always constructs a non-von Neumann program.

To illustrate the use of the program-forming operations in L* we show how they can be used to construct the L*-images of the entities of L.

Included in our comparison of L and L* is a discussion of the reasons why the L*-concept of "program" is disturbing and difficult to accept. They relate to our profound conditioning by our use of natural language, by our use of a name to stand in the place of the thing it refers to. This conditioning is reflected in our mathematical notation and thought; it is the basis for our choice of the domain that defines "von Neumann" languages, for the role of the "von Neumann store" is to preserve our traditional naming practices.

Despite the mental anguish many computer scientists may suffer in moving from the thought-world of von Neumann languages like L to the thought-world of non-von Neumann languages like L*, I believe that the comparison below will show that it is the tradition-dictated choice of the domain of L -- the set of von Neumann stores -- that has led us into a complex, un-mathematical world, that of conventional languages. I believe the comparison shows the difficulty, perhaps impossibility, of creating reasonable order in the constrained L-world, and the strong possibility that in the richer L*-world "programs" can become elegant mathematical objects subject to some very practical and far-reaching theorems.

Following the sections on L and L* there is a section that discusses the power and the difficulties of the program-forming operations for lambda-calculus-based languages. Then comes a brief review of functional programming as a prelude to the final section. The latter develops the notion of "forms", abstract program-forming operations in one or more variables, in an FP setting.

This final section defines "linear" forms Hf in the variable f to be those for which there exists a corresponding form H_1 such that, for all functions (programs) a, b, and c, the following holds:

$$H(a \rightarrow b; c) = H_1 a \rightarrow Hb; Hc \ .$$

We then consider "linear" equations of the form

(1) $\quad f = p \to q;\ Hf\ ,$

where p and q are any programs, Hf is linear and $\bar{I}$-preserving (i.e., $H\bar{I} = \bar{I}$, where $\bar{I}$ is the everywhere-undefined program), and the program $p \to q;r$ applied to an object x, written $(p \to q;r):x$, yields $q:x$ if $p:x$ is "true", or $r:x$ if $p:x$ is "false", and otherwise yields $\perp$. We prove that the least solution f of any such equation can be expressed as an "infinite conditional":

(2) $\quad f = p \to q;\ H_1p \to Hq;\ldots;\ H_1{}^np \to H^nq;\ldots$

where H_1 is the "predicate transformer" associated with the linear H. This solution not only gives a case-by-case description of the mapping represented by f (that is often of great value in proving the "partial correctness" of f), but also provides necessary and sufficient conditions for the termination of f.

The recursive definition (1) of f and its solution (2) apply to any language that has (a) the program-forming operation condition (that maps programs p, q and r into the program $p \to q;r$), and (b) continuous, linear and $\bar{I}$-preserving program-forming operations H that map a program f into Hf. Given any such linear and $\bar{I}$-preserving forms Hf and Gf, it is shown that their composition $HGf = H(Gf)$ is also linear and $\bar{I}$-preserving, and that the predicate transformer $(HG)_1$ of HG is the composition of their individual predicate transformers H_1G_1.

This section also shows that the basic program-forming operations of L* -- composition, condition, and construction -- are linear in all of their variables by virtue of the algebraic laws that interrelate them. Thus a great many forms built by composing the basic program-forming operations are linear, consequently the above "linear expansion theorem" applies to a great many recursive definitions in L* as well as in FP.

2. Basic concepts and goals of programming languages

The main purpose of a general purpose programming language is to allow its users to express any algorithm of a large class clearly and concisely. The concepts it employs should also comprise a powerful set of intellectual tools to help users think about algorithms at the highest possible level. Programs should be mathematical objects in the sense that it is possible to state and prove useful theorems about large classes of programs, rather than just theorems about individual programs.

There are two common meanings of "program", one refers to a mapping or function, the other to a representation of such a mapping, some finite object that "represents" the mapping. Thus a program in the first sense is an infinite object, in the second, a finite one. There may be many program-representations that represent the same program-mapping. In the following I shall use "program" to refer primarily to mappings but also to representations. A subsequent section shows how this ambiguity can be tolerated without much confusion.

If programs are to be freely composed they must all have the same domain and range D. When we say that "p is a program on D" we mean that if x is any element of D, then the result of applying p to x, written "p:x", is an element of D. Following Scott [ ], we include partially defined programs in the term "programs on D" by assuming that any domain D has an element "undefined" or "$\perp$", such that $p:x = \perp$ if p is "not defined" for x; we also assume some suitable partial ordering on D and that each program is a continuous mapping of D into D.

Furthermore, if p is a program on D and D is a subset of D', then p on D corresponds to a unique p' on D', an extension of p, such that $p:x = p':x$ for all x in D and $p':x = \perp$ for all x in D'-D. In this situation we shall speak of the program p on D as if it were also a program on D', where in the latter case we really mean its unique extension p'.

We shall think of a programming language as having two main components, the first being the class of all programs -- continuous functions -- on some domain D; it will presumably contain representations for all programs on D. The other principal component of a programming language is its set of operations used to build programs.

If the concept of "program" is to become an elegant mathematical one -- rather than the complex mess it is now -- then I believe that the following points must be accepted:

(a) To each domain D there corresponds a concept of "program", namely, the set of programs on D.

(b) If the concept of "programs on D" is to be simple and powerful, then the constructions used to build D must be simple and powerful. D should be closed under these constructions.

(c) Programs on D are constructed by a set of program-forming operations, each of whose arguments is a program on D or an element of D.

(d) Finally, and most importantly, the set of basic program-forming operations should satisfy algebraic laws that interrelate their properties. The extent and power of these laws will determine the degree to which there exist general theorems about "programs".

Points (a) and (b) are already widely accepted, at least in theory, as a result of the work of Scott and Strachey [ ]; but (c) and (d) are less widely accepted. The consequences of this general view for language design, including the last two points, have not been generally understood and so far it has had almost no impact on the practice of language design. In particular, few computer scientists have recognized the promise and practical importance of having a large body of general theorems about the behavior of large classes of programs, where each class is determined by the structure of the program-forming operations used to build programs in the class. Such general theorems

could be used by programmers both to guide their choice of structure for a program and to prove its correctness. Our failure to recognize the benefits of such theorems results from the fact that few such theorems exist for the programs of most languages, since their program-forming operations obey almost no useful algebraic laws.

The viewpoint we are advancing emphasizes three levels in constructing a programming language, level 0 being a domain D, level 1 being the programs on D -- the continuous mappings of D into itself, and level 2 being a set of continuous program-forming operations that map tuples (composed of programs on D and elements of D) into programs on D. We shall see that there are important program-forming operations for building programs on a well-constructed D that are inadmissable for building programs on a poorly constructed D'. Thus it becomes intuitively clear that our hope of having program-forming operations at level 2 that obey simple and powerful algebraic laws depends heavily on the proper construction of D at level 0.

3. Programs as mappings vs. programs as expressions

So far we have tended to identify "programs" with "mappings" [abstract programs]; we shall also want to think of programs as finite expressions [concrete programs]. Our purpose here is to sketch a correspondence between abstract and concrete programs so that when we speak of building "programs", either sense of the word can be used and yet give closely corresponding interpretations.

We assume there is a set R of finite expressions [R may be a a subset of D] such that each r in R represents some abstract program p in P = [D → D]. (There may be many expressions r that represent a given p.) We let ρ map each r into the p it represents: $\rho \in [R \to P]$. We write $r \simeq r'$ when r and r' represent the same program p, i.e., when $\rho r = \rho r' = p$.

If we identify the notion of an abstract program p with the equivalence class of expressions in R that represent p, then we can

speak of this "program" usually without caring whether p, or the equivalence class, or some expression in it is meant. It remains now to establish a similar correspondence between abstract-program-forming operations and concrete-program-forming operations; we shall then be able to use both "program" and "program-forming operation" in either sense without confusion in most contexts.

Program-forming operations take one or more arguments that are either programs on, or elements of, D. Thus if π is an abstract-program-forming operation, its arguments are either abstract programs or objects: $\pi \in [(P \cup D)^n \to P]$. Similarly, a concrete-program-forming operation $\tilde{\pi}$ maps concrete programs or objects into concrete programs, thus $\tilde{\pi}$ belongs to $[(R \cup D)^n \to R]$. But $\tilde{\pi}$ must also be faithful with respect to the way expressions r represent programs: if $r \simeq r'$, then we must have $\tilde{\pi}(...r...) \simeq \tilde{\pi}(...r'...)$; that is, the program represented by $\tilde{\pi}(...r...)$ does not change when r is replaced by another representation r' of the same program.

For each abstract-program-forming operation π there is a corresponding concrete-program-forming operation $\tilde{\pi}$, so that, if π builds p from $p_1,...,p_n$, $p = \pi(p_1,...,p_n)$, then $\tilde{\pi}$ builds a representation r of p from any representatives $r_1,...,r_n$ of $p_1,...,p_n$; that is, there is an expression $r = \tilde{\pi}(r_1,...,r_n)$ in R that represents p in P, where $\rho r = p$ and $\rho r_i = p_i$. With this understanding we can speak of building programs with program-forming operations without specifying whether we are referring to abstract or to concrete programs and/or program-forming operations.

4. Algebraic sets of operations

We asserted earlier that it is important to have a set of program-forming operations that satisfy some set of algebraic laws. Here we discuss briefly the basis for that assertion.

Consider the set A of expressions built from numbers with the operations "+" and "×"; e.g., -7, 3 + 4.2, (1+(5×2))×3. Each of the

operations of A obeys certain algebraic laws. The associative and commutative laws concern single operations and apply to both "+" and "×". The distributive law relates the behavior of the two; it holds for all numbers a, b, and c and states that a sum equals a product:

$$(a \times b) + (a \times c) = a \times (b+c) .$$

Since the operations, and hence the expressions of A, all yield numbers as results, the distributive law can be extended so that it holds for all expressions a, b and c. Such laws are structural laws in the sense that they are not concerned with the values of the individual numbers a, b and c, nor, if a, b and c are expressions, are they concerned with their internal structure or values; rather, such a law is concerned only with the structure of the operations used to build an expression. Algebra is concerned with the consequences of such laws.

We shall say, loosely, that a set of operations is "algebraic" to the extent that there exist algebraic or structural laws relating pairs of operations in the set (plus perhaps other laws concerning individual operations). Thus the set of operations used to build the expressions of A above is a "strongly" algebraic set, and consequently there are many powerful and useful structural theorems about these expressions and about equations involving them. The general solution for all quadratic equations is an example of such a theorem.

Now let us consider the set B of all expressions built from numbers with the operations "+" and "$\sqrt{\ }$". The set of operations of B is not algebraic; if one has few laws relating a set of operations, then it is difficult to prove general theorems about them, consequently there are few general theorems about the expressions of B. Thus for example, there is no form E(a,b) in B so that

$$\sqrt{a} + \sqrt{b} = E(a,b)$$

holds for all numbers a and b, unless E(a,b) differs only trivially

from the left side of the equation. For the most part we can only prove theorems about particular expressions, e.g.,

$$\sqrt{4} + \sqrt{9} = 5 .$$

Finally, we note for future reference that B can be extended by extending (a) the domain of its operations and (b) its set of operations. Thus we can extend "numbers" to "complex numbers" and we can add the operation "×" to the operations of B. The result is a set B' of expressions that has an algebraic set of operations, and general theorems now exist. For example, there is now a general expression for E(a,b) above, where E(a,b) is in B' and differs non-trivially from $\sqrt{a} + \sqrt{b}$; thus

$$\sqrt{a} + \sqrt{b} = \sqrt{(a+b+(2\times\sqrt{(a\times b)}))}$$

holds for all numbers a and b.

5. The structure of a programming language vs. the structure of a program

In any language that builds programs by means of a distinguished set of program-forming operations, a program is "structured" by the structure of the operations used to build it. Thus "while p do A" and "while q do B" have the same top-level structure, although the structure of programs A and B may differ. The term "structured programming language" has been loosely applied to languages with (a) a distinguished set of program-forming operations and (b) a set of primitive program elements that make it possible to describe simply the program built from arbitrary elements by a given operation. Thus a language with GOTOs is not "structured" because it is difficult to describe the programs "if p then S else T" and "S;T" if programs S and T have certain GOTOs in them.

The above sense of "structured programming language" is both trivial and misleading. If programs are to be built by reasonable operations, then of course those operations must be easily definable

for arbitrary parameters. If programs are built by such well defined operations, then every program is "structured" by those operations. And, most importantly, such a language does not necessarily have any structure at all, only its programs have structure.

The set of program-forming operations of a language is one of its primary attributes, and that set can be said to have "structure" to the extent that it is an algebraic set, since interrelating algebraic laws provide a useful notion of structure for a set of operations. Consequently I propose that a programming language should be called "structured" to the degree that its set of program-forming operations is algebraic.

6. A maximal extension of the von Neumann concept of "program"

6.1 Introduction

Earlier we suggested that the notion of "program" should correspond, following Scott and Strachey [ ], to the continuous mappings of some properly constructed program domain D into itself and that the structure of D would have an important influence on the potential "structure" -- the set of algebraic laws -- of the program-forming operations available for building programs. In this section we show that the kind of program domain that defines the concept of "von Neumann program" is poorly chosen in that it leads to complex program-forming operations that have almost no algebraic structure. We show that there is a natural extension D* of the program domain D (the set of "von Neumann stores" that defines "von Neumann program") such that D* defines a much richer class of programs. We shall argue that D* defines a concept of "program" that (a) maximally extends the conventional concept, (b) has a simpler, more direct interpretation (as a mapping of input into output, instead of stores into stores), and (c) admits simpler, more powerful sets of program-forming operations that possess useful algebraic structure.

We shall be explaining an extension of the concept of "program" that may be as puzzling and disturbing to many computer scientists

as the extension of "number" to "complex number" was to 17th-century mathematicians. It violates many of our mental habits that result from the profound conditioning of our thought processes by our use of natural language. Extending the concept of "program" demotes to a minor role the ability to refer to something by mentioning its name, that is, demotes it from the central role it plays in conventional languages. Furthermore, it requires us to think more about combining programs, and less about combining objects (as we are accustomed to do).

Since we shall be dealing with broad notions such as "conventional programming language", we shall not be concerned with many significant details but nevertheless hope to accurately reflect major issues. Although the basic ideas underlying our extension of von Neumann languages come from functional programming, my purpose is not to justify the latter but to present some constructive insights that can be used to radically improve conventional programming languages. I hope these insights can one day bring new life to the study of programming languages by extending the subject beyond its present narrow boundaries and by stimulating the invention of new kinds of languages in which "programs" become interesting mathematical objects.

the following

The reader may suspect that ~~my~~ description of the von Neumann language L is setting up a "straw man" to be knocked down by the extended language L*; but the fact is that I am trying to show that the problems of L are at the most fundamental level and hence have no need to introduce extraneous difficulties. If any violence has been done to the concept of "von Neumann language" in L, it is to make it simpler than real languages.

6.2 Basics of the von Neumann language L

Consider some typical von Neumann language L. Its programs are defined by the program domain S, the set of "von Neumann stores for L", where ~~such a~~ every store is a set of cells, each cell having a "name" and a "contents". We shall call any such name or contents

an object of L and let O be the set of all objects of L. We shall assume that L has a rich set of objects built from elementary objects such as "true", "false", numbers, symbols, etc. by constructions such as sequences, arrays, stacks, records, files, etc.

The programs of L are the continuous mappings of S into itself, of stores into stores. Let us call the set of all stores and objects of L, S∪O, the total domain of L. Observe that the program domain S of L is a ~~tiny~~ subset of its total domain.

6.3 Definition of L*; the extended concept of "program"

We take as the set O* of objects of L* the closure under cartesian products of the total domain of L, S∪O. Thus if $x_1,\ldots,x_n$ are stores or objects of L, or objects of L*, then each x_i and $\langle x_1,\ldots,x_n\rangle$ are objects of L*. The set O* of objects of L* will be both its program domain and its total domain. Thus the programs of L* are the continuous mappings of O* into itself; each such program maps L*-objects into L*-objects, and these are either stores of L, L-objects, or sequences built from these.

The programs of L* represent our extended notion of "program". Let us look at some consequences of this extension in the following subsections.

6.4 L* is an extension of L

Let us roughly define "L_2 is an extension of L_1" to mean (a) the total domain of L_2 includes that of L_1, and (b) the program domain of L_2 includes that of L_1. From (b) it follows that every program of L_1 is a program of L_2 in the sense discussed earlier: each program of L_1 has a unique extension in L_2 that is everywhere "undefined" outside of the program domain of L_1.

In the above sense L* is an extension of L, every program of L is a program of L*, and L* is a maximal extension of L that has O* as its total domain, since that is also the program domain of L*. We have not lost any of the very restrictive von Neumann programs of

L in the extension, if and when they are useful, and we have added a vast new territory of "non-von Neumann" programs that do not ~~necessarily~~ map stores into stores.

6.5 Interpretation of programs in L* is simpler than that of programs in L

It would be nice to think of a "program" as mapping its input into its output. But, for example, if we want to describe a von Neumann program for factorial, we are compelled to say that it maps a store s (in which some cell "a" contains n) into a store s' (in which some cell "b" contains n!). We do not really care about the names of the cells that hold n and n!; furthermore, this description of the program is incomplete. It does not say what the program does to all the other cells (and it usually will do something to a few of them). In the strict sense of "program" we have been using, there does not exist a von Neumann program that maps n into n!; no von Neumann program can do that directly (since it maps stores into stores).

In contrast, the non-von Neumann programs of our extension L* can map objects into objects as well as stores into stores, consequently there is a real factorial program that maps n into n! without involving any store. It does not require cells for n and n!; we do not have to concern ourselves with what it does to all the cells of a store, either directly, by convention, or by ignoring the question.

Thus our extended notion of "program" admits a simpler, more direct kind of program with a simpler, more direct interpretation. At the same time, it includes the familiar von Neumann programs.

6.6 The programs of L* admit algebraic sets of simpler program-forming operations that cannot be used in L

A number of subsections are required to make this point; these include the following topics: (a) a survey of program formation in L, (b) a description of new program-forming operations in L*, (c) a demonstration that program formation in L, with its many diverse operations, can be modelled in L* with its few new and orderly operations, and

(d) a demonstration that the program-forming operations of L* form an algebraic set, whereas those of L do not.

6.6.1 Program-forming operations in L

Programs in the von Neumann language L are built from many different kinds of entities at several levels; each level of entity has its own entity-forming operations, with programs and program-forming operations at the highest level. A rough survey is sufficient to exhibit the confused variety of types that makes the program-forming and entity-forming operations of L so algebraically unstructured.

6.6.1.1 Constant and variable formation. Variables are built from identifiers and subscript expressions; they map stores into objects, hence they are of type "expression", E = [S → O]. They are built by a variable-forming operation of type [I×E → E], where I is the set of identifiers of L. For example, this operation builds the variable v[i+1] from the identifier v and the expression i+1. We shall regard constants also as constant-valued mappings of type E.

6.6.1.2 Expression formation. Expressions are built from constants, variables, and expressions by object-forming operations such as "+" and "cosine". Expressions are of type E. From each object-forming operation such as "+", where "+" is of type [O×O → O], we form a corresponding expression-forming operation "$+_e$" of type [E×E → E] as follows. (We write e:s for the object obtained by applying expression e to store s; thus if b is a variable then b:s = contents of cell b, and 3:s = 3 .) Then "$+_e$" maps the pair of expressions <a,b> into the expression "a$+_e$b", where the mapping in [S → O] represented by "a$+_e$b" is given by the following rule for any store s:

$$(a +_e b):s = (a:s) + (b:s) .$$

Thus $(a +_e 3):s = a:s + 3:s = a:s + 3$, and $(sqrt_e\ 4):s = sqrt\ (4:s) = sqrt\ 4 = 2$, and $(cos_e\ b):s = cos\ (b:s)$, etc. In this way operations of type O → O are converted to type E → E and operations of type O×O → O are converted to operations of type E×E → E, etc.

6.6.1.3 Assignment formation. Assignments are built from identifiers and expressions. They are of type "program" P = [S → S]. They are built by the program-forming operation ":=" of type [I×E → P]. Thus ":=" builds the assignment "v:=e" from the identifier "v" and the expression "e". In this context "v" is not a variable, i.e. expression, since no mapping associated with "v" is used in building the assignment. Thus "v" serves only as an identifier on the left, although it may occur as a variable in e. The assignment "v:=e" denotes the mapping that satisfies

v:((v:=e):s) = e:s and v':((v:=e):s) = v':s for all v'≠v

for any store s, where (v:=e):s is the store obtained by applying v:=e to s. Thus assignments have the peculiar property of converting an identifier "v" into an expression, a mapping.

6.6.1.4 Program formation. Programs are built from expressions, assignments, and programs by several program-forming operations. Of course, they are of type "program". The program-forming operation composition maps a pair of programs <p,q> into the program "p;q" whose mapping is (p;q):s = q:(p:s). Composition is of type [P×P → P].

The operation conditional maps a boolean expression b and programs p and q into the program "if b then p else q". Its mapping, (if b then p else q):s , is p:s if b:s = "true", and q:s if b:s = "false". Conditional is of type [E×P×P → P].

Similarly, the operation iterate maps <b,p> into "while b do p" which maps s into s if b:s is "false" and into ~~((while b do p);p):s~~ (p;(while b do p)):s if b:s is "true". Iterate is of type [E×P → P].

6.6.1.5 Formation of procedure definitions. One remaining way of building expressions and programs in L involves the combination of a function or a procedure statement and a procedure declaration which they invoke. The ~~latter~~ former, roughly speaking, involves (a) a definition (e.g., identifying a function name with its defining expression) and (b) the operation of lambda abstraction that, for example, converts a form H (a program, built as above, involving variables x and y)

into the function $\lambda(x,y).H$ or $H(x,y)$.

Although in other settings lambda abstraction permits the definition of many types of functions including program- or function-forming operations, procedure declarations cannot define program-forming programs in L. For no program in L can produce a program (unless programs are represented by stores, since programs produce only stores; that would be a very poor choice of representation).

On the other hand, if programs in L are represented by objects (a reasonable possibility), then defined functions could produce program representations. However, in L functions can only appear as expressions, and expressions can never appear where programs can. This makes the use of defined program-forming functions unwieldy at best; the result is that they are of no practical value in von Neumann languages and that all of the above complex entity- and program-forming operations must be built into such languages.

6.6.1.6 Remarks about program-forming operations in L. The main points to notice about the construction of L-programs are:
(a) the large variety of types and entity-forming operations in L arise largely from the restrictive demand that only stores shall comprise the domain of expressions and the domain and range of programs;
(b) expressions really describe the actual computations of L; and
(c) expressions are built with object-forming operations.

Thus the principal parts of L-programs (expressions) are not built by program-forming, but by object-forming operations. Some of these object-forming operations obey algebraic laws that permit the proof of general theorems about the equivalence of expressions, where equivalence, $e \simeq e'$, means $e:s = e':s$ for all stores s. These theorems about expressions give rise to corresponding theorems about programs: if e is part of the program p(e) and $e \simeq e'$, then (with some restrictions) $p(e) \simeq p(e')$, that is, $p(e):s = p(e'):s$ for all stores s. These theorems are weak and not very helpful however; strong theorems about program equivalence require that the program-forming operations of L obey

algebraic laws. However, the existing program-forming operations for L listed above, and others such as Dijkstra's (composition, if-fi, do-od) [ ] that are based on the von Neumann concept of "program", satisfy few such laws (as we shall see below). Moreover, the jumbled type structure of languages like L (and of course our L is very simple compared to real languages) makes it unlikely that new program-forming operations will be found for them that have a strong algebraic structure.

6.6.2 Program-forming operations in L*

We have seen earlier that each program p of L has a unique extension p* in L* such that p:s = p*:s for all stores s. In a similar manner each expression e in L has a unique extension into a program q* in L* such that e:s = q*:s for all stores s.

In this way the entities of L all have images (i.e., extensions) that are programs in L*; but these images are a tiny subset of all the programs of L*. The entity-forming operations of L also have images in L*, but most of them are useful only for building those L*-programs in the tiny subset of L-images. Therefore, to build the majority of programs in L* we shall need new program-forming operations, some of which are extensions of those of L (e.g., composition) and others that are new. It will turn out that four operations, plus the solutions of recursive equations, are adequate to build all the programs of L*, including the images of L-programs.

(Note that we have a free choice of program-forming operations we use to build programs in both L and L*; we have discussed only those traditionally used in L, and we shall rely almost entirely on just four in L*.)

We shall describe four basic program-forming operations for use in L* and then (a) sketch how they can be used to construct the L*-image of any L-entity, (b) show that three of these four operations form a strongly algebraic set, and (c) show that one of these three operations can never be used in L to build its programs.

6.6.3 The basic program-forming operations of L*

The basic program-forming operations we shall use in L* are the following. We shall use "object" to refer to L*-objects.

Constant maps any object x into the program $\bar{x}$, where $\bar{x}:y = x$ for any object $y \neq \perp$. Constant is of type $[O^* \to P^*]$.

Composition maps programs p and q into the program $p \circ q$, where $(p \circ q):x = p:(q:x)$ for all objects x. Composition is of type $[(P^*)^2 \to P^*]$.

Condition maps the programs p, q, and r into the program $p \to q;r$, where $(p \to q;r):x$ is $q:x$ if $p:x$="true", and is $r:x$ if $p:x$="false", and is " " otherwise, for all objects x. Condition is of type $[(P^*)^3 \to P^*]$.

Construction maps programs $p_1,\ldots,p_n$ into the program $[p_1,\ldots,p_n]$, where $[p_1,\ldots,p_n]:x = \langle p_1:x,\ldots,p_n:x \rangle$ for all objects x. Construction is of types $[(P^*)^n \to P^*]$ for $n=0,1,2,\ldots$

Recall that the set O^* of L*-objects is closed under cartesian products: $(O^*)^n \subset O^*$, hence the result of a program built by construction is an L*-object. Note that construction plays a fundamental role: it is the principal means for building programs that produce elements of $(O^*)^n$ from elements of O^*; thus it builds argument-forming programs that produce the arguments for n-ary programs that map $(O^*)^n$ into O^*. For example, if p is an L*-program defined on pairs $\langle x,y \rangle$, then programs involving p will often contain an expression $p \circ [q,r]$ in which $[q,r]$ constructs the argument for p. Thus the L*-program for x + logx is $+ \circ [id,log]$, where $id:x = x$, and so $+ \circ [id,log]:x = +:\langle id:x, log:x \rangle = +:\langle x, logx \rangle = x + logx$.

6.6.4 L*-images of L-programs and L-entities

If x is some entity of L (constant, variable, expression, program), we shall denote its image program in L* by $\tilde{x}$. For the various entities x of L we shall show how to build its image $\tilde{x}$ in L* using the above four program-forming operations. It is hoped that our examples will

convince the reader that any program of L can be built in this way, although the L*-images of L-programs are seldom programs to be preferred in L*, since we shall seldom be interested in mapping stores into stores.

6.6.4.1 L*-images of L-constants. The image $\tilde{3}$ in L* of the constant 3 in L is the program $\bar{3}$, since if s is any store, then $\bar{3}$:s = 3 in L*, just as 3:s = 3 in L.

6.6.4.2 L*-images of L-variables. The image $\tilde{v}$ in L* of the L-variable v requires the use of a program-forming operation fetch in L* that can be defined in terms of certain primitive L*-programs and the four basic program-forming operations of L*. We assume its existence; it maps a program p into the program ↑p, where ↑p:s is the contents of the cell in store s whose name is p:s. Thus the image $\tilde{v}$ is ↑$\bar{v}$, since ↑$\bar{v}$:s in L* is the contents of the cell named $\bar{v}$:s = v, which is the same as v:s in L.

6.6.4.3 L*-images of L-expressions. If e and f are expressions in L and if $\tilde{e}$ and $\tilde{f}$ are their image-programs in L*, then the image of, say, e+f in L* is $+\circ[\tilde{e},\tilde{f}]$. Thus for any store s, $+\circ[\tilde{e},\tilde{f}]:s = +:\langle\tilde{e}:s, \tilde{f}:s\rangle = +:\langle e_s, f_s\rangle = e_s+f_s$, where $e_s = \tilde{e}:s$ (in L*) = e:s (in L) and similarly for f_s. Thus $+\circ[\tilde{e},\tilde{f}]:s$ in L* gives the same result as (e+f):s in L, for any store s.

6.6.4.4 L*-images of L-assignments. Here we shall need another program-forming operation, store, analogous to fetch; we shall assume it is available, as we did for fetch. Store maps a program p into the program ↓p; ↓p maps <x,s>, where s is a store, into a store s' that has a unique cell whose name is p:<x,s> and whose contents is x; s' has all the other cells of s.

The image of the assignment v:=e (in L) is then $\downarrow\bar{v}\circ[\tilde{e}, id]$. For if s is any store, then $\downarrow\bar{v}\circ[\tilde{e}, id]:s = \downarrow\bar{v}:\langle\tilde{e}:s, id:s\rangle = \downarrow\bar{v}:\langle e_s, s\rangle = s'$ in L*, where s' is s with the new cell named $v = \bar{v}:\langle e_s, s\rangle$ having contents $e_s = e:s$. Thus s' = (v:=e):s in L.

6.6.4.5 L*-images of L-programs. If p and q are L-programs, then the L*-image of "p;q" is $\tilde{q} \circ \tilde{p}$. If b is a boolean L-expression and p and q are L-programs, then the L*-image of "if b then p else q" is $\tilde{b} \rightarrow \tilde{p}; \tilde{q}$. The reader may easily check that applying either L*-image to a store s gives the same store as applying the corresponding L-program to s.

It would be a simple matter to define an iterative program-forming operation for L* analogous to "while-do" in L, but we will not do so. The principal program-forming operations in L*, composition, condition, and construction, form, as we shall see, a strongly algebraic set whose laws permit us to solve recursive equations that define a much larger class of programs than can be built with "while-do". Consequently we choose to use such equations to define these programs in L*.

If b is a boolean expression and p a program in L, then the L*-image of the L-program "while b do p" is defined by the equation

(1) $\quad q = \tilde{b} \rightarrow q \circ \tilde{p}; \mathrm{id}$.

It is fairly easy to see that if q is a program defined on the smallest possible domain that satisfies (1), then q:s = (while b do p):s for any store s.

In a later section we shall exhibit a general solution for the class of "linear" equations in L* and in other systems, of which the equations of the form (1) form a small subset.

6.6.4.6 L*-images of procedure definitions in L. As noted earlier, procedure definitions in L have two components, definition and lambda abstraction. Consider a simple Algol-like procedure declaration for a function:

```
Procedure f(x,y)  result (f);
     f := x + y + 1
```

The definition component identifies the symbol "f" with the lambda abstraction $\lambda(x,y).(x + y + 1)$. The lambda abstraction component is effected by listing "x" and "y" in the first line. (The need to name the result in the first line is a consequence of the requirement that the "body" of a declaration be a program and that the result of a program be a store, not an object.)

When L*-programs are represented in a suitable way by L*-objects, a lambda abstraction operation can easily be defined (I plan to describe a combinatory version of such an operation in a future paper on extended functional programming). But such an operation is not needed to model most procedure declarations of L in L*.

If we look closely at our above example in L, we find that the function f is defined by describing how to combine the objects denoted by "x", "y", and "1" to obtain the result. The lambda abstraction operation's purpose is to indicate that x is an "abstract" object that is to be the first argument of f, and similarly that y is "abstract" and ~~to be~~ the second argument. Thus the lambda abstraction operation "erases" x and y and replaces them by the argument sequence $\langle arg1, arg2 \rangle$, and thereby elevates a description of an "abstract" object, x+y+1, to the level of a function.

In L* the need for lambda abstraction can normally be dispensed with. Instead of describing how to combine <u>objects</u> and then converting that description into a function by lambda abstraction, the preferred technique in L* is to describe how to combine functions (i.e., programs) directly to form a desired function. In effect, after "abstracting" objects x to programs $\bar{x}$, we then remain at the "program" level in building programs, without descending to the "object" level.

Thus to define the L*-image of the L-program f above, we first build a totally "abstracted", variable-free program and then identify it with the symbol "f" by a <u>definition</u>:

<u>Def</u> $f = +\circ[1, +\circ[2,\bar{1}]]$

where "1" and "2" denote primitive programs that select the first and second elements, respectively, of a sequence. Thus f:<e,g> = +:<e,+:<g,1>> = e + g + 1. The above f is our L*-image of the procedure declaration in L.

The L*-image of the L-expression "f(e,g)" for any L-expressions e and g is then

$f \circ [\tilde{e},\tilde{g}]$

since $f \circ [\tilde{e},\tilde{g}]:s = f:<\tilde{e}:s, \tilde{g}:s> = f:<e_s,g_s> = e_s + g_s + 1$ in L*; and $f(e,g):s = f(e:s, g:s) = e_s + g_s + 1$ in L. Thus we have replaced definition and lambda abstraction in L by definition, a kind of combinatory abstraction, and functional argument construction.

6.6.5 Programming style in L* vs. programming style in L

The preceding example emphasizes the principal differences between the programming style of L and the preferred style in L*. In L, programming is largely a matter of combining objects by use of their names, followed by the use of lambda abstraction to convert a particularized program into a "general" one, a "procedure", by "denaming" its arguments. These general programs are then used by converting them back into particular ones by "calling" them, i.e., by naming the desired arguments.

In L*, programming is never a matter of combining objects (and seldom a matter of naming them, although there is a style of defining programs that use symbols in a way that resembles conventional use of names [a future paper will deal with that issue]). Objects are elevated to programs (x becomes $\bar{x}$); thereafter programming consists of combining only programs to form new programs (all of which are general and never name their arguments) by the use of program-forming operations. The arguments for programs are built at the "program" level, instead of at the object level, by the use of construction.

6.6.6 The program-forming operations of L* form an algebraic set; those of L do not

The three program-forming operations of L* (other than constant) obey, with one essential exception, virtually every reasonable "distributive" law between pairs of operations. For all programs p, f, g, h, $f_1,\ldots,f_n$ the following laws hold.

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

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

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

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

For the pair composition and condition there are two laws that express a composition involving a condition as a condition involving composition. There is one such law-scheme for the pair composition and construction, the missing law for that pair corresponds to the necessary inequality

$f\circ[g,h] \neq [f\circ g,\ f\circ h]$, ~~for example $+\circ[id,id] \neq [+\circ id,\ +\circ id] = [+,+]$~~

Finally there is a very useful law-scheme relating condition and construction. (Its validity depends on sequence-construction being strict: $\langle\ldots\perp\ldots\rangle = \perp$.)

As we shall see in the section on linear functional equations, these laws permit us to obtain a general solution for ~~all such equations~~ such equations built from these basic forms. They also form the basis for a great many general theorems that concern the properties of programs whose program-forming operations have a given structure, theorems that do not depend on the details of the subprograms that occupy that structure.

Now let us examine the program-forming operations of L. They obey one non-trivial interrelating law, the analogue of (1), (keeping in mind that f∘g is read right-to-left whereas p;q is read left-to-right):

(if b then p else q);r = if b then (p;r) else (q;r) .

But, even if we permit (p;b) as an "expression", where p is a program and b an expression, and (p;b):s = b:(p:s), even then the analogue of law (2) does not hold in L:

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

since if p adds 1 to some variable, then the left side will increase its value by 1, but the right side would increase it by two. (The semantics of L could be changed, and probably should be, to make the equality hold; but this would mean discarding the notion of a single, serially modified state; the store received by the two arms of the conditional on the right would have to be the original one, not the one produced from it by the application of p.) Thus in L as it stands only one analogue of the two laws in L* relating composition and condition holds.

Since the program-forming operation iterate (while-do) of L obeys no interrelating algebraic law with either composition or conditional, we are left with only one law for the three principal program-forming operations of L. If we consider other possible program-forming operations for L that might give it some algebraic structure, the picture is discouraging because of the variety of entities of L (with their variety of entity-forming operations) and because of the restrictive program domain of L.

In particular, since the operation of construction plays such a fundamental role in argument-formation and is so strongly related by algebraic laws to composition and condition, we might want to use it as a program-forming operation in L. But immediately we see that is impossible. The stores of L are not closed under cartesian products: a sequence of stores is not a store, not a set of cells. If p and q are L-programs and s is a store, then [p,q]:s = <p:s, q:s> thus [p,q] maps a store into the non-store <p:s, q:s>, hence [p,q] is not an L-program.

6.7 The "naming" and "forgetting" problems of programming in L*

The above sections conclude our comparison of the concepts of "program" associated with the von Neumann language L and with its extension L*. This comparison raises one obvious question: why has computer science clung so long to the restricted and complex von Neumann concept of "program", based on serial changes in a store, when a richer, simpler and better-structured alternative exists, the L* concept, that includes the former? The answer may be that we simply did not have the latter concept around to consider; but now that we do there are still many factors that will make it difficult for us to give up the old von Neumann approach. I would like to discuss two of these factors.

The first factor is that L-programs strongly reflect traditions of our natural spoken languages, involving the use of names to stand in place of the objects they denote, with the store as the implicit source of name-object associations. The demotion of stores in L* to the role of ordinary objects weakens our ability to use names in the traditional way. This is the "naming" problem in L*. There are a number of solutions that use functions or symbols, in place of "names" that require an implicit store [I plan to deal with them in a future paper].

The second problem that makes L*-programming seem strange and difficult at first is the "forgetting" problem. In an L-program some data may go unmentioned for much of the program, but when it is needed it is magically there, preserved in some little-used cell. In a non-von Neumann L*-program that transforms an input object into an output object, there is no implicit store to preserve incidental data. Even if the L*-program uses a store for data, the forgetting problem arises when evaluating an expression, because stores must be handled explicitly along with other objects.

For example, if $\uparrow\bar{a}$ and $\uparrow\bar{b}$ are programs for fetching the contents of cells a and b from some store s, and if we want to compute

$\uparrow\bar{a}:s + \uparrow\bar{b}:s = c$ (the equivalent of "a+b" in L), then the program $p = +\circ[\uparrow\bar{a},\uparrow\bar{b}]$ when applied to s gives c. But at a terrible price, for p transforms the store s into the number c, and s, with all its information has been lost, "forgotten". If c is to be our final result, then that is all right; but if c is subsequently to be combined with other values from s, we need to write a program that "remembers" s in addition to computing c. Thus we want the program [p,id], which when applied to s gives <c,s>, and we have s preserved along with the result c.

The forgetting problem is taken care of automatically and implicitly in L, whereas it must be dealt with explicitly in L*. I believe that this fact, plus our unfamiliarity with the new and basic operation of construction, which is needed to deal with the problem, accounts for much of the difficulty the L* concept of "program" will encounter.

7. Program-forming operations in lambda-calculus-based languages

[In preparation]

8. Review of functional programming and its algebra of programs

[Please see accompanying paper or following pages for this draft]

9. Linear forms and the solution of linear functional equations

[Please see accompanying paper or following pages for this draft]

Acknowledgments

References

1. Introduction

The purpose of this article is to provide a new basis for solving equations for programs in the algebra of functional programs described in [1]. This new basis supplies simpler and clearer machinery for solving equations, greatly enlarges the class of equations for which solutions were given in [1], and gives a single general solution for the entire class -- the class of "linear" equations.

This new basis makes clear why linearity of functional forms in a single variable is important for solving functional equations. As in [1] the general solution for the class of linear equations provides both a necessary and sufficient condition for the termination of the solution-function and a case-by-case description of its behavior.

The new basis for solving equations replaces all the definitions and theorems in sections 12.4, 12.5, 12.6 and 12.8 in [1]. The "recursion" and "iteration" theorems of [1] are special cases of the "linear expansion theorem" given here.

Section 2 of this article gives a brief review of functional programming and its algebra of programs. Section 3 discusses forms in a single variable and gives examples. Section 4 defines 'linear' forms and shows that many basic functional forms are linear in all of their variables. Section 5 shows that the composition of linear forms yields linear forms and that the composition of $\bar{I}$-preserving forms yields $\bar{I}$-preserving forms. Section 6 states the main result -- the linear expansion theorem -- that gives the solution of all equations of the form

$$f = p \rightarrow q;\ Hf$$

where H is any linear and $\bar{I}$-preserving form.

Section 7 discusses the range of applicablility of the linear expansion theorem and gives examples to show that it covers a

large class of equations. Sections 8 and 9 restate some standard definitions and results about limits, continuity, and fixed points and then give the proof of the linear expansion theorem. Section 10 discusses forms in two variables and, for some forms Hfg, gives necessary and sufficient conditions for Hff to be a linear form in the one variable f.

2. Functional programming (FP) and its algebra of programs

For a fuller description than this brief summary, see [1]. A functional program is a function from objects into objects. Objects are (a) $\perp$ "bottom" or "undefined", or (b) *atoms* or (c) *sequences* built from atoms or sequences (but not from $\perp$). An FP *program* is either (a) a *primitive* function or (b) a *functional form* that combines its "parameters" -- programs or objects -- to form a new program, or (c) it is *defined*, as in

Ø denotes the empty sequence; unlike [1], here Ø is a sequence but not an atom.

$$\underline{\text{Def}} \quad f = H(f, g_1, \ldots, g_n) \ ,$$

where H is a functional form involving the parameters g_1 through g_n and possibly the defined function symbol f. The effect of this definition is that when the application $f{:}x$ is to be evaluated, the defined function symbol f is replaced by its definition, $H(f, g_1, \ldots, g_n)$. We write $f{:}x$ to denote the object resulting from applying the function f to the object x.

Examples

Objects

atoms 1.4 *A* *ACD* 1 *X2*

sequences $\langle A \rangle$ $\langle A, \langle B, C, D \rangle, \langle B, C \rangle \rangle$ Ø (the empty sequence) ($\langle A, \perp \rangle = \perp$ is not a sequence)

Primitive functions (where x_i denotes an object)

$1{:}\langle x_1, \ldots, x_n \rangle = x_1$ $\quad 2{:}\langle x_1, x_2, \ldots, x_n \rangle = x_2$

$tl{:}\langle x_1 \rangle = \emptyset$ $\quad tl{:}\langle x_1, x_2, \ldots, x_n \rangle = \langle x_2, \ldots, x_n \rangle$

$+{:}\langle 1.4, 3.5 \rangle = 4.9$

$null{:}\emptyset = T$, $\quad null{:}x = F$ for $x \neq \emptyset$ and $x \neq \perp$

Functional forms

composition: $f \circ g$ where $f \circ g : x = f : (g : x)$

construction: $[f,g]$ where $[f,g] : x = \langle f:x,\ g:x \rangle$

condition: $(p \rightarrow f; g)$

$$\text{where } (p \rightarrow f:g):x = \begin{cases} f:x & \text{if } p:x = T \\ g:x & \text{if } p:x = F \\ \perp & \text{otherwise} \end{cases}$$

constant: $\bar{x}$ where $\bar{x}:y = \begin{cases} x & \text{if } y \neq \perp \\ \perp & \text{if } y = \perp \end{cases}$

(note that this form has an object as its parameter)

apply-to-all: αf where $\alpha f : \langle x_1, \ldots, x_n \rangle = \langle f:x_1, \ldots, f:x_n \rangle$
$\alpha f : \phi = \phi$

insert: $/f$ where $/f : \langle x \rangle = x$
$/f : \langle x_1, x_2, \ldots, x_n \rangle = f : \langle x_1, /f : \langle x_2, \ldots, x_n \rangle \rangle$

[Note: for anyone who finds functional programming difficult: learning to use functional programming consists, almost entirely, of becoming familiar with a few functional forms and how each of these produces a new function by the use of its parameters. If one really learns the significance and use of the above principal functional forms, one will then understand functional programming.]

$[1,\text{tl}] : \langle A, B, C \rangle = \langle A, \langle B, C \rangle \rangle$

$(\text{null} \circ \text{tl} \rightarrow 1; \text{tl}) : \langle A \rangle = A$

$(\text{null} \circ \text{tl} \rightarrow 1; \text{tl}) : \langle A, B \rangle = \langle B \rangle$

Definitions

Def innerprod ≡ (/+)∘(α×)∘transpose

innerprod:<<1,2>,<3,4>> = (/+)∘(α×)∘transpose:<<1,2>,<3,4>>

= (/+):((α×):(transpose:<<1,2>,<3,4>>))

= (/+):((α×):<<1,3>,<2,4>>) = (/+):<×:<1,3>, ×:<2,4>>

=(/+):<3,8> = +:<3, /+:<8>> = +:<3,8> = 11

Def last ≡ null∘tl → 1; last∘tl

last:<*C*,*D*> = (null∘tl → 1; last∘tl):<*C*,*D*>

= last∘tl:<*C*,*D*> since null∘tl:<*C*,*D*> = null:<*D*> = *F*

= last:(tl:<*C*,*D*>) = last:<*D*>

= (null∘tl → 1; last∘tl):<*D*>

= 1:<*D*> since null∘tl:<*D*> = null:∅ = *T*

= *D*

Algebra of programs

The application of FP programs to objects comprises a *calculus* of functional programming. The corresponding *algebra* of programs has variables that range over over FP programs and operations that are FP functional forms. Laws of the algebra correspond to universally quantified equations of the calculus.

Thus the algebraic law $[f,g]\circ h = [f\circ h, g\circ h]$ asserts that the following equation holds for all functions f, g, h and all objects x:

$$[f,g]\circ h:x = [f\circ h, g\circ h]:x \quad ,$$

i.e., the law asserts that $[f,g]\circ h$ denotes the same function as $[f\circ h, g\circ h]$, regardless of what functions one chooses for f, g, and h.

Notice that such laws do not hold for conventional programs: if h is a program that adds 1 to some variable, then the left side of our law would add 1 to it, whereas the right side would add 2 to it. The reason the law works for FP programs is that they operate on an operand rather than by changing a state.

To establish a law in the algebra one must prove the corresponding proposition in the calculus. Thus to prove the above law, one must prove that the above equation of the calculus holds for all f, g, h and all x; for the proof, see [1].

The basic rules of the algebra are given by its laws. Twenty-four such laws are given in [1]. The reader can easily discover others. Here is a small sample of laws, some of which we shall use later in this paper.

I.1 $[f,g]\circ h = [f\circ h, g\circ h]$

I.2 $/f\circ[g_1,\ldots,g_n] = f\circ[g_1, /f\circ[g_2,\ldots,g_n]]$ for $n\geq 2$

I.5 $1\circ[f,g] = f$ (holds only for objects x where $g:x\neq\perp$)

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

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

IV.1 $[f,\ (p\rightarrow g;h)] = p \rightarrow [f,g];\ [f,h]$

(The validity of this law depends on the fact that the sequence constructor is $\perp$-preserving: $\langle x,\perp\rangle = \perp$.)

3. Forms in a single variable

In the following we shall be discussing arbitrary functional

expressions built from FP functions, functional forms, and a single variable ranging over FP functions. Thus a form Ef (or simply E, or F or G, etc.) is an FP expression for a function in which a ~~subexpression~~ function symbol has been replaced by the variable f. We may also refer to an expression, e.g., h∘[i,f∘j], as a *form in the variable* f, without assigning it a capital letter name. We use lower case roman letters for arbitrary, but fixed, functions in a particular form; e.g., h, i and j above.

In the following discussion we shall refer to these example forms:

Gf = f∘r	Kf = p→f;r
Hf = [i,f]	Lf = αf
If = h∘f	Mf = [r∘f, s∘f]
Jf = h∘[i, f∘r]	Nf = p → Ef; Ff

where lower case roman letters denote arbitrary fixed functions, and Ef and Ff are arbitrary forms.

We can replace occurrences of f in a form Ef by any function or form; thus G(f∘g) = (f∘g)∘r and Gh = h∘r. If the variable f is replaced by a function h, then Eh denotes a particular function.

Forms may be *composed* by replacing the variable of one by the other form; thus IHf = I(Hf) = h∘Hf = h∘[i,f], and IHGf = Jf. We use the name ID for the *identity form*: IDf = f.

4. Linear Forms

The functional form condition, p→f;g plays an essential role in recursive definitions. Thus, in finding the solution of the recursive equation

$$f = p \rightarrow q; Hf,$$

it is necessary to deal with expressions of the form $H(p \rightarrow q; h)$; if this expression can be rewritten as $p' \rightarrow Hq; Hh$, we find we can make easy progress towards the solution, otherwise, progress is difficult. This provides the motivation for the following definition of linear forms. As we shall see, a great many forms are linear.

the predicate transformer of E

Definition Ef is *linear* iff there is a form E_1 such that, for all functions a, b, c :

$$E(a \to b; c) = E_1 a \to Eb; Ec$$ □

We show that the following basic forms are linear:

1) $G(a \to b; c) = (a \to b; c) \circ r = a \circ r \to b \circ r; c \circ r = Ga \to Gb; Gc$
Thus $G_1 = G$.

2) $H(a \to b; c) = [i, (a \to b; c)] = a \to [i,b]; [i,c] = a \to Hb; Hc$
Thus $H_1 = ID$. Similarly, $[\ldots, f, \ldots]$ is linear in f.

3) $I(a \to b; c) = h \circ (a \to b; c) = a \to h \circ b; h \circ c = a \to Ib; Ic$. And $I_1 = ID$.

4) For this example we shall need the following law:

$$p \to (a \to b; c); r = (p \to a; \bar{T}) \to (p \to b; r); (p \to c; r)$$

Thus $K(a \to b; c) = p \to (a \to b; c); r = K_1 a \to Kb; Kc$

where $K_1 a = p \to a; \bar{T}$. Similarly, $p \to r; f$ and $f \to r; s$ are linear in f.

5) If E and F are linear, it is not hard to show that
$Nf = p \to Ef; Ff$ is linear, where $N_1 a = p \to E_1 a; F_1 a$, since

$$N(a \to b; c) = (p \to E_1 a; F_1 a) \to (p \to Eb; Fb); (p \to Ec; Fc)$$

$$= N_1 a \to Nb; Nc$$

The form M is not "basic" since it is merely an instance of $If = h \circ f$ with $h = [r,s]$. Thus M is linear. L is not linear. In the next section we shall see that J, or any other composition of linear forms, is linear.

5. Composition of linear and $\bar{I}$-preserving forms

Composition theorem

If E and F are linear, then so is EF; and $(EF)_1 = E_1F_1$.

proof

$EF(a \to b;c) = E(F_1a \to Fb;\ Fc) = E_1F_1a \to EFb;\ EFc$ □

Example: Since we know that G, H, and I are linear, we can conclude that J=IHG is linear and that $J_1 = G$, since $G_1 = G$ and $H_1=I_1=ID$. Thus $J(a \to b;c) = a \circ r \to Jb;\ Jc$.

Note One must not confuse form-composition, $EFf = E(Ff)$, with function-composition, $Ef \circ Ff$.

$\bar{I}$-preserving forms, composition of $\bar{I}$-preserving forms

A form Ef is $\bar{I}$-*preserving* if $E\bar{I}$ is the everywhere-$\perp$ function $\bar{I}$. All of our example forms G through N are $\bar{I}$-preserving except K and L $(\alpha\bar{I}:\phi = \phi)$. Nf is $\bar{I}$-preserving only if both E and F are. It is obvious that if E and F are $\bar{I}$-preserving, then so is their composition, EF.

6. The linear expansion theorem

We can now state the solution for a large class of recursive equations that define functions. The technical background (limits, continuous functionals, fixed points, etc.) and proof of the linear expansion theorem follow, but readers interested only in solving linear equations can effectively do so without understanding that background and proof. Section 7 discusses the range of applicability of the theorem.

Definition

An *infinite conditional expansion* is an expression of the form

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

where the p_i and q_i are functions. It denotes a function f, such that, for any object x, $f{:}x = q_n{:}x$ if there is a non-negative integer n such that $p_i{:}x = F$ for all $i<n$, and $p_n{:}x = T$; otherwise $f{:}x = \perp$ □

Linear expansion theorem

Let H be a linear and $\bar{1}$-preserving form built from FP functions and functional forms (see [1]). Let f be the least solution (every other solution is an extension of f) of

$$f = p \to q;\ Hf \qquad \text{(LE1)}$$

Then

$$f = p \to q;\ \ldots;\ H_1^n p \to H^n q;\ \ldots \qquad \text{(LE2)}$$

where $H^n = HH^{n-1}$ □

Note that the solution (LE2) of the equation (LE1) gives a necessary and sufficient condition for the 'termination' of f (by the definition of infinite conditional expansion: if the required n does not exist, f is undefined). The solution also gives a case-by-case description of the computation for f.

7. Range of applicability of the linear expansion theorem

As our examples have indicated, the following basic forms are both linear in f and $\bar{1}$-preserving provided that Ef and Ff are also.

Following each form H is its corresponding predicate transformer $H_1 a$ as applied to an arbitrary function a.

(1) $Ef \circ r \quad | E_1 a \circ r$

(2) $r \circ Ef \quad | E_1 a$

(3) $r \to Ef;\ Ff \quad | r \to E_1 a;\ F_1 a$

(4) $Ef \to r;\ s \quad | E_1 a$

(5) $[\ldots, Ef, \ldots] \quad | E_1 a$

Thus the most fundamental functional forms (composition, condition, and construction) are 'transparent' to linearity. Only condition is not transparent to $\bar{1}$-preservation (unless f occurs in the predicate or in *both* consequent positions; e.g., $p \to \bar{1}; r \neq \bar{1}$).

Since the composition of two linear and $\bar{I}$-preserving forms is linear and $\bar{I}$-preserving, a great many forms possess these properties; and for any such form H the linear expansion theorem gives the solution of

$$f = p \to q;\ Hf$$

for any p and q. For example, the following two forms are easily seen to be linear and $\bar{I}$-preserving:

$h \circ f \circ k$ (from (1) above with $Ef = h \circ f$ and r=k)

$h \circ [i, f \circ j]$ (from (2) (5) and (1) above) .

Using these forms for H, the linear expansion theorem reduces, respectively, to the "iteration theorem" and the "recursion theorem" of [1].

8. Limits, continuous forms, least fixed points

Before we can prove the linear expansion theorem we need some standard definitions and facts. Following Manna et al [2], pp492-494, we define below: partial orderings on objects and on functions; the limit of an increasing sequence of functions; continuous forms; the least fixed point of a form. In these definitions a form is treated as a mapping from functions into functions. We then state without proof some facts about continuous forms; for a more complete discussion, further references, and proofs, see [2].

Definitions (with respect to the objects, functions, and forms of some given FP system)

D1) If x and y are objects, then $x \leq y$ iff $x = \perp$ or $x = y$.

D2) If f and g are functions, then $f \leq g$ iff $f:x \leq g:x$ for all objects x.

D3) If $\{f_i | i\in N\}$ where $N=\{1,2,\ldots\}$ is an increasing sequence of functions ($f_i \leq f_{i+1}$ for $i\in N$), then f is the *limit*, $\lim_{i\to\infty}\{f_i\}$, of the sequence iff (1) $f_i \leq f$ for all $i\in N$, and (2) for every g such that $f_i \leq g$, for all $i\in N$, then $f\leq g$.

D4) A form E is *continuous* iff for every increasing sequence of functions $\{f_i | i\in N\}$, $\{Ef_i | i\in N\}$ is an increasing sequence of functions and

$$E \lim_{i\to\infty}\{f_i\} = \lim_{i\to\infty}\{Ef_i\}.$$

D5) A function f is the *least fixed point* of a form E iff f is the least function (under $\leq$) such that $f=Ef$.

Facts

F1) If E is a continuous form, then $\{E^i\bar{I} | i\in N\}$ is an increasing sequence of functions, and E has the least fixed point $\lim_{i\to\infty}\{E^i\bar{I}\}$.

F2) All the basic functional forms given in [1] are continuous.

F3) If E and F are continuous forms, then so is EF.

The above standard facts give the following direct conclusions:

C1) All forms built from the functional forms of [1] are continuous.

C2) Every such form E in one variable has the least fixed point $\lim_{i\to\infty}\{E^i\bar{I}\}$.

C3) If $f_{i+1} = p_0 \to q_0; \ldots; p_i \to q_i; \bar{I}$ for all $i\in N$, then $\{f_i | i\in N\}$ is an increasing sequence of functions and $\lim_{i\to\infty}\{f_i\} = p_0 \to q_0; \ldots; p_n \to q_n; \ldots$

by definition of the infinite conditional expansion on the right and of $\lim_{i\to\infty}\{f_i\}$.

9. Proof of the linear expansion theorem

Using the above three conclusions we can now prove the linear expansion theorem. We shall need the following lemma.

Linear approximation lemma

If H is linear and $\bar{I}$-preserving and

$$Ef = p \to q;\ Hf\ ,$$

then $E^{n+1}\bar{I} = p \to q;\ldots;\ H_1^n p \to H^n q;\ \bar{I}$ for $n\geq 0$ (1)

proof By induction on n.
Now (1) holds for n=0, since $E\bar{I} = p \to q;\ H\bar{I} = p \to q;\ \bar{I}$. We assume that (1) holds for some $n\geq 0$ and prove that it holds for n+1.

$E^{n+2}\bar{I} = E(E^{n+1}\bar{I}) = p \to q;\ HE^{n+1}\bar{I}$ by defs of E^{n+2} and E.

$= p \to q;\ H(p \to q;\ldots;H_1^n p \to H^n q;\bar{I})$ by induction

$= p \to q;\ H_1 p \to Hq;\ldots;H_1^{n+1} p \to H^{n+1} q;\ H\bar{I}$ by linearity of H used n+1 times

$= p \to q;\ldots;H_1^{n+1} p \to H^{n+1} q;\bar{I}$ since H is $\bar{I}$-preserving

Thus (1) holds for n+1 when it holds for $n\geq 0$, and it holds for n=0, hence it holds for all $n\geq 0$. □

Linear expansion theorem

Let H be a linear and $\bar{I}$-preserving form built from FP functions and functional forms (see [1]). Let f be the least solution of

$$f = p \to q;\ Hf\ .$$

Then $f = p \to q;\ldots;\ H_1^n p \to H^n q;\ldots$.

proof

Let $Ef = p \to q;\ Hf$. Then, by C1, E is continuous, and, by C2, the least fixed point of E is $\lim_{i\to\infty}\{E^i\bar{1}\}$. By the linear approximation lemma, $E^{i+1}\bar{1} = p \to q;\ldots;H_1^i p \to H^i q;\bar{1}$. Hence, by C3, the least fixed point of E is $\lim_{i\to\infty}\{E^i\bar{1}\} = p \to q;\ldots;H_1^n p \to H^n q;\ldots$.

Since, by hypothesis, f is the least fixed point of E, the conclusion of the theorem follows. □

10. Forms in two variables, Hfg; linearity of Hff

Here we consider forms Hfg in two variables. We say that Hfg is *linear in f* if there is a form H_1fg such that, for all a, b, c and all g

$$H(a\to b;c)g = H_1ag \to Hbg;\ Hcg\ .$$

Similarly, Hfg is *linear in g* if there is an H_2fg such that, for all a, b, c and all f

$$Hf(a\to b;c) = H_2fa \to Hfb;\ Hfc\ .$$

If Hfg is linear in both f and g and if H_1fg depends only on f and H_2fg depends only on g, then Hfg is *bilinear*. Thus when H is bilinear, we have, for all f, g, h

$$H_1fg = H_1fh \quad \text{and} \quad H_2fg = H_2hg\ .$$

When H is bilinear we shall write H_1f [$=H_1fg$] and H_2g [$=H_2fg$] for the two predicate transformers of H, since H_1fg and H_2fg do not depend, respectively, on g and on f.

Examples

We shall show that most basic forms in two variables are bilinear,

with one key exception. Thus, if E and F are linear forms in one variable, $p \to Ef;\ Fg$ and $[Ef,\ Fg]$ are bilinear, whereas $Hfg = Ef \circ Fg$ is linear in both f and g but not bilinear since $H_1 fg$ depends on both f and g.

(1) Let $Hfg = p \to Ef;\ Fg$. Then

$$\begin{aligned} H(a \to b;c)g &= p \to (E_1 a \to Eb;Ec);\ Fg \\ &= (p \to E_1 a;\bar{T}) \to (p \to Eb;Fg);(p \to Ec;Fg) \\ &= (p \to E_1 a;\bar{T}) \to Hbg;\ Hcg \end{aligned}$$

Similarly,

$$Hf(a \to b;c) = (p \to \bar{T};F_1 a) \to Hfb;\ Hfc\ .$$

Thus $H_1 f = H_1 fg = (p \to E_1 f;\bar{T})$ and $H_2 g = H_2 fg = (p \to \bar{T};F_1 g)$ and H is bilinear.

(2) Let $Hfg = [Ef,\ Fg]$. Then

$$\begin{aligned} H(a \to b;c)g &= [(E_1 a \to Eb;Ec),\ Fg] \\ &= E_1 a \to [Eb,Fg];\ [Ec,Fg] \\ &= E_1 a \to Hbg;\ Hcg\ . \end{aligned}$$

Similarly,

$$Hf(a \to b;c) = F_1 a \to Hfb;\ Hfc\ .$$

Thus $H_1 f = E_1 f$ and $H_2 g = F_1 g$ and H is bilinear.

(3) Let $Hfg = Ef \circ Fg$. Then

$$\begin{aligned} H(a \to b;c)g &= (E_1 a \to Eb;Ec) \circ Fg \\ &= E_1 a \circ Fg \to (Eb) \circ Fg;\ (Ec) \circ Fg \\ &= E_1 a \circ Fg \to Hbg;\ Hcg \end{aligned}$$

Similarly,

$$\begin{aligned} Hf(a \to b;c) &= F_1 a \to (Ef) \circ Fb;\ (Ef) \circ Fc \\ &= F_1 a \to Hfb;\ Hfc\ . \end{aligned}$$

Thus $H_1fg = (E_1f)\circ Fg$ and $H_2g = H_2fg = F_1g$ and Hfg is linear in both f and g but not bilinear since H_1fg depends on both f and g.

Composition

If Hfg is linear in both f and g [resp., bilinear] and if Ef is linear, then $EHfg$ [=$E(Hfg)$], $H(Ef)g$ and $Hf(Eg)$ are linear in f and g [resp. bilinear]; $(EH)_1 = E_1H_1$ and $(EH)_2 = E_1H_2$ and if $Ifg = H(Ef)g$, then $I_1fg = H_1(E_1f)g$ and $I_2fg = H_2(Ef)g$ etc..

Linearity of Hff

If Hfg is linear in both f and g, the question arises: when is Hff linear in f? The following theorem answers this question when H is bilinear by providing a necessary and sufficient condition for Hff to be linear. The problem is more complex when H is not bilinear.

Theorem Let Hfg be bilinear; then Hff is linear in f if and only if there exist functions r and s with values in $\{T,F\}$ except at $\perp$, such that, for all functions a, f, g

(1) $r \twoheadrightarrow Hfg = Hff$

(2) $s \twoheadrightarrow Hfg = Hgg$

(3) $\sim s \;\&\; \sim H_1a \;\&\; H_2a \twoheadrightarrow r$

(4) $\sim r \;\&\; H_1a \;\&\; \sim H_2a \twoheadrightarrow s$.

And, when $Gf = Hff$ is linear, then

(5) $G_1a = (H_1a \;\&\; H_2a)\vee(r \;\&\; H_1a \;\&\; \sim H_2a)\vee(s \;\&\; \sim H_1a \;\&\; H_2a)$.

(Where $\sim r$ = not$\circ$r and $p \twoheadrightarrow f{=}g$ holds when $p{:}x{=}T$ and $f{:}x{=}g{:}x$, or $p{:}x{=}F$.)

Note This theorem, while valid, is not very helpful. Future revisions will give more helpful theorems for characterizing the class of linear forms and the linearity of Hff.

~~proof (If. The existence of r and s satisfying (1) through (4)~~ implies that ~~Hff is~~ linear.) We shall show that our hypothesis implies that G, as defined above, is linear, with G_1 being the ~~form defined in (5). To do so we choose functions a, b, c and object x and show that, for this arbitrary choice,~~

References [1] Backus, John, "Can programming be liberated from the von Neumann style; a functional style and its algebra of programs" CACM 21 8 (8/78)

[2] Manna, Z., Ness, S., and Vuillemin, J. "Inductive methods for proving properties of programs" CACM 16 8 (8/73).