Function Level Programs as Mathematical Objects

John Backus

IBM Research Laboratory
5600 Cottle Road
San Jose, California 95193

1. Introduction

Most programs written today are "object-level" programs. That is, programs describe how to combine various "objects" (i.e., numbers, symbols, arrays, etc.) to form other objects until the final "result objects" have been formed. New objects are constructed from existing ones by the application of various object-to-object functions such as + or matrix inversion.

Conventional, von Neumann programs are object level; "expressions" on the right side of assignment statements are exclusively concerned with building an object that is then to be stored. Lambda calculus based languages, such as LISP and ISWIM [Landin 66], are also, in practice, object level languages, although they have the means to be more.

To see that "lambda style" programs are primarily object level, consider the definition of a new object-to-object function (the usual kind of definition), $f = \lambda x.E$; here x must be an object variable (since the argument of f is an object) and E must denote an object (since f's result is an object). Typically E is an expression involving the application of object-forming functions to object variables and constants. A few object-forming functions are used that have both function and object arguments.

If we include object variables in the term "objects", then the object level view of programming is one of building objects by the application of existing programs (object-forming operations) to objects. Lambda style programs then build a new program from the result-object by abstracting the object variables.

There is a different approach to building programs, a "function level" one. In the function level style a program is built directly from the programs that are given at the outset, by combining them with program-forming operations (PFOs). Thus instead of applying the given programs to objects to form a succession of objects culminating in the result object, the function level approach applies program-forming operations to the given programs to form a succession of programs culminating in the desired program.

Thus the object level approach invites the study of the space of objects under the object-forming operations, and of the algebraic properties of those operations. On the other hand, the function level approach invites study of the space of programs under the program-forming operations and of the algebraic properties of these PFOs.

The study of the space of objects under the object-forming operations is now called the study of data types. It has advanced from focussing on the objects themselves and their structure to a primary concern with the object-forming operations and their structure as given by certain axioms or algebraic laws. This movement toward the algebraic study of data types is exemplified by [Burstall & Goguen 80], [Guttag & Horning 78], [Thatcher, Wagner & Wright 78], and [Zilles 79].

One goal of the function level approach is to now move our attention in a similar fashion from programs themselves and their "structure" to the program-forming operations and their structure as given by various algebraic laws. Just as the study of data types has brought out that "objects" comprise a "mathematical" space by emphasizing the algebraic properties of the operations on that space, so the function level approach offers the possibility of making the set of programs a mathematical space by emphasizing the algebraic properties of the program-forming operations over the space of programs.

One purpose of this paper is to contrast

the object level and the function level approaches to programming. The latter approach is not well understood by many users of lambda style languages, who mistakenly regard it as merely a restrictive variant of the lambda style, which is an object level style.

Section 2 is a very brief summary of the FP style of programming [Backus 78] which we shall use as an example of the function level style. Section 3 discusses what we mean by a set of "mathematical objects", the advantages of a concept of "programs" in which programs comprise a set of mathematical objects, and why the object level view blocks, whereas the function level view advances this objective.

Section 4 compares object level function definitions and function level ones. It gives a mapping, "lift", that transforms any object level definition into a corresponding function level one, where object variables in the first kind are replaced by function variables in the second. This introduction of function variables makes "extended" FP definitions as readable as object level ones, thereby removing one of the principal difficulties people have experienced with the FP variable-free style. Extended definitions also turn out to be useful algebraic laws about the functions they define.

Section 5 shows why variables are essential in object level definitions but not in function level ones. It shows how function variables in the latter can be removed, giving a "proper", variable-free definition of the same function.

Section 6 discusses two further advantages of the function level view. One is the ability to use only strict functions and thereby have bottom-up semantics (the simplest kind) that are "safe" for computing least fixed points. The other is the existence of function level definitions that are not the "lifted" image of any object level one. These "terse" function level definitions represent a more powerful style of programming not available at the object level; they are often easier to understand and to reason about.

Section 7 compares lambda style and FP style programs. It shows that the former tend to be unstructured whereas the latter are highly structured. The lambda style tends to obscure function level identities that are clear in function level expressions; it obscures and complicates function level reasoning about programs.

2. Brief summary of the FP function level style of programming

Here we give the shortest description we can of the essential elements of FP programming. For a complete description see [Backus 78].

The "objects" that FP programs map into one another comprise a set whose primary property is closure under "sequence formation": if $x_1,\ldots,x_n$ are objects, then the sequence $\langle x_1,\ldots,x_n\rangle$ is an object. Thus the objects can be built from any set of atoms (which should contain numbers, truth values, symbols, etc.).

FP programs are functions f that each map a single object x into another. We write f:x for the object that results from applying the function f to the object x. Functions are either primitive (given) or are built from the primitives by program-forming operations, or PFOs (also called combining forms or functional forms). One such PFO, constant, transforms an object x into the constant-valued function $\bar{x}$, where

$$\bar{x}:y = x$$

for all objects y (except the "undefined" object, which all functions map into itself). Next there are the three principal PFOs of FP, which map, respectively, two, n≥1, or three functions into a single function as follows:

composition $f\circ g$

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

construction $[f_1,\ldots,f_n]$

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

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

condition $(p\rightarrow f;g)$

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

Other PFOs are apply-to-all, αf, where $\alpha f:\langle x_1,\ldots,x_n\rangle = \langle f:x_1,\ldots,f:x_n\rangle$; right insert, /f, where $/f:\langle x\rangle = x$ and $/f:\langle x_1,x_2,\ldots,x_n\rangle = f:\langle x_1,/f:\langle x_2,\ldots x_n\rangle\rangle$; and left insert \f, which works like right insert except that the computation associates to the left. Right or left insert of f applied to the empty sequence gives the right or left unit of f if f has such a unit.

In addition to being constructed from primitives by PFOs, a function may be defined recursively by an equation, the simplest kind (without variables, more on equations with variables later) has the form:

$$f = Ef$$

where Ef is an expression built from primitives, defined functions, and the function symbol f by PFOs.

Of the many possible primitive functions here we note only the basic "selector" functions, denoted by 1,2,... , where, e.g.,

$1:\langle x_1,\ldots,x_n\rangle = x_1$, $2:\langle x_1,x_2,\ldots,x_n\rangle = x_2$, etc.

3. Programs as mathematical objects.

When we say that a set S is a set of "mathematical objects", we are speaking of S and a set of operations on S (that map S^n into S) such that these operations are interrelated by algebraic laws. The "strongest" laws are "symmetric" ones like the distributive law that relate two operations A and B, in which operation A, combining objects formed by operation B, is expressed as operation B combining elements formed by operation A. Other "weaker", non-symmetric laws may relate several operations. The more and the stronger laws that relate the operations on S, the "stronger" is their algebraic structure and the stronger is the "mathematical structure" of the set S.

If the operations of a set of mathematical objects obey a strong set of interrelating laws, then there will exist a large body of general theorems about the set. Thus there are many such theorems concerning the set of numbers under addition and multiplication, or about rings in general, or about categories, etc. On the other hand, if the operations on a set obey only a few weak laws, there will be few general theorems. Thus the set of numbers under addition and square root hardly deserves to be called a set of "mathematical objects" since there are few general laws and theorems about it (unless other operations, e.g., multiplication, are added and obey laws relating to the other two).

If programming is to become a truly mathematical discipline, it is important to find a way to make the space of programs a mathematical one with respect to the operations (PFOs) over that space (as distinguished from the space of objects and the operations (programs) over it). We would then be able to produce a large body of carefully proven general theorems (whose universally quantified variables would denote programs). These theorems would express many reliable, useful facts about large classes of programs and about the solutions of equations whose "unknowns" are programs, just as ordinary algebra gives us theorems about numerical expressions and, for example, the general solution for all quadratic equations. This saves us the labor of repeatedly and separately solving many individual problems.

If programs were themselves mathematical objects, we might derive many theorems that would guide us -- at the outset, not after the fact -- in how to structure a proposed program, and later help us prove its correctness or optimize it. Our reasoning could be direct: theorems would concern the actual PFOs we use to construct our programs and the structure of the programs themselves.

Von Neumann programs do not form a set of mathematical objects for reasons relating to their object level nature; for further discussion of this question see [Backus 81a].

Traditional "functional" programs, those written in the lambda style of LISP or ISWIM, also tend not to form a set of mathematical objects. These programs use lambda abstraction as their principal program-forming operation. This PFO obeys no strong algebraic laws of the kind that are helpful in transforming and reasoning about programs. Its use in building an object-transforming program requires that it be applied to an object expression, one built by object-forming operations from object variables, whereupon lambda abstraction "elevates" it to a function expression.

Thus the use of lambda abstraction as the PFO of the lambda style means that most programs are largely object level; existing programs are combined, not by PFOs directly, but by application to "objects" to form an expression for a "result object" that is then elevated to a program by abstraction of object variables.

Since von Neumann programs and traditional functional programs are not mathematical objects, we reason about them by mapping them into a logical or mathematical domain, reason about their images in that domain and then translate the results of that reasoning back to the realm of programs. Even in these other, richer domains we do not have a satisfactory set of laws and generally useful theorems relating to programs.

Our inability to change our notion of "program" into a more orderly concept has resulted from our tendency to keep to the object level view of programs in their active role as entities that combine objects. This view has caused us to neglect the study of program-forming operations themselves in their active role as entities that combine programs.

The main purpose of this paper is to point out the importance and usefulness of this second, function level view of "programs" as entities operated upon by PFOs. Only by taking this view can we hope to change "programs" into mathematical objects.

FP programs serve as one example of "programs" that are mathematical objects; we need to develop other examples. Thus the three principal PFOs of FP obey the kind of symmetric interrelating laws required of the operations on a set of objects with a strong mathematical structure; they obey the following interrelating laws (in addition to laws concerning a single operation, such as associativity of composition and various laws concerning properties of condition):

Composition and construction

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

This is a symmetric law since it expresses composition involving construction as construction involving composition, just as the distributive law of ordinary algebra expresses multiplication involving addition as addition involving multiplication:

$$(a+b)c = ac + bc\ .$$

Composition and condition

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

Again both laws are symmetric ones, composition involving a condition is expressed as a condition involving composition.

Construction and condition

$$[f,(p\to g;h)] = p \to [f,g];[f,h]$$

(The same symmetry principle applies to this law, which is only an example of many similar ones.)

In addition to these "strong" basic laws there are many others, usually "weak" ones, some involving the other PFOs, others involving particular constants (that is, particular functions), corresponding to laws about units and other entities. Thus, e.g., the identity function is the "unit" of composition: $f\circ id = id\circ f = f$. The law

$$h = [1\circ h, 2\circ h] \quad \text{where} \quad h = [f,g]$$

corresponds to the important "natural" one-one correspondence

$$h \leftrightarrow [1\circ h, 2\circ h]$$

in category theory [Mac Lane 71, p2].

From the laws governing the PFOs and primitive functions of FP one can derive many general theorems and the solutions for several large classes of functional equations. For early examples of such results see [Williams 80], [Backus 81], and [Backus 78].

The approach of viewing programs as mathematical objects is a fairly recent one and the development of theorems in a domain rich in algebraic laws has only begun. When the mathematical community begins to take up this enterprise we can perhaps hope for some really interesting new insights into the mathematical structure of the domain of programs with their PFOs. As some second-order PFOs -- operations for combining first-order PFOs -- are adjoined to such systems we shall have need of insight to help us identify the cleanest, most useful properties for these operations (and for new first-order PFOs).

4. Object level and function level definitions; lifting object expressions to function expressions: an example.

A typical object level definition looks like this (we use FP syntax to clarify later comparisons):

$$f{:}\langle x,y\rangle = \underline{\text{if}}\ \ \text{neg}{:}x\ \ \underline{\text{then}}\ \ 0\ \ \underline{\text{else}}\ \ +{:}\langle \text{sqrt}{:}x, y\rangle\ .$$

This defines a function f that sends a pair of numbers <x,y> into 0 when x is negative and otherwise into $\sqrt{x} + y$. Notice that the corresponding definition using lambda abstraction concerns itself with exactly the same objects and constructions:

$$f = \lambda(x,y).(\underline{\text{if}}\ \ \text{neg}{:}x\ \ \underline{\text{then}}\ \ 0\ \ \underline{\text{else}}\ \ +{:}\langle \text{sqrt}{:}x, y\rangle).$$

Therefore, for the present, we shall dispense with lambda abstraction (we shall have more to say about the "lambda style" of programming later) and use traditional object level definitions as above.

Notice that object expressions, like that on the right above, are built up from atomic objects (e.g., 0) and object variables (x and y) by the application of ordinary object-forming functions (neg, +, sqrt) and by two object-forming operations that are treated specially, if-then-else and sequence-formation or tupling (that maps n objects $x_1,\ldots,x_n$ into the single object $\langle x_1,\ldots,x_n\rangle$).

Thus objects and object variables are object expressions and if e, $e_1,\ldots,e_n$ are object expressions and if f is an object-forming function, then

(a) $(f{:}e)$

(b) $\underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3$

(c) $\langle e_1,\ldots,e_n\rangle$

are object expressions.

It is clear that an object expression containing no variables denotes an object that is the value of the expression, thus the expression +:<2,3> denotes the value 5.

Now consider the following mapping that "lifts" object expressions e onto function expressions $\tilde{e}$ = lift(e) .

The mapping "lift" : {e} → {$\tilde{e}$}

We give the values of lift(e) for the five possible cases:

a) e is an object:

lift(e) = $\tilde{e}$

b) e is an object variable:

$$lift(e) = e$$

where, e.g., lift of the object variable x is the function variable $\tilde{x}$.

c) e is an application (f:d) where f is an object-forming function and d is an object expression:

$$lift((f:d)) = f \circ \tilde{d}$$

d) e is formed by if-then-else:

$$lift(\underline{if}\ e_1\ \underline{then}\ e_2\ \underline{else}\ e_3) = \tilde{e}_1 \to \tilde{e}_2; \tilde{e}_3$$

e) e is a sequence:

$$lift(\langle e_1,\ldots,e_n\rangle) = [\tilde{e}_1,\ldots,\tilde{e}_n]$$

(Note that if-then-else maps three objects into one, whereas condition, on the right in (d), maps three functions into one.) Here is an example:

$$lift(+:\langle g:x, h:3\rangle) = +\circ[g\circ x, h\circ\bar{3}]\ ,$$

where x is an object variable on the left and a function variable on the right.

Notice that an object expression e without variables is lifted onto a function expression $\tilde{e}$ whose function-value is the constant-valued function $\bar{v}$, where v is the object-value of e. This relationship is indicated by the following commuting diagram:

$$\begin{array}{ccc} e & \xrightarrow{\quad lift \quad} & \tilde{e} \\ \text{object value}\ \downarrow & & \downarrow\ \text{function value} \\ v & \xrightarrow[\quad lift \quad]{} & \bar{v} = \tilde{v} \end{array}$$

Thus, for example:

$$value(+:\langle 2,3\rangle) = 5$$

$$lift(+:\langle 2,3\rangle) = +\circ[\bar{2},\bar{3}]$$

$$function\text{-}value(+\circ[\bar{2},\bar{3}]) = \bar{5} = lift(5)$$

But keep in mind that most function expressions are not lift-images of object expressions.

Using the lift mapping we can write the function level version of our earlier example:

(1) object level:

$$f:\langle x,y\rangle = \underline{if}\ neg:x\ \underline{then}\ 0\ \underline{else}\ +:\langle sqrt:x, y\rangle$$

(2) function level:

$$f\circ[x,y] = neg\circ x \to \bar{0};\ +\circ[sqrt\circ x, y]$$

Just as the object level versrion (1) is to hold for all object values of the variables x and y, so the function level version (2) is to hold for all function values of the variables x and y.

If, for the moment, we think of the variables x and y as having fixed object or function values respectively in the two equations, denoted x and y in both cases, then in (1) <x,y> is the value of f's argument, whereas in (2) [x,y] is the function that constructs f's argument. Thus when $f\circ[x,y]:w$ is computed for any object w, f "sees" the argument <x:w,y:w>. When the right side of (2) is applied to the same object w, then every occurrence of x will produce x:w, just as it does on the left side; similarly every occurrence of y will produce the same object on both sides of the equation. So the functional equation (2) asserts that, to compute $f\circ[x,y]:w = f:\langle x:w, y:w\rangle$ one computes

$$(neg\circ x \to \bar{0};\ +\circ[sqrt\circ x, y]):w\ ,$$

which is $\bar{0}:w = 0$ if $neg:(x:w) = T$, or

$$+:\langle sqrt:(x:w), y:w\rangle$$

if $neg:(x:w) = F$.

This computation for the "extended" FP definition (2) corresponds exactly with what we understand from (1), except that the (variable) object x in (1) is represented in (2) by x:w, where both the function x and the object w are variable elements (but in (2) the variable object w is only implicit by virtue of the fact that f=g means f:w = g:w for all objects w).

Another way to see that (1) and (2) define the same function f is to compare the object level computation of (1) for a particular argument, say <4,3>, with the corresponding function level calculation for (2).

(1') $f:\langle 4,3\rangle = +:\langle sqrt:4, 3\rangle$

since $neg:4 = F$

$= 5$

Lifting 4 and 3 yields the constant-valued functions $\bar{4}$ and $\bar{3}$. Thus the lift of <4,3> is the function $[\bar{4},\bar{3}]$, and (2) becomes

(2') $f\circ[\bar{4},\bar{3}] = neg\circ\bar{4} \to \bar{0};\ +\circ[sqrt\circ\bar{4},\ \bar{3}]$.

It is easy to see that $neg\circ\bar{4} = \bar{F}$, $(\bar{F}\to g;h) = h$, $sqrt\circ\bar{4} = \bar{2}$, and $+\circ[\bar{2},\bar{3}] = \bar{5}$ are trivial function level identities (if you wish you can view them as lifted from their object level counterparts). From these we obtain at the function level

$$f\circ[\bar{4},\bar{3}] = \bar{5}\ .$$

And in general, if (1) yields $f:\langle x,y\rangle = z$, then (2) will give

$$f\circ[\bar{x},\bar{y}] = \bar{z}\ .$$

Thus (1) expresses the fact that the object $f:\langle x,y\rangle$ is the same as the object-value of the expression on the right, no matter what objects one chooses as x and y. As expected, (2) expresses the "same" thing

except in its "lifted" version: the function $f \circ [x,y]$ is the same as the function-value of the expression on the right, no matter what functions one chooses as x and y.

5. Why the function level is preferable to the object level; deriving variable-free function level definitions from those with function variables: example continued.

What, if any, are the advantages of using the lifted version of a definition? (We continue to refer to definitions (1) and (2) of the last section.) Now (1) allows us to replace the object $f:\langle x,y\rangle$, for any objects x and y, by the object-expression on the right. This forces us, in reasoning about the program f or some program that uses f, to descend from the domain of programs to the domain of objects. Clearly it is simpler and more direct if we can reason about the program f in the program domain without referring to objects at all; the function level definition (2) allows us to do just that. Thus (2) allows us to replace the program $f \circ [x,y]$, for any programs x and y, by the program-expression on the right. Therefore (2) is a general, useful law about f that can be used to reason about f and programs using f without leaving the domain of programs.

Many programs that use f will construct an argument for f with a function of the form [g,h], and so $f \circ [g,h]$ will occur in the program using f. Thus (2), regarded as a law about f, allows us to replace $f \circ [g,h]$ with the function expression

$$\text{neg} \circ g \to \bar{0}; + \circ [\text{sqrt} \circ g, h] \ ;$$

this elimination of f may help us to reason about or transform the using program.

In addition to serving as a useful algebraic law about f, (2) has another advantage over (1). In (1) the variables x and y are absolutely essential in defining the function f: the object level approach, by definition, is dedicated to describing functions by describing their values for all arguments, in this case denoted by x and y. But in (2), as we shall see, the function variables x and y are not essential in defining f; their purpose is (a) to make the definition more readable and (b) to provide the variables of an algebraic law about f.

Let us try to derive a "proper", variable-free definition of f from the definition (2). Since (2) is to hold for all functions x and y, we might ask if there are particular functions s_1 and s_2 such that $[s_1,s_2]$ is the identity on pairs, the domain on which f is to be defined. If s_1 and s_2 exist, then, in the domain of pairs, $f \circ [s_1,s_2] = f$, and so, if we substitute s_1 and s_2 for x and y in (2), we get

$$(3) \quad f = \text{neg} \circ s_1 \to \bar{0}; + \circ [\text{sqrt} \circ s_1, s_2]$$

For $[s_1,s_2]$ to be the identity on pairs, it is necessary and sufficient that the functions s_1 and s_2 satisfy

$$s_1 \circ [x,y] = x \quad \text{and} \quad s_2 \circ [x,y] = y$$

when both x and y are defined. Now of course the selector functions 1 and 2 have this property and are the functions that we want for s_1 and s_2, since [1,2] is the identity function on pairs:

$$[1,2]:\langle x,y\rangle = \langle 1:\langle x,y\rangle, \ 2:\langle x,y\rangle\rangle = \langle x,y\rangle.$$

Substituting these values in (3) gives

$$(4) \quad f = \text{neg} \circ 1 \to \bar{0}; + \circ [\text{sqrt} \circ 1, 2] \ ,$$

which defines the same function as (1) and (2) on the domain of pairs. But the right side of (4) is defined for sequences of any length whose first two elements are numbers. If we want f to be undefined for all non-pairs, we must insert a predicate that ensures this:

$$(5) \quad f = \text{eq} \circ [[1,2],\text{id}] \to E; \ \bar{I}$$

where eq is the equality function, id is the identity function, E is the right side of (4), and $\bar{I}$ is the everywhere-undefined function. Now (5) defines exactly the function we intend by (1) and (2): it is undefined for arguments that are not pairs of numbers.

In the two preceding sections we have tried to give an informal idea of how "extended" FP definitions using function variables can be used to make easily readable definitions that correspond exactly to object level definitions. We have indicated how these extended definitions serve as useful laws for reasoning about the defined function and others that use it. And we have indicated how extended definitions can be converted to proper, variable-free ones.

The technical definitions and theorems needed to make the notion of extended definitions precise and to prove the equivalence of the functions defined by an extended definition and by the corresponding proper definition are to be found in [Backus 81].

6. Safe computation rules; "terse" function level programs.

Another advantage of the function level approach concerns the elimination of the object level function if-then-else, a function that must be non-strict to be meaningful (i.e., it must be defined for some undefined arguments). Object level functional languages require at least this one non-strict function. As Manna et al. [73] and Cadiou [72] observe, bottom-up

computation rules are not "safe" for computing least fixed points in systems with non-strict functions, thus bottom-up rules (giving the simplest operational semantics) are not safe in object level languages using if-then-else, hence the simplest operational semantics are incompatible with fixed point semantics in such languages.

On the other hand, function level languages can use condition, the lifted version of if-then-else, a non-strict functional; all functions can then be strict and bottom-up rules become safe for computing least fixed points. It is interesting to note that the object level view has so dominated our thinking that Manna and Cadiou, despite the thoroughness of their studies of fixed points, never considered the possible existence of systems with all strict functions and hence of safe bottom-up computation. (For a fuller discussion of this point see [Williams 80].)

Perhaps the most important advantage of the function level approach is that it makes possible a more powerful and terse style of programming than is possible at the object level, a style that often has no object level counterpart of comparable simplicity. Furthermore, this terse style is often much easier to reason about. Let me illustrate this point with an example similar to one in [Williams 81].

Williams defines the function "length" at the function level as follows:

(6) length = /+ ∘ $\alpha\bar{1}$.

This means (a) apply (with apply-to-all of $\bar{1}$) the everywhere-1 function to every member of an argument sequence, giving a sequence of all 1's, then (b) sum this (with /+), thereby giving the length of the original.

This definition uses the PFOs insert (/), composition (∘) and apply-to-all (α) to build length from the functions + and $\bar{1}$. It is not clear what object level definition might approximate (6) without these PFOs. Thus, unlike our earlier example, (6) is not the lift of any object level definition. The usual object level definition of length is recursive:

(7) length:x = if null:x then 0
else +:<1, length:(tl:x)>

The lifted version of (7), with the variable removed, is

(8) length = null → $\bar{0}$; +∘[$\bar{1}$, length∘tl] .

One indication of the usefulness of the general theorems of the FP style is that one can use the general solution for all "linear" equations [Backus 81] to prove that (8) and (6) define the same function.

But the proof of a theorem about length is often much easier when it starts with the closed form (6) than when it starts with the recursive definitions (7) or (8). For example, the following theorem has a relatively simple proof when length is defined by (6), whereas its proof is harder and involves induction when length is defined by (7).

THEOREM length∘apndr∘[f,g] = +∘[length∘f,$\bar{1}$]

for all functions f and g when g is defined, where apndr is append on the right:

apndr:<<x_1,...,x_n>,y> = <x_1,...,x_n,y> .

This theorem is a lifted version of one that says: the length of a sequence s = f:x with an element (g:x) added on the right, is one greater than the length of s.

The proof uses the following identities:

(a) αh∘apndr∘[f,g] = apndr∘[αh∘f, h∘g]

(b) if h is associative then \h = /h

(c) \h∘apndr∘[f,g] = h∘[\h∘f, g]

(d) $\bar{x}$∘g = $\bar{x}$

proof We transform the left side of the equation into the right by the use of the above identities.

length∘apndr∘[f,g]

= /+∘$\alpha\bar{1}$∘apndr∘[f,g] by def of length

= /+∘apndr∘[$\alpha\bar{1}$∘f, $\bar{1}$∘g] by (a)

= \+∘apndr∘[$\alpha\bar{1}$∘f, $\bar{1}$] by (b)(+ assoc) and (d)

= +∘[\+∘$\alpha\bar{1}$∘f, $\bar{1}$] by (c)

= +∘[/+∘$\alpha\bar{1}$∘f, $\bar{1}$] by (b)

= +∘[length∘f, $\bar{1}$] by def of length

□

The point of this example is that, after becoming familiar with the function level style, one can construct functions that are more tersely expressed, easier to understand and to reason about.

7. Comparison of FP and lambda style programming; object vs. function level, structure and reasoning.

So far we have discussed a number of reasons for preferring a function level style over an object level one. Now I would like to extend that discussion with a more specific comparison of the FP style with the "lambda style", that used in lambda calculus based languages like LISP and ISWIM [Landin 66], from the viewpoint of program structure.

In this section we shall point out some of the basic differences between the FP and

the lambda styles of functional programming. We shall suggest that (a) the FP style leads to "structured" functional programs, whereas the lambda style leads to unstructured ones, and that (b) the FP style encourages reasoning at the "function level" whereas the lambda style leads to reasoning at the "object level". In general, we suggest that the FP style offers a framework in which one can perceive and reason about program structure, truths, and transformations at a higher level of generality than that presently available for reasoning about lambda style programs.

What do we mean by the "structure" of a program? In a conventional, von Neumann language, a program is "structured" if it has single entry and exit points and is built up from subprograms of this same kind by a small set of program-forming operations (PFOs). For example, the program

p = if a then (while b do c) else d

is built from an expression a and two programs, (while b do c) and d by the PFO if-then-else. Its first subprogram is built by the PFO while-do. Thus the "structure" of p is the operation if-then-else composed with while-do in its first program-argument position.

In similar terms FP programs are completely structured. For example, the program

(1) $f = p \to q; h \circ [r,s]$

employs the PFO, condition, to build f from three programs, p, q, and $h \circ [r,s]$, where the third of these is built by the PFO composition from the program h and the program formed by the PFO construction from programs r and s.

In contrast to the FP emphasis on the use of program-forming operations to build structured programs at the function level, the lambda style emphasizes object-forming operations and is more often concerned with combining objects than with combining functions. For example, the lambda style analogue of (1) is

(2) $f = \lambda x.(p:x \to q:x; h(r:x,s:x))$

In (1) we simply combine the functions p, q, h, r, and s to form the function f. In (2) we are given the same functions to start with and want to define the same result. But we proceed quite differently. We begin by introducing an "object" x, and from it we form the objects p:x, q:x, r:x, and s:x, combine r:x and s:x to form the object h(r:x,s:x), and finally we combine p:x, q:x and h(r:x,s:x) with the object-forming operation conditional to form the "result", an object. Only at this point do we use the primary program-forming operation of the lambda style, lambda abstraction [Church 41]. By writing "$\lambda x.$" in front of the object we have so far produced, we transform it into the desired function.

As the above example shows, the typical method of building a function f in the lambda style is to immediately descend from the level of functions (those supplied to build f) to the level of objects, and there combine objects to form the desired "result-object". One then ascends to the function level by abstraction of the original "objects", i.e., the object variables. This down-then-up-again approach and its concern with object-forming operations avoids the use of PFOs that could achieve the same result more directly; therefore it obscures the "structure" of the program.

It is clear that it is the use of lambda abstraction as the principal PFO that leads to the object-oriented, structure-obscuring nature of the lambda style. If a function f maps objects into objects and is built by lambda abstraction, $f = \lambda x.E$, then x must be an object variable (since the argument of f is an object) and E must denote an object (since the result of f is an object).

The relative structurelessness of lambda style programs makes it difficult to recognize even simple relationships between programs. For example, it is harder to recognize an instance of the following simple identity in the lambda style,

(3) $\lambda y.((\lambda x.\langle f:x,g:x\rangle):(h:y)) =$
$\lambda y.\langle \lambda x.(f:(h:x)):y,\ \lambda x.(g:(h:x)):y\rangle$

than it is to recognize the more structured FP law

(4) $[f,g] \circ h = [f \circ h, g \circ h]$,

which is the same identity expressed at the function level, without the superfluous application of functions to "abstract" objects that is required by lambda abstraction and that demotes this statement about functions to substatements about objects. (The identity (3) is unlikely to be recognized as a useful function level identity for a second reason: if it is simplified to eliminate x, both sides reduce to

$\lambda y.\langle f:(h:y),\ g:(h:y)\rangle$

and the functional relationship has vanished.)

The simpler structure of FP-style programs is important if programming is to become a mathematical discipline that is useful to the ordinary practitioner. Such a discipline can be helpful by providing a body of carefully proven general laws and theorems about programs. The simpler the structure of programs, the more easily can a programmer recognize that the major structure of his program provides an instance of one or more theorems that will help him prove its correctness or make it more effi-

cient.

If it is difficult to recognize an instance of a simple law like (4) when expressed in the lambda style, then the chances of recognizing instances of important theorems are very small. For example, Williams [80] proves the following theorem for all $n \geq 1$ and for all functions f, a, b, and c:

$$(5) \quad [f, 2\circ a]^n \circ [b,c] = [\backslash f\circ[b,c,a\circ c,\ldots,a^{n-1}\circ c],\ a^n\circ c],$$

where $f^{n+1} = f\circ f^n$. It is unlikely anyone is going to recognize an instance of such a theorem when his program and the theorem are expressed in the lambda style, simply because of the sheer complexity of its statement.

We have seen how the lambda style tends to obscure the function level structure of a program. The need to move from the function level down to the object level leads to an over-reliance on object level reasoning. In either the FP or lambda style it is sometimes necessary to reason at the object level: to show that f=f' one may have to show that for every object x, f:x is the same object as f':x. (Such reasoning can be done in a lifted form at the function level.) But often such reasoning is unnecessary and tends to lose touch with important function level identities. For example, consider the following object level definitions of f and f'.

$$f:x = 4x + g(3x) \quad \text{where} \quad g:x = even(x) \to x;\ 2x\ .$$

$$f':x = even(3x) \to 7x;\ 10x\ .$$

In trying to prove f:x = f':x at the object level, one may be lead into considering various cases and making various calculations. If, on the other hand, f is expressed at the function level (using m_i for "multiply by i"), then we get

$$f = +\circ[m_4,\ g\circ m_3] \quad \text{where} \quad g = even \to id;\ m_2$$

or

$$f = +\circ[m_4,\ (even \to id;\ m_2)\circ m_3]\ .$$

Now anyone familiar with FP theorems would either (a) recognize the first expression for f as a form linear in g [Backus 81] and use the properties of such forms to obtain the desired result or (b) he would recognize the combined expression for f as an instance of the simple theorem

$$h\circ[i,(p\to q;r)\circ j] = p\circ j \to h\circ[i,q\circ j];\ h\circ[i,r\circ j]$$

that gives, in this case,

$$f = even\circ m_3 \to +\circ[m_4, id\circ m_3];\ +\circ[m_4, m_2\circ m_3]$$

or, after simplification,

$$f = even\circ m_3 \to m_7;\ m_{10}\ .$$

This is an extremely simple example; consequently the object level reasoning needed to show that f:x = f':x for every x is simple. However, if instead of the subprograms m_i we had used others requiring complex calculations, the object level reasoning might have become much more difficult unless the reasoner happened to recognize the usefulness of the function level identities that the FP approach makes clear. Much object level reasoning can be compared to proving (a+b)c = ac+bc for given numbers a, b, and c by doing arithmetic instead of by using algebra. If a, b, c are small numbers, either method is viable, but if they are very large, then algebra is definitely better!

One further difference between the lambda and FP styles is worth noting. Languages in the former style tend to use functions of more than one argument, whereas all FP functions are unary. Thus in the lambda style, if h(x,y) is a function of two arguments and the values of g(x) are pairs <y,z>, then the following locutions are needed to express the function f, where

$$f:x = h(y,z) \quad \text{where} \quad g:x = \langle y,z\rangle$$

or, using selector functions and lambda abstraction,

$$f = \lambda x.h(1:(g:x),\ 2:(g:x))\ .$$

In the FP style h would be a function on pairs, h:<y,z>, therefore the definition of f would be

$$f = h\circ g\ .$$

Thus the use of n-ary functions in the lambda style is another important factor that obscures program, function level structure and leads to object level reasoning. (The PFO "construction" is important in a style using only unary functions; it is the PFO used to build a subprogram to create an argument for a function on tuples; e.g., if h is defined on pairs, then f = h∘[r,s] corresponds to f:x = h(r:x, s:x) .)

To summarize: use of lambda abstraction as the primary program-forming operation obscures function level structure of programs built with it. By requiring a descent to object-forming operations, it often leads to unnecessarily complicated object level reasoning, reasoning that may fail to take advantage of function level theorems that can be seen to apply when the function level structure of a program is made clear.

At this point we must note that we are not proposing the FP style as a panacea. When object level reasoning is necessary, as it often is (e.g., when a proof depends on the detailed properties of a primitive function), then lambda expressions can be

helpful (but of course we would propose using the lifted function level analogue).

The lambda abstraction PFO is more powerful than any or all of the PFOs of the FP style. Using lambda abstraction one can define all of the FP PFOs and an infinity of others. But this reminds one of the relationship between Fortran and the conventional "structured languages". Using IFs and GOTOs in the former, one can model the "structured" PFOs allowed in the latter and one can model an infinity of other PFOs.

Thus, just as one can write "structured" programs in Fortran, so one can write "structured" functional programs in LISP or ISWIM. But just as it is easier to see the structure of a program written in Pascal rather than in Fortran, so it is easier to see the structure of a functional program written in FP rather than in LISP or ISWIM, since FP and Pascal are designed to emphasize program structure whereas LISP and Fortran tend to obscure it.

It is perhaps going too far to say that lambda style languages are the "Fortrans" of functional programming languages, since they are more powerful than Fortran. However, LISP has been around almost as long as Fortran and it and other lambda style languages tend to produce unstructured programs, as indicated above. Therefore perhaps it is time to begin designing a new generation of functional languages, languages that emphasize function level structure and function level reasoning, languages whose programs and program-forming operations comprise a space of mathematical objects.

Acknowledgments

I am grateful to Gordon Plotkin for his helpful discussion of category theory and relationships between certain aspects of it and some laws concerning PFOs of FP. I am also grateful to John H. Williams for many discussions of some of the questions discussed in the paper.

References

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

Backus, J. (1981a) Is computer science based on the wrong fundamental concept of "program"? An extended concept. Proc. International Symposium on Algorithmic Languages, IFIP TC2, Amsterdam, (October) (to appear).

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

Cadiou, J. M. (1972) Recursive definitions of partial functions and their computations. Report CS-266 Computer Science Dept., Stanford Univ., Stanford.

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

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

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

Mac Lane, S. (1971) Categories for the working mathematician. Springer-Verlag, New York.

Manna, Z., Ness, S., and Vuillemin, J. (1973) Inductive methods for proving properties of programs. CACM 16, 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. 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.