THE ALGEBRA OF FUNCTIONAL PROGRAMS: FUNCTION LEVEL REASONING, LINEAR EQUATIONS, AND EXTENDED DEFINITIONS

John Backus

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

Introduction

In [Back78] we introduced a functional style of programming in which variable-free programs are built from a set of primitive programs by a small set of combining forms or functionals and by recursive definitions. The programs of this "FP" style are all strict functions (of a single argument) that map a set of "objects" into itself. This set of objects is the closure under cartesian products of some given set of atoms. The earlier paper suggested that FP systems have the following properties:

(a) extremely simple semantics,

(b) existence of closed-form, non-recursive expressions for many functions that are normally recursively defined,

(c) programs exhibit a clear hierarchical structure in which high level programs can be combined to form higher level programs,

(d) the principal FP combining forms are the operations of a powerful algebra of programs that can be used to transform programs and to solve equations for recursively defined programs.

The present paper is concerned with three topics about FP systems, the first of which relates to (c) above and the other two relate to (d). After a brief review of the basics of FP systems and the algebra of programs, the first main section discusses the relationship between the FP style and the "lambda style" of programming and of reasoning about programs, where the lambda style is that used in lambda-calculus-based languages such as LISP and ISWIM [Land64].

Lambda abstraction -- the principal program-forming operation of the lambda style -- leads to an emphasis on objects and on object level reasoning, whereas the program-forming operations of the FP style emphasize programs (=functions), program structure, and function level reasoning. The lambda style thus obscures the function level structure of programs in its concern with objects and object level operations, whereas the FP style emphasizes the function level structure of programs.

The second main section of the paper generalizes and greatly simplifies the work in [Back78] on solving equations for recursively defined functions. It introduces the notion of "forms", the program schemas of FP, and of "linear" forms. It shows why the interaction of linear forms with the combining form "condition" makes possible a general solution for all linear equations of the form

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

where p and q are arbitrary functions and H is any linear form. A few theorems give a partial characterization of linear forms in terms of their structure. These show the existence of large classes of linear forms and of methods for building larger linear forms from simpler ones. They extend and modify results of an earlier draft paper [Back79]. The main result is the linear expansion theorem that gives the solution of (1) as an infinite condition

(2) $f = p \to q; \ldots; H_t^n p \to H^n q; \ldots$,

where H_t is the "predicate transformer" of H, a form associated with every linear form H. The solution gives necessary and sufficient conditions for termination and is useful in reasoning about the function f.

The third main section proposes a new form for defining FP functions that corrects a major problem in the proper definitions allowed in [Back78]: the difficulty of reading variable-free function definitions. Here we propose a style of function definition employing <u>function</u> variables that is as readable as the usual object level definitions of functions that use <u>object</u> variables to denote the argument or parts of the argument. By introducing the notions of "distributive" and "invertible" forms, we carefully define "extended definitions". Given an extended definition, we show how to make a corresponding variable-free proper definition and prove that both define the same function. Furthermore, we prove that the extended definition serves as a useful algebraic law about the defined function, one that is satisfied by all values of the function variables of the definition.

The sections that follow this introduction are the following:

(a) Review of FP functional programming.
(b) Review of the algebra of FP programs.
(c) A comparison of two styles of functional programming, the FP style

and the lambda style; function level versus object level structure and reasoning.

(d) Forms.

(e) Linear forms and the solution of linear functional equations.

(f) Extended definitions.

(g) Related work.

(h) Acknowledgments.

(i) References.

Review of FP functional programming

For a fuller description than this brief summary see [Back78]. An FP program is a function from objects to objects. Objects are (a) $\perp$, "bottom" or "undefined", or (b) atoms, or (c) sequences built from atoms or other sequences (but not from $\perp$); ϕ denotes the empty sequence; unlike [Back78], here ϕ is a sequence but not an atom. 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 (FP programs can choose from only a small set of such functional forms for building programs), or (c) it is defined, as in

Def $f = E(f,g_1,\ldots,g_n)$

where E is a function expression (built from functional forms) involving the parameters $g_1 \ldots g_n$ and possibly the defined function symbol f. The effect of this (proper) definition (we shall discuss "extended" definitions later in this paper) is that when the application f:x (denoting the result of applying the function f to the object x) is to be evaluated, the function symbol f is first replaced by its definition, $E(f,g_1 \ldots g_n)$. In the following examples we mark with an asterisk those items that depart from or add to definitions in [Back78].

Examples of objects

atoms 1.4 A ACD 23 X2 T (true) F (false)

sequences A <A,<B,C,D>,<B,C>> ϕ (the empty sequence)

(<A,$\perp$>=$\perp$ is not a sequence)

Examples of primitive functions

(All FP functions are strict, $f:\perp = \perp$, for all functions f. In describing primitives we shall assume this to be so and that primitives yield $\perp$ for unreasonable arguments.)

Selector functions 1:<A,B,C> = A 2:<A,B,C> = B similarly, any integer i denotes a selector function that selects the ith element

of its argument if there is one, otherwise it produces $\perp$.

tl (tail) $\quad tl:\emptyset = \perp \quad tl:\langle A\rangle = \emptyset \quad tl:\langle A,B,C\rangle = \langle B,C\rangle$

arithmetic operators (+,-,etc) $\quad +:\langle 4,5\rangle = 9 \quad \times:\langle 4,5\rangle = 20$

null (equals $\emptyset$) $\quad null:\emptyset = T \quad null:\langle A\rangle = F$

(*) $\subset$ (append left) $\quad \subset:\langle A,\emptyset\rangle = \langle A\rangle \quad \subset:\langle A,\langle B,C\rangle\rangle = \langle A,B,C\rangle$

(*) $\supset$ (append right) $\quad \supset:\langle\emptyset,A\rangle = \langle A\rangle \quad \supset:\langle\langle A,B\rangle,C\rangle = \langle A,B,C\rangle$

eq (equals) $\quad eq:\langle A,B\rangle = F \quad eq:\langle\langle C,D\rangle,\langle C,D\rangle\rangle = T$

id (identity function) $\quad id:x = x$ for all objects x.

and (logical and) $\quad and:\langle T,F\rangle = F \quad and:\langle T,T\rangle = T$

distl (distribute from left) $\quad distl:\langle A,\langle B,C,D\rangle\rangle = \langle\langle A,B\rangle,\langle A,C\rangle,\langle A,D\rangle$

distr (distribute from right) $\quad distr:\langle\langle A,B\rangle,C\rangle = \langle\langle A,C\rangle,\langle B,C\rangle\rangle$

ad (add 1) $\quad ad:3 = 4$

sb (subtract 1) $\quad sb:9 = 8$

Examples of functional forms (the examples show how one applies a function (built by a functional form from its parameters) by the use of those parameters.)

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

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

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

condition $\quad (p\rightarrow f;g) \quad$ 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 $\quad \bar{x} \quad$ where

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

(note that constant has an object as its parameter, in

particular $\bar{I}$ is the function that yields $\perp$ everywhere)

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

(*) right insert $/f$ where $/f:\langle x\rangle = x$
$/f:\langle x_1,x_2,\ldots,x_n\rangle = f:\langle x_1,/f:\langle x_2,\ldots,x_n\rangle\rangle$
(If f has a right unit u we extend this definition: $/f:\emptyset = u$.)

(*) left insert $\backslash f$ where $\backslash f:\langle x\rangle = x$
$\backslash f:\langle x_1,\ldots,x_n\rangle = f:\langle \backslash f:\langle x_1,\ldots,x_{n-1}\rangle,x_n\rangle$
(If f has a left unit u, we extend this definition: $\backslash f:\emptyset = u$.)

Examples of FP programs

$[1,tl,2]:\langle A,B,C\rangle = \langle A,\langle B,C\rangle,B\rangle$

$(null\circ tl \to 1;tl):\langle A\rangle = \perp$ $(null\circ tl \to 1;tl):\langle A,B\rangle = \langle B\rangle$

Def $last = null\circ tl \to 1;\ last\circ tl$

$last:\langle A,B\rangle = (null\circ tl \to 1;\ last\circ tl):\langle A,B\rangle$
$= last\circ tl:\langle A,B\rangle$ since $null\circ tl:\langle A,B\rangle = F$
$= last:(tl:\langle A,B\rangle) = last:\langle B\rangle$
$= (null\circ tl \to 1;\ last\circ tl):\langle B\rangle$
$= 1:\langle B\rangle$ since $null\circ tl:\langle B\rangle = T$
$= B$

$/c\circ\supset:\langle\langle A,B\rangle,\langle C,D\rangle\rangle = /c:(\supset:\langle\langle A,B\rangle,\langle C,D\rangle\rangle) = /c:\langle A,B,\langle C,D\rangle\rangle$
$= c:\langle A,/c:\langle B,\langle C,D\rangle\rangle\rangle$
$= c:\langle A,c:\langle B,\langle C,D\rangle\rangle\rangle$
$= c:\langle A,\langle B,C,B\rangle\rangle$
$= \langle A,B,C,D\rangle$ (thus $/c\circ\supset$ is concatenate)

Semantics of FP programs

Understanding the informal semantics of FP programs requires that one can determine the value of f:x for any FP function f and any object x. There are four possibilities for f: (1) it is a primitive function, (2) it is a functional form, (3) it is defined, or (4) none of these. If it is a primitive, one knows how to find f:x from the description of the primitive. If it is a functional form, the description of that form tells how to find f:x in terms of computing g:x for the parameters g of the functional form. If f is defined, Def $f = E(f,g_1,\ldots,g_n)$,

then to compute f:x, one computes $E(f,g_1,\ldots,g_n):x$, which can be done by further use of these rules. If none of the above, $f:x = \perp$.

It is important to note that all FP functions and the "sequence constructor" are strict: $f:\perp = \perp$, and $\langle\ldots\perp\ldots\rangle = \perp$.

(*) One important issue about FP semantics that was not covered in [Back78] is the relationship between the operational semantics above of a recursive definition and the least fixed point of the right side of such a definition when viewed as a continuous functional. If all FP functions were not strict, the operational definition of a recursively defined function and the corresponding least fixed point would differ, since the former depends on an innermost computation rule and such rules are not normally "safe" rules for computing the least fixed point. However, in [Will80] Williams points out that when all functions are strict, then innermost rules are safe, hence the operational bottom-up FP evaluation rules compute the least fixed point. Williams observes that the systems studied by Manna et al. [MNV73] and by Cadiou [Cadi72] employ conditional as a function (which cannot be strict, hence in such systems innermost rules are not safe) whereas FP systems represent condition as a (non-strict) functional. For a fuller discussion of this issue see [Will80].

Review of the algebra of FP programs

The application of FP programs to objects comprises a calculus of functional programming. The corresponding algebra of FP programs has variables that range over FP programs and operations that are the 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, and h and all objects x:

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

i.e., the law asserts that $[f,g]\circ h$ denotes the same function as $[f\circ h, g\circ h]$, no matter what functions one chooses for f, g, and h. Note that when we write $f = g$, we mean that f:x and g:x are both defined and equal or that they are both undefined ($=\perp$).

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

(This proof appears in [Back78].)

In addition to laws that hold for all objects x, we shall want to express the fact that a law holds only in some restricted domain. For such laws we provide qualified equations. Thus

$p \twoheadrightarrow f = g$

holds iff for every object x, either $p:x = T$ and $f:x = g:x$, or $p:x \neq T$. Similarly, a qualified assertion

$p \twoheadrightarrow q$

holds iff for every object x, either $p:x = T = q:x$, or $p:x \neq T$. For example, to express the fact that

$1 \circ [f,g] = f$

holds only when g is defined, we write the qualified law

$\bar{T} \circ g \twoheadrightarrow 1 \circ [f,g] = f$

since the qualification $\bar{T} \circ g:x = T$ whenver $g:x \neq \perp$.

The basic rules of the FP algebra are given by its laws. Twenty-four such laws are given in [Back78]. The reader can easily discover others. Here is a sample of such laws, sometimes in a bit less general form than given there, when the generalization is clear. We use the same numbering as [Back78] and give one law that does not appear there.

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 $\bar{T} \circ g \twoheadrightarrow 1 \circ [f,g] = f$

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

(*) V.1 $p \rightarrow (q \rightarrow r; s); (t \rightarrow u; v) = (p \rightarrow q; t) \rightarrow (p \rightarrow r; u); (p \rightarrow s; v)$

This concludes our brief review of the FP algebra of programs. Later sections of this paper will contain examples of the use of some of

these laws to prove various theorems.

A comparison of two styles of functional programming, the FP style and the lambda style; function level versus object level structure and reasoning

In the foregoing we have given a brief review of the basics of FP programming, one of two basically different styles of functional programming. The other style is associated with languages like LISP and ISWIM [Land64] and is based on the lambda calculus [Chur41]. LISP is a language almost as ancient as Fortran; however it is primarily a functional language and one of the most powerful available today. Most of its users, when they consider the FP style at all, see the latter as simply a weak, restrictive variant of the general functional style represented by "lambda style" or lambda-calculus-based languages such as LISP.

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 structures, 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 → q; h∘[r,s]

employs the PFO, or functional form, condition to build f from three

programs, p, q, and h∘[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) $f = \lambda x.(p:x \rightarrow q:x;\ h(r:x,s:x))$.

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. By writing "λx." 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 efficient.

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 [Will80] proves the following theorem for all $n\geq 1$ and for all functions f, a, b, and c:

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

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. 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'.

$$f{:}x \equiv 4x + g(3x) \ , \quad \text{where} \quad g{:}x \equiv even(x) \to x;\ 2x \ .$$
$$f'{:}x \equiv even(3x) \to 7x;\ 10x \ .$$

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

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

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 and use the properties of such forms to obtain the desired result (see below) 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.

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 $f = h \circ [r,s]$ 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. Furthermore, object level definitions of functions using variables are often more readable than variable-free definitions in the FP style (however, we have begun to study function level definitions -- using function variables instead of object variables -- that are comparably readable; see the following section on "extended definitions".)

The PFO of lambda abstraction 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 that would be considered highly "unstructured". 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.

Forms

Introduction

In the following we shall want to make general statements about large classes of function expressions. For example, we would like to say that Hfg is any function expression involving two functions f and g that has the property that

$(Hfg)\circ h = H(f\circ h)(g\circ h)$

no matter what functions f, g and h one chooses. To make such general statements we want to build expressions like Hfg using variable symbols f and g instead of particular functions f and g. Having built such a form Hfg we can then easily discuss the large class of functions Hfg that result when the variables f and g are replaced by any particular functions f and g.

Thus forms are program schemas for FP functional programs and comprise the means for expressing the most general questions we shall study. We shall define the set of FP forms with respect to some set of function variables V, where we assume that the set of objects, the set of function symbols, and the set of function variables V are pairwise disjoint.

Definition. Let $V = \{f_1 \ldots f_n\}$ be a set of (function) variables. Then we shall say that E is a form in V, or alternatively, that $Ef_1 \ldots f_n$ is

a form, if exactly one of the following holds:

1) $Ef_1 \cdots f_n = r$ for some FP function r, or

2) $Ef_1 \cdots f_n = f_i$ for some variable f_i in V, or

3) There are forms $E_1 \cdots E_k$ in V and an FP functional form Γ with k parameters such that

$$Ef_1 \cdots f_n = \Gamma(E_1, \ldots, E_k) \ .$$

(In the present paper we assume that Γ is one of the following: composition, construction, condition, left or right insert, apply to all.)

If $Ef_1 \cdots f_n$ is a form, then we write $Ef_1 \cdots f_n$ to denote the function obtained by replacing each variable f_i by the function f_i. We may also write $EE_1 \cdots E_n$ to denote the form that results from replacing each variable f_i by the form E_i.

Example

Let $Efg = f \circ [r, g \circ h]$. Then Epq is the function $p \circ [r, q \circ h]$ and $E(a \to b; c)d$ is the function $(a \to b; c) \circ [r, d \circ h]$ and $E[f,r]h$ is the form (in $\{f\}$) $[f,r] \circ [r, h \circ h]$. Note that Efg can be thought of as a form in $\{f,g\}$ or in $\{f,g,h\}$ or in any superset of $\{f,g\}$ of function variables. This ambiguity about the set of variables of a form will not cause difficulties in the following material. Of course, the set of variables of a form includes all the variables that actually occur in it.

Forms in a single variable, composition of forms

If Gf and Hf are forms in a single variable, then the composition of G and H, written GH, is simply $GHf = G(Hf)$, that is, the result of replacing f in G with the form Hf. For example, if

$Gf = f \circ r$

$Hf = p \to g; f$

then

$GHf = (p \to g; f) \circ r$

$HGf = p \to g; f \circ r$.

One must be careful not to confuse function composition with form composition, particularly when the former is used to build forms. Thus $GHf = (p \to g; f) \circ r$ is quite different from

$Gf \circ Hf = f \circ r \circ (p \to g; f)$.

For composition of a form G with itself, GGf, we shall write $G^2 f$ and $G^3 f$ for $GGGf$ and so on.

Equality and ordering of forms

If $Ef_1 \cdots f_n$ and $Ff_1 \cdots f_n$ are two forms, then we say that E = F,

that E and F are equal, iff

$$Ef_1 \ldots f_n = Ff_1 \ldots f_n$$

for all functions $f_1 \ldots f_n$. Analogously, $E \leq F$ iff

$$Ef_1 \ldots f_n \leq Ff_1 \ldots f_n$$

for all $f_1 \ldots f_n$. Thus our algebraic laws are merely statements of equality or ordering between forms.

Linear forms and the solution of linear functional equations

Introduction

In [Back78] we gave general solutions for two kinds of recursive equations for defining functions, the so-called "recursion" and "iteration" theorems. Here we give a simpler, more general description of of a much larger class of recursive equations -- the "linear" equations -- and provide a general solution for all of them. We believe the present treatment makes the basic issues of solving recursive equations much clearer than the earlier treatment.

As in the earlier solutions in [Back78], the solution for all linear equations presented here gives both necessary and sufficient conditions for the termination of the solution function and a case-by-case description of its behavior. Thus the solution is often useful for reasoning about or simplifying the defined function.

In the following subsections we introduce the notion of linear forms in a single variable. (This notion is closely related to that of "linear program schemas" (see the section "Related work") although the former is defined mathematically and the latter is defined syntactically.) We show that the basic FP functional forms, composition, construction, and condition, are each linear (in every variable position) and that if G and H are linear forms, then so is their composition GH, and therefore that the class of linear forms is large. We also give two theorems that partially characterize linear forms by their structure and enlarge the class of known linear forms to those containing multiple occurrences of the single variable.

We then give the main result, the linear expansion theorem, that gives a solution for all linear equations, equations of the form

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

where p and q are arbitrary functions and H is any linear form. The solution of (A) takes the form of an infinite condition:

(B) $f = p \to q; \ldots; H^n_t p \to H^n q; \ldots$,

where $H_t f$ is a form associated with any linear form Hf called its predicate transformer. (Recall that $H^{n+1}q = H(H^n q)$.)

In [Will80] Williams extends this result to two significant classes of non-linear equations like (A) in which H is either "overrun-tolerant" or "p-q-distributive". He shows that both classes have solutions very similar to (B).

A weaker form of the linear expansion theorem appeared in [Back79]. There the definition of linearity had slightly weaker requirements than the present definition, and the linear expansion theorem there required the hypothesis that $H\bar{1} = \bar{1}$ in addition to the linearity of H. I owe to Michael Arnold [Arno80] much of my present understanding of the behavior of the predicate transformer H_t of H in the domain for which $H\bar{1}$ is defined.

Linear forms

We wish to specify conditions under which we can formally solve recursive equations of the form

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

If we let $Ef = p \to q;\ Hf$, the usual approach to solving (1) [Klee52] is to take $\lim E^n\bar{1}$ as the least solution. We shall find that it is easier (and acceptable) to use $\lim E^n(p \to q;\bar{1})$ as the solution. Thus we want to find the following approximating functions.

$f_0 = p \to q;\bar{1}$

$f_1 = Ef_0 = p \to q;H(p \to q;\bar{1})$

$f_2 = p \to q;H(p \to q;H(p \to q;\bar{1}))$

and so on. If we can express $H(p \to q;\bar{1})$ in no other way, then finding a solution for (1) will be difficult. If, for any p_i, q and r, there is some p_{i+1} such that

(2) $\quad H(p_i \to q;r) = p_{i+1} \to Hq;\ Hr$

then, using (2) twice and taking $p = p_0$, we obtain

$f_2 = p \to q;\ p_1 \to Hq;\ p_2 \to H^2q;\ H^2\bar{1}$.

and in general,

(3) $\quad f_n = p \to q;\ p_1 \to Hq;\ldots;\ p_n \to H^nq;\ H^n\bar{1}$.

This suggests that

(4) $f = p \to q; \ldots; p_n \to H^n q; \ldots$

might be a solution of (1), where the infinite conditional in (4) represents

$\lim_{n\to\infty}(p\to q;\ldots;p_n\to H^n q;\bar{I})$.

However, (4) will represent the limit of the f_n of (3) only if

(5) For every object x there is an n such that, if $p_i{:}x = F$ for all $i \le n$, then $H^n\bar{I}{:}x = \perp$.

Otherwise there will be an x such that $f_n{:}x$ will always be given by the last term in (3), and hence (4) will not converge to the limit of the f_n in (3) at x. To see how this works, consider the following example. Let $Hf = r$ for some fixed function r. Then taking $p_{i+1} = \bar{F}$ for any p_i satisfies (2) and we obtain the following instance of (3)

$$\begin{aligned} f_n &= p \to q;\ \bar{F} \to Hq;\ldots;\ \bar{F} \to H^n q;\ H^n\bar{I} \\ &= p \to q;\ \bar{F} \to r;\ldots;\ \bar{F} \to r;\ r \\ &= p \to q;\ r \end{aligned}$$

Thus $\lim f_n = p\to q;r$, whereas (4) would give

$$\begin{aligned} f &= p \to q;\ \bar{F} \to r;\ldots;\ \bar{F} \to r;\ldots \\ &= p \to q;\ \bar{I}. \end{aligned}$$

The above discussion motivates the following definition of linear forms.

Definition (linear forms)

A form Hf is linear if and only if there is a form $H_t f$, called the predicate transformer of H, such that the following two conditions are satisfied.

(L1) For all functions a, b, and c,
$H(a\to b;c) = H_t a \to Hb;\ Hc$.

(L2) For all objects x, if $H\bar{I}{:}x \neq \perp$, then, for all functions a,
$H_t a{:}x = T$.

The purpose of H_t is to map p_i into p_{i+1} in (2); thus (L1) corresponds

to (2). The purpose of (L2) is to assure that (5) holds.

Examples of basic linear forms

We shall now show that basic forms Hf, built with one principal FP functional form (composition, construction, condition) and having zero or one occurrence of the variable f, are all linear.

(a) $Hf = r$. Then, with $H_t f = \bar{T}$:
(L1) $H(a \to b;c) = r = \bar{T} \to r;r = H_t a \to Hb;\ Hc$.
(L2) If $H\bar{I}:x \neq \perp$, then $x \neq \perp$, (since r is strict) and $H_t a:x = \bar{T}:x = T$.

(b) $Hf = r \circ f$. Then, with $H_t f = f = ID f$,
(L1) $H(a \to b;c) = r \circ (a \to b;c) = a \to r \circ b;\ r \circ c = H_t a \to Hb;\ Hc$.
(L2) $H\bar{I}:x = \perp$ for all x, since r is strict.

(c) $Hf = f \circ r$. Then, with $H_t f = Hf$,
(L1) $H(a \to b;c) = (a \to b;c) \circ r = a \circ r \to b \circ r;\ c \circ r = H_t a \to Hb;\ Hc$.
(L2) $H\bar{I}:x = \perp$ for all x.

(d) $Hf = [g,f]$. Then, with $H_t f = f = ID f$,
(L1) $H(a \to b;c) = [g,(a \to b;c)] = a \to [g,b];\ [g,c] = H_t a \to Hb;\ Hc$.
(L2) $H\bar{I}:x = \perp$ for all x.
Similarly, $[\ldots f \ldots]$ is linear with $H_t = ID$.

(e) $Hf = p \to q;\ f$. Then, with $H_t f = p \to \bar{T};f$,
(L1) $H(a \to b;c) = p \to q;a \to b;c = (p \to T;a) \to (p \to q;b);\ (p \to q;c) = H_t a \to Hb;\ Hc$.
(L2) If $H\bar{I}:x \neq \perp$, then $p:x = T$ and $x \neq \perp$, hence $H_t a:x = (p \to \bar{T};a):x = T$.
Similarly, $p \to f;q$ is linear with $H_t f = p \to f;\bar{T}$. Also, $f \to g;h$ is linear with $H_t f = ID f$.

Thus all three basic FP functional forms with one occurrence of a single variable are linear in every variable position.

Composition of linear forms

We now show that if G and H are linear, then their composition (form composition) GH is also linear and $(GH)_t = G_t H_t$. (I owe the following lemma and the proof of the composition theorem to John H. Williams.) We shall need the following lemma.

Lemma If H is linear, then, for any function p,
$H\bar{I} = H_t p \to H\bar{I};\bar{I}$.

Proof Let p be given. Let x be given. We show that the two sides of the above equation give the same result at x. We consider two cases.
Case 1 $H\bar{I}:x \neq \perp$. Then, by (L2) since H is linear, $H_tp:x = T$. Therefore $(H_tp \to H\bar{I};\bar{I}):x = H\bar{I}:x$ and our equation holds for x.
Case 2 $H\bar{I}:x = \perp$. There are three possibilities for the value of $H_tp:x$; either it is T or it is F or it is neither of these. If it is T, then $(H_tp \to H\bar{I};\bar{I}):x = H\bar{I}:x$; if it is F, then $(H_tp \to H\bar{I};\bar{I}):x = \perp = H\bar{I}:x$; finally, if it is neither T nor F, then $(H_tp \to H\bar{I};\bar{I}):x = \perp = H\bar{I}:x$. Thus in all three cases our equation holds for x.

Q.E.D.

We can now prove the

COMPOSITION THEOREM If G and H are linear forms with predicate transformers G_t and H_t, then GH is linear with the predicate transformer $(GH)_t = G_tH_t$.

Proof First we show that GH and G_tH_t satisfy (L1). Let a, b, and c be given. Then
(L1) $GH(a \to b;c) = G(H_ta \to Hb;Hc) = G_tH_ta \to GHb;\ GHc$
by (L1) used once for H, then once for G. Thus (L1) is satisfied.
To show that (L2) is satisfied we must show that if $GH\bar{I}:x \neq \perp$, then $G_tH_ta:x = T$ for any function a. Let x be given such that $GH\bar{I}:x \neq \perp$. (If no such x exists, then (L2) is vacuously satisfied.) Let some function a be given. We consider two cases.
Case 1 $G\bar{I}:x \neq \perp$. Then by (L2), $G_tp:x = T$ for any function p; hence $G_tH_ta:x = T$ as required.
Case 2 $G\bar{I}:x = \perp$. By the preceding lemma

$$H\bar{I} = H_ta \to H\bar{I};\ \bar{I} .$$

Taking G of both sides gives

$$GH\bar{I} = G_tH_ta \to GH\bar{I};\ G\bar{I}$$

by (L1) since G is linear. Now by our choice of x, $GH\bar{I}:x \neq \perp$ and, in this case 2, $G\bar{I}:x = \perp$. Therefore, if the above equation is to hold, as it must, we must have $G_tH_ta:x = T$, as required.

Q.E.D.

Examples of composite linear forms

Using the composition theorem and the linearity of basic forms with a single occurrence of the variable, we see that all forms Hf built with composition, construction and condition, and having a single occurrence of f are linear. For example, if

$$Hf = f \circ j \quad Gf = [i,f] \quad Ff = h \circ f ,$$

then

$$FGHf = h \circ [i, f \circ j]$$

and FGH is linear, since it is the composition of linear forms. Furthermore $(FGH)_t f = f \circ j = ID(ID(H_t f))$ since $G_t = F_t = ID$. Similarly if

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

then

$$IHf = p \rightarrow q;\ f \circ j$$

is linear and $(IH)_t f = p \rightarrow \bar{T};\ f \circ j = I_t H_t f$.

Partial characterization of linear forms

From the above we know that there are many linear forms Hf in which f occurs once. Unfortunately, we have no test as yet (even if one exists) to determine whether a form Hf in which there are multiple occurrences of f is linear (many are). Such a test must probably await a classification of forms yet to be developed. However, we can give necessary and sufficient conditions for H to be linear when H has the form

$$Hf = p \rightarrow Ef;\ Gf \ .$$

(It is sufficient for H to be linear to have E and G linear, and a bit less will do.) We can also give a sufficient condition for H to be linear when

$$Hf = [Ef, Gf] \ .$$

(It requires that E and G be linear and have predicate transformers that "differ only by a constant".) Although this condition is not known to be necessary, it appears to be "close": we can prove that if H has the above form and is linear, then the following must hold for all a, b, and c,

$$[E(a \rightarrow b; c), G(a \rightarrow b; c)] = [(H_t a \rightarrow Eb;\ Ec), (H_t a \rightarrow Gb;\ Gc)] \ .$$

Unfortunately, this does not mean that E and G satisfy (L1) individually and in fact, as the sufficient condition shows, E and G need not have identical predicate transformers as this result seems to indicate they should, even if they are linear.

For the other ways of building a form H, we have no useful results at this time. Thus we have no linearity test in terms of the components of H when H has one of the following forms:

$$Hf = Ef \circ Gf \ ,$$

$Hf = Ef \to Ff; Gf$

$Hf = /Ef$

$Hf = \alpha(Ef)$.

In the important case $Hf = Ef \circ Gf$, H is not easily made linear unless one of the forms is linear and the other does not depend on f.

In [Back79] another approach to studying linearity of forms with multiple occurrences of the variable was begun. It involved the study of forms in two or more variables, say Hfg, that are linear in each variable independently and then asking, when is $Gf = Hff$ linear?

To summarize our present state of knowledge about which forms Hf are linear: if H is built only by composition, construction, and condition, and has only one occurrence of f, it is linear. If $Hf = p \to Ef; Gf$, then H is linear if and only if E is linear when p is true and G is linear when p is false. If $Hf = [Ef, Gf]$, E and G are of the form $p \to r; E'f$ and $q \to s; G'f$ respectively and E' and G' are linear and $E'_t = G'_t$, then H is linear (a simpler corollary: if E and G are linear and $E_t = G_t$, then H is linear). Below we give the theorems that support these results (beyond those given earlier). We begin with the case

$Hf = p \to Ef; Gf$.

We shall need the following definition.

Definition Hf is p-linear (linear in the domain in which p is true) iff there is an $H_t f$ such that

(L1') For all a, b, c,

$$p \to\to H(a \to b; c) = H_t a \to Hb; Hc .$$

(L2') For all objects x, if $p{:}x = T$ and $H\bar{1}{:}x \neq \perp$, then $H_t a{:}x = T$ for all functions a.

THEOREM 1 (conditional linear forms) If $Hf = p \to Ef; Gf$, then Hf is linear if and only if E is p-linear and G is $(not \circ p)$-linear; and

$H_t f = p \to E_t f; G_t f$.

Proof If. Let E be p-linear and G be $(not \circ p)$-linear. Then

$$H(a \to b; c) = p \to\cdot E(a \to b; c); G(a \to b; c)$$
$$= p \to (E_t a \to Eb; Ec); (G_t a \to Gb; Gc) .$$

Now the following holds for all p, q, r, s, t, u, v, (law V.1):

$$p \to (q \to r; s); (t \to u; v) = (p \to q; t) \to (p \to r; u); (p \to s; v) .$$

Applying this law to the above gives:

$$H(a \to b; c) = (p \to E_t a; G_t a) \to (p \to Eb; Gb); (p \to Ec; Gc)$$

$= (p \to E_t a; G_t a) \to Hb; Hc$.

Thus H satisfies (L1) with $H_t f = p \to E_t f; G_t f$. Now suppose that $H\bar{I}:x \neq \perp$. There are two cases, $p:x = T$ or $p:x = F$; otherwise $H\bar{I}:x = \perp$. If $p:x = T$, then $E\bar{I}:x \neq \perp$, hence $E_t a:x = T$ for all a. Therefore $H_t a:x = T$ for all a. Similarly, if $p:x = F$, then $G_t a:x = T$ and $H_t a:x = T$ for all a and (L2) is satisfied. Thus H is linear with the specified H_t.

Only if. Let H be linear with H_t. Then

(a) $H(a \to b;c) = H_t a \to Hb;Hc$

$= H_t a \to (p \to Eb;Gb);(p \to Ec;Gc)$ by definition of H.

By using law V.1 cited above, we obtain

(b) $H(a \to b;c) = (H_t a \to p;p) \to (H_t a \to Eb;Ec);(H_t a \to Gb;Gc)$

$= p \to (H_t a \to Eb;Ec);(H_t a \to Gb;Gc)$.

(The last line follows from the fact that when $H_t a$ is undefined, both lines give undefined.) From the definition of H we have

(c) $H(a \to b;c) = p \to E(a \to b;c); G(a \to b;c)$.

From (b) and (c) ,

(d) $p \to\to E(a \to b;c) = H_t a \to Eb; Ec$

and

(e) $not \circ p \to\to G(a \to b;c) = H_t a \to Gb; Gc$.

Thus E and G satisfy (L1') for p-linearity and for (not∘p)-linearity with $E_t = G_t = H_t$. Now let x be such that $p:x = T$ and $E\bar{I}:x \neq \perp$. Then, by definition of H, $H\bar{I}:x \neq \perp$; therefore $H_t a:x = E_t a:x = T$ for all functions a. Thus E satisfies (L2') for p-linearity with $E_t = H_t$. Similarly, G satisfies (L2') for (not∘p)-linearity. The required form, $H_t a = p \to E_t a;G_t a = p \to H_t a;H_t a$ is consistent with the above.

Q.E.D.

We shall need the following lemma for our next theorem.

Lemma For all a,b,c,d,e,p,q,r,s, the following holds

$$[(p \to r;a \to b;c),(q \to s;a \to d;e)] = Ppqa \to [(p \to r;b),(q \to s;d)];[(p \to r;c),(q \to s;e]].$$

where $Ppqa = p\&q \to \bar{T}; a$, and $p\&q = and \circ [p,q]$.

Proof outline We shall use laws IV.1 (in the general form given in [Back78]) and V.1 and the fact that $q = \bar{T} \to q;q$. We proceed to transform the left side of the equation by these laws until it becomes the right side.

$= p \to [r,(q \to s;a \to d;e)];[(a \to b;c),(q \to s;a \to d;e)]$

$= p \to (q \to [r,s]; [r,(a \to d; e)]); (q \to [(a \to b; c), s]; [(a \to b; c), (a \to d; e)])$

$= p \to (q \to [r,s]; (a \to [r,d]; [r,e])); (q \to (a \to [b,s]; [c,s]); (a \to [b,d]; [c,e]))$

Letting $[r,s] = \bar{T} \to [r,s]; [r,s]$ and applying V.1,

$= p \to ((q \to \bar{T}; a) \to (q \to [r,s]; [r,d]); (q \to [r,s]; [r,e]));$
$((q \to a; a) \to (q \to [b,s]; [b,d]); (q \to [c,s]: [c,e]))$

$= (p \to (q \to \bar{T}; a); (q \to a; a)) \to (p \to (q \to [r,s]; [r,d]); (q \to [b,s]; [b,d])));$
$(p \to (q \to [r,s]; [r,e]); (q \to [c,s]; [c,e])))$

$= (p \& q \to \bar{T}; a) \to (p \to [r, (q \to s; d)]; [b, (q \to s; d)]);$
$(p \to [r, (q \to s; e)]; [c, (q \to s; e)])$

$= Ppqa \to [(p \to r; b), (q \to s; d)]; [(p \to r; c), (q \to s; e)]$.

Q.E.D.

THEOREM 2 (constructed linear forms) Given

(a) $Hf = [E'f, G'f]$

(b) $E'f = p \to r;\ Ef$

(c) $G'f = q \to s;\ Gf$

(d) E and G are linear with E_t and G_t

(e) $E_t = G_t$

Then Hf is linear with $H_t f = p \& q \to \bar{T};\ E_t f$.

Proof By the above hypotheses we have

$H(a \to b; c) = [(p \to r; E_t a \to Eb; Ec), (q \to s; G_t a \to Gb; Gc)]$.

Using the above lemma with the following instantiations (theorem element → lemma variable):
$(p \to p)(r \to r)(E_t a \to a)(Eb \to b)(Ec \to c)(q \to q)(s \to s)(G_t a \to a)$ [since $E_t = G_t$] $(Gb \to d)$ and $(Gc \to e)$, we get

$H(a \to b; c) = Ppqa \to [(p \to r; Eb), (q \to s; Gb)]; [(p \to r; Ec), (q \to s; Gc)]$
$= (p \& q \to \bar{T};\ E_t a) \to [E'b, G'b];\ [E'c, G'c]$
$= H_t a \to Hb;\ Hc$.

Thus H and H_t satisfy (L1). Now suppose $H\bar{I}:x \neq \perp$. Then by (a) $E'\bar{I}:x \neq \perp$ and $G'\bar{I}:x \neq \perp$. Hence by (b) and (c) p and q must have boolean values at x. Now E' is a linear form whose predicate transformer is

$E'_t f = p \to \bar{T}; E_t f$,

since $E'f$ is the composition of the linear form $p \to r; f$ and the linear form Ef (E is linear by (d)). Therefore, since $E'\bar{I}:x \neq \perp$, then by (L2), $E'_t a:x = T$; thus either $p:x = T$ or $E_t a:x = T$. Similarly, either $q:x = T$ or $G_t a:x = E_t a:x = T$. (Both of these hold for every a.) Thus if $p \& q:x = T$ then $H_t a:x = T$ for every a, and if $p \& q:x = F$ then $E_t a:x = T$ for every a. Thus $H_t a:x = T$ for every a and (L2) is satisfied.

Q.E.D.

COROLLARY If $Hf = [Ef, Gf]$ and E and G are linear and $E_t = G_t$, then H is linear with $H_t = E_t = G_t$.

Proof Let $p = q = \bar{F}$ in the above theorem. Q.E.D.

This concludes our partial characterization of linear forms.

The solution of linear functional equations

So far wa have defined linear forms and given some characterization of them. Now we come to the main business of showing that linear equations,

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

where H is linear, have a general solution of the form

(7) $f = p \to q; \ldots; H_t^n p \to H^n q; \ldots$,

where by the right side of (7) we mean $\lim f_n$ where

(8) $f_n = p \to q; \ldots; H_t^n p \to H^n q; \bar{\perp}$.

By the solution of (6) we mean the least fixed point of E where
$Ef = p \to q; Hf$.

Since the bottom-up operational semantics of FP definitions as given earlier is not, in general, a safe computation rule for computing the least fixed point of E (see [MNV73] and [Cadi72]), one might think the FP function computed by

Def $f = p \to q; Hf$

might be less defined than the least fixed point of E. However, when all the functions in E are strict, as they are in FP, then the bottom-up rule is a safe rule. Manna et al. and Cadiou do not consider this case because the programs they treat all require the non-strict function conditional, whereas FP functions are all strict but use the (non-strict) functional condition. This question of the correct correspondence of FP and least fixed point semantics was first raised and answered by Williams. For a fuller discussion of this point see [Will80].

In our proof of the linear expansion theorem we shall assume and use a number of standard notions and arguments about ordering of objects

and functions, about limits of increasing sequences of functions and about least fixed points of continuous functionals, etc. We shall follow the treatment of these questions given in [MNV73], which the reader may consult for details and references. To bring the FP material into that treatment we note that the basic ordering on FP objects x and y is

$x \leq y$ iff $x = \perp$ or $x = y$.

We shall treat an FP form Ef as a functional and assume without proof that all FP forms Ef are continuous functionals. Our treatment of standard issues will be rather limited.

As stated above, in order to solve (6) we want to find the least fixed point f of

(9) $Ef = p \rightarrow q;\ Hf$.

We know that $f = \lim E^n\bar{\perp}$, but it will be more convenient to start with $p\rightarrow q;\bar{\perp}$ rather than with $\perp$. That is, we want to show that f, the least fixed point of E, is the limit of the following f_n,

(10) $f_n = E^n(p\rightarrow q;\bar{\perp})$.

Now

(11) $\bar{\perp} \leq p\rightarrow q;\bar{\perp} \leq E\bar{\perp} = p\rightarrow q;H\bar{\perp}$.

Therefore

(12) $E^n\bar{\perp} \leq E^n(p\rightarrow q;\bar{\perp}) \leq E^{n+1}\bar{\perp}$.

And so we must have: $\lim E^n\bar{\perp} = \lim E^n(p\rightarrow q;\bar{\perp})$ = the solution of (6). With this fact in hand we can prove the following lemma and theorem.

Linear approximation lemma If H is linear and

(13) $Ef = p \rightarrow q;\ Hf$,

then, for all $n\geq 0$,

(14) $E^n(p\rightarrow q;\bar{\perp}) = p \rightarrow q;\ldots;\ H_t^n p \rightarrow H^n q;\ \bar{\perp}$.

Proof By induction on n. Now (14) holds for n=0; both sides reduce to $p\rightarrow q;\bar{\perp}$, since for any form F, $F^0 f = f$. We assume (14) holds for $n\geq 0$ and prove it holds for n+1.

$E^{n+1}(p\rightarrow q;\bar{\perp}) = E(E^n(p\rightarrow q;\bar{\perp}))$ by def of form composition

$= E(p\rightarrow q;\ldots;H_t^n p\rightarrow H^n q;\bar{\perp})$ by induction hypothesis

$= p\rightarrow q;H(p\rightarrow q;\ldots;H_t^n p\rightarrow H^n q;\bar{\perp})$ by (13)

$= p\rightarrow q;H_t p\rightarrow Hq;\ldots;H_t^{n+1} p\rightarrow H^{n+1} q;H\bar{\perp}$ by use of (L1) for H n times

$$= p \to q; H_t p \to Hq; \ldots; H_t^{n+1} p \to H^{n+1} q; \bar{I} \ .$$

The last line follows from its predecessor, since for any object x, if $H\bar{I}:x = \perp$, then replacing $H\bar{I}$ by $\bar{I}$ makes no change at x. On the other hand, if $H\bar{I}:x \neq \perp$, then by (L2) $H_t p:x = T$; thus in this case the value of $E^{n+1}(p \to q; \bar{I})$ at x is $Hq:x$ and it does not matter what function fills the last place in the expansion. Thus (14) holds for n+1 (for every x) when it holds for $n \geq 0$.

Q.E.D.

Now we can prove the

LINEAR EXPANSION THEOREM Given H linear. Then the least solution of

(15) $f = p \to q; Hf$

is

(16) $f = p \to q; \ldots; H_t^n p \to H^n q; \ldots$

Proof By our earlier discussion [(9)(10)(11)(12)] we know that the least solution of (15) is $f = \lim E^n(p \to q; \bar{I})$. From the linear approximation lemma we know that

$$E^n(p \to q; \bar{I}) = p \to q; \ldots; H_t^n p \to H^n q; \bar{I} \ .$$

And, by our definition of infinite conditions (8), the right side of (16) is the limit of $E^n(p \to q; \bar{I})$; thus it equals f, the least solution of (15), as required.

Q.E.D.

To see that the above treatment of linear forms and the linear expansion theorem replaces and generalizes the treatment of these matters in [Back78], it is only necessary to consider the "recursion theorem" of that paper. It states that the equation

$$f = p \to g; Qf \ , \quad \text{where } Qf = h\,[i, f \circ j] \ ,$$

has the solution

$$f = p \to g; \ldots; p \circ j^n \to Q^n g; \ldots \ .$$

Now our discussion of linear forms makes it clear that Q is linear, since it is the composition of basic linear forms, and that

$$Q_t f = f \circ j \ .$$

Therefore the "recursion theorem" is a very special case of the present linear expansion theorem for one particular linear form Q, since its conclusion is that of the linear expansion theorem for H = Q.

The earlier discussion of linear forms shows, for example, that we could apply the linear expansion theorem to solve (15) when H is the

form

$Hf = a \rightarrow f \circ b;\ h \circ [(e \rightarrow d; f \circ g), (i \rightarrow j; k \circ [m, f \circ g])]$

and

$H_t f = a \rightarrow f \circ b;\ e \& i \rightarrow \bar{T};\ f \circ g$,

although it is not likely that $H_t^n p$ and $H^n q$ are going to be tractable expressions unless the parameters of H conspire to make them so.

Extended definitions

Introduction

One of the principal difficulties with the FP style of programming is that function definitions without variables are hard to read. The problem is illustrated by the FP definition of the function f, whose object level definition is,

(1) $f{:}\langle x, \langle y, z\rangle\rangle = \langle\langle x, y\rangle, \langle x, z\rangle\rangle$.

The corresponding proper FP definition at the function level is:

(2) Def $f = [[1, 1 \circ 2], [1, 2 \circ 2]]$.

Not only is this definition less clear than (1), but also (2) really defines an extension of the function defined by (1). For example, (2) gives

$f{:}\langle a, \langle b, c, d\rangle, e\rangle = \langle\langle a, b\rangle, \langle a, c\rangle\rangle$,

whereas the intent of (1) is that f be undefined for this argument.

Thus proper FP definitions have two severe problems: (A) the compositions of selector functions that replace variables are difficult to read as compared to (1), and (B) the FP definition does not properly limit the domain of the defined function unless a cumbersome explicit condition clause is introduced.

We propose here a new style of FP function definition, called extended definitions that (a) provides the readability of (1) and (b) limits the domain of definition in the manner of (1), while at the same time the new style remains at the function level and provides an immediate algebraic law about the defined function. For example,

(3) xdef $f \circ [x, [y, z]] = [[x, y], [x, z]]$

is an extended definition that defines precisely the same function as (1).

That is, f is undefined unless its argument has the form indicated in (1).

The principal difference between (1) and (3) is that x, y, and z in (3) are function variables (whereas in (1) they are object variables) and (3), like (2), is a function level definition, but is as readable (with some practice) as (1).

The extended definition (3) holds for all functions x, y, and z; thus it serves not only as the definition of f, but also as a very useful algebraic law about f. In most FP programs g that use f, f will occur in a context such as

f∘[r,[s,t]]

where r, s, and t are particular functions used to construct the three parts of f's argument for this use within the program g. Thus in reasoning about g, one can use the extended definition of f as an algebraic law and replace the function f∘[r,[s,t]] by [[r,s],[r,t]] in the function g, just as one can use (1) as a law about objects that allows one to replace the object f:<x,<y,z>> by the object <<x,y>,<x,z>> . Of course, in reasoning about programs, it is more directly helpful to be able to replace one subprogram by another than it is to be able to replace one object by another.

Another important requirement that extended definitions must meet is that they can be easily converted into proper, variable-free FP definitions. We shall carefully define extended definitions and prove that the function they define is the same as the function defined by a corresponding proper definition, and that the function defined by the proper definition satisfies the extended definition equation for all values of the function variables of the extended definition. This last will guarantee that the extended definition serves as a universal algebraic law for the defined function.

Before looking at the general form of extended definitions, let us consider some further examples, along with the proper definitions of the functions they define.

xdef last∘c∘[x,y] = null∘y → x; last∘y

In this definition, last is defined only when y produces a sequence; when y does not, last will "see" $\perp$ and produce $\perp$; when y does, last will "see" a non-empty sequence whose first element is produced by x. Thus x "names" the function that will produce the first element of

the argument of last, and y "names" the function that will produce the tail of the argument. This extended definition is the equivalent of the following proper definition.

Def last = p → (null∘tl → 1; last∘tl);Ī

where p = eq∘[⊂∘[1,tl],id] is the predicate formally needed to assure that last will be defined only for non-empty sequences (in this instance it is redundant). One further example:

xdef f∘[x,x] = g∘x .

This defines a function on pairs whose elements are required to be equal. The corresponding proper definition of f is

Def f = eq∘[[1,1],id] → g∘1; Ī .

Observe that extended definitions can be changed into object level definitions by changing "∘" to ":" and square brackets to angle brackets. Thus the above example becomes

f:⟨x,x⟩ = g:x .

As the above examples indicate, the general form of an extended definition for a function symbol f is

(4) xdef f∘E$x_1 \ldots x_n$ = F f$x_1 \ldots x_n$,

where E and F are suitably restricted forms in $x_1 \ldots x_n$ (F may have an additional variable, which is replaced by f, when the definition is recursive). Our examples also indicate that we have two implicit assumptions about (4) that we can make explicit as follows:

Assumption 1 If, for some functions $x_1 \ldots x_n$ and some object y, $(Ex_1 \ldots x_n):y = \perp$ and $(Ffx_1 \ldots x_n):y \neq \perp$, then $(f \circ Ex_1 \ldots x_n):y = \perp$, that is, f is to be strict, even if (4) indicates otherwise.

Assumption 2 For any object z, if z cannot be produced by $(Ex_1 \ldots x_n):y = z$ for any functions $x_1 \ldots x_n$ and object y (thus f will never "see" such a z in (4)), then f:z = ⊥, that is, f is to be defined only for arguments that f can "see" in (4).

Now it is not certain that (4) with the above assumptions defines anything at all. However, we shall confirm that it does. First we shall define

a) distributive forms

b) invertible forms

c) range predicates p_E of a distributive and invertible form E.

Then we shall say that (4) is an extended definition if

a) $Ex_1 \ldots x_n$ is distributive and invertible, and

b) $Ffx_1 \ldots x_n$ is distributive in $x_1 \ldots x_n$,

and that (4) defines the least function f that (i) satisfies

(5) $f \circ Ef_1 \ldots f_n = \bar{T} \circ Ef_1 \ldots f_n \rightarrow Fff_1 \ldots f_n ; \bar{I}$

for all functions $f_1 \ldots f_n$ ((5) already incorporates assumption 1) and (ii) satisfies assumption 2.

Finally we shall prove (the extended definition theorem) that every function that satisfies (5) plus assumption 2 also satisfies the following equation (and vice versa).

(6) $f = p_E \rightarrow Ffs_1 \ldots s_n ; \bar{I}$

where p_E is the range predicate of E and $s_1 \ldots s_n$ are certain "inversion functions" associated with E.

Thus the extended definition theorem asserts that extended definitions (4) define the same function that is defined by the proper definition (6) that is directly obtained from the extended definition.

(The discussion of extended definitions in this paper covers some of the work on this subject that will appear in a paper being prepared by the present author and S. W. Smoliar. In the latter paper definitions will be further extended to allow the introduction of new function variables in the predicates on the right side of a definition, variables whose scope is the consequent of the predicate in which they appear.)

The following more formal treatment is accompanied by examples illustrating the definitions and issues involved.

Formal description and justification of extended definitions

Definitions and examples

(A) Distributive forms

If $Ex_1 \ldots x_n v_1 \ldots v_m$ is a form, then it is said to be distributive in (the variables) $v_1 \ldots v_m$ iff

$$(Ef_1 \ldots f_n g_1 \ldots g_m) \circ h = Ef_1 \ldots f_n (g_1 \circ h) \ldots (g_m \circ h)$$

holds for all functions $f_1 \ldots f_n$ assigned to the non-distributive variables $x_1 \ldots x_n$, all functions $g_1 \ldots g_m$ assigned to the distributive variables $v_1 \ldots v_m$, and all functions h. If $Ex_1 \ldots x_n$ is distributive in $x_1 \ldots x_n$, then we say that $Ex_1 \ldots x_n$ is simply distributive.

Examples of distributive forms

$Exy = r \circ [g \circ x, y]$

$Fxy = p \circ x \rightarrow h \circ x;\ g \circ [x, y]$

$Gxy = [\bar{z}, y]$ (for some object z)

The above are distributive forms (in both x and y).

$Exy = x \circ y$

$Fxy = x \circ [y, g \circ y]$

$Gxy = [x \circ \bar{z}, h \circ y]$

The above forms are distributive in y but not in x.

$Exy = p \rightarrow x; y$

$Fxy = [p, g \circ x, h \circ y]$

$Gx = \alpha x$

Here E and F are not distributive in either variable unless p is a constant-valued function; if it is, both are distributive. G is not distributive.

As the above examples show, a form $Ex_1 \ldots x_n v_1 \ldots v_m$ is distributive when the only variables or functions in E that "see" the argument of E directly are either distributive variables or constant-valued functions. Note that $Exy = x$ is distributive in y as well as in x. It is clear that if $Ex_1 \ldots x_n$ is distributive and if $E_1 \ldots E_n$ are each distributive in $v_1 \ldots v_m$, then $EE_1 \ldots E_n$ is a form that is distributive in $v_1 \ldots v_m$.

(B) Invertible forms

A form $Ex_1 \ldots x_n$ is invertible for x_i iff there is an inversion function s_i (for x_i in $Ex_1 \ldots x_n$) such that, for all functions $f_1 \ldots f_n$,

$\bar{T}\circ Ef_1\ldots f_n \twoheadrightarrow s_i\circ Ef_1\ldots f_n = f_i$.

(Recall that $\bar{T}$ is a strict function, thus the qualification is that $Ef_1\ldots f_n$ be defined.) If $Ex_1\ldots x_n$ is invertible for all variables $x_1\ldots x_n$ (by $s_1\ldots s_n$), we say that $Ex_1\ldots x_n$ is <u>invertible</u>.

<u>Examples of invertible forms</u>

$Exy = [x,y]$

E has inversion functions $s_1 = 1$ (for x) and $s_2 = 2$ (for y).

$Fxy = c\circ[x,y]$

has inversion functions $s_1 = 1$ and $s_2 = tl$.

$Gxyz = [x,c\circ[y,z]]$

has inversion functions $s_1 = 1$, $s_2 = 1\circ 2$, and $s_3 = tl\circ 2$.

$Hx = [x,x]$

has two distinct inversion functions for x: $s_1 = 1$ and $s_1' = 2$.

$Ixyz = distl\circ[x,[y,z]]$

has inversion functions $s_1 = 1\circ 1$ or $s_1' = 1\circ 2$, $s_2 = 2\circ 1$, and $s_3 = 2\circ 2$.

<u>(C) Range predicates of a form</u>

Let $Ex_1\ldots x_n$ be a distributive and invertible form with inversion functions $s_1\ldots s_n$. Then

$p_E = eq\circ[Es_1\ldots s_n, id]$

is a <u>range predicate</u> for the form $Ex_1\ldots x_n$ (which has the property that $p_E{:}x = T$ if there are $f_1\ldots f_n$ and object y such that $x=(Ef_1\ldots f_n){:}y$).

<u>Examples of range predicates</u>

$Exy = [x,y]$

has $p_E = eq\circ[[1,2],id]$. Note that $p_E{:}z$ is $\perp$ for z an atom or a sequence of length 1, $p_E{:}z$ is T for pairs and F for longer sequences.

$Fxy = [x,y,x]$

has two distinct range predicates since there are two inversion functions for x, $s_1 = 1$ and $s_1' = 3$. Thus we have

$p_F = eq\circ[[1,2,1],id]$ and $p_F' = eq\circ[[3,2,3],id]$.

Of course, by definition of range predicates, it is not possible to use different inversion functions for x in the different positions in which it occurs in F, thus

$eq\circ[[1,2,3],id]$

is not a range predicate for F.

Observe that $p_F{:}\langle u,v\rangle = F$, whereas $p_F'{:}\langle u,v\rangle = \perp$. Hence there may be

some ambiguity about the range predicate for a form. However, we shall use range predicates p_E only to restrict the domain of definition of a function, that is, only in the context

$$p_E \rightarrow f;\bar{I} \quad .$$

In this context there is no ambiguity. If p_E and p'_E are two range predicates for E, we prove that

$$p_E \rightarrow f;\bar{I} \;=\; p'_E \rightarrow f;\bar{I} \quad .$$

Since a form E has distinct range predicates only when it has distinct inversion functions for a variable, the following theorem suffices:

THEOREM Let $Ex_1 \ldots x_n$ be distributive and invertible by $s_1 \ldots s_n$. Let s_i and s'_i be inversion functions for x_i. Let $e = Es_1 \ldots s_i \ldots s_n$ and $e' = Es_1 \ldots s'_i \ldots s_n$. Let $p_E = eq \circ [e,id]$ and $p'_E = eq \circ [e',id]$. Then, for any f,

$$p_E \rightarrow f;\ \bar{I} = p'_E \rightarrow f;\ \bar{I} \quad .$$

proof It suffices to show that for any object y, $p_E:y=T$ iff $p'_E:y=T$. By symmetry, we need show this in only one direction. So we let $p_E:y=T$ and prove that $p'_E:y=T$. If $p_E:y=T$, then by definition of p_E, $e:y=y$ or

(a) $e \circ \bar{y} = \bar{y}$

By distributivity of E,

(b) $e \circ \bar{y} = Es_1 \circ \bar{y} \ldots s_i \circ \bar{y} \ldots s_n \circ \bar{y}$.

Since s'_i is an inversion function of E,

(c) $\bar{T} \circ e \circ \bar{y} \twoheadrightarrow s'_i \circ Es_1 \circ \bar{y} \ldots s_i \circ \bar{y} \ldots s_n \circ \bar{y} = s_i \circ \bar{y}$.

Or, using (b),

(d) $\bar{T} \circ e \circ \bar{y} \twoheadrightarrow s'_i \circ e \circ \bar{y} = s_i \circ \bar{y}$.

But, from (a) we obtain

(e) $s'_i \circ e \circ \bar{y} = s'_i \circ \bar{y}$.

Now since by (a) $e \circ \bar{y} = \bar{y}$, we have $\bar{T} \circ e \circ \bar{y} = \bar{T} \circ \bar{y} = \bar{T}$ (since y cannot be $\perp$ since $p_E:y = T$). Therefore the qualification of (d) can be dropped and from (d) and (e) we have

(f) $s_i \circ \bar{y} = s'_i \circ \bar{y}$.

Therefore, replacing $s_i \circ \bar{y}$ by $s'_i \circ \bar{y}$ in (b),

(g) $e \circ \bar{y} = Es_1 \circ \bar{y} \ldots s'_i \circ \bar{y} \ldots s_n \circ \bar{y} = e' \circ \bar{y}$.

And from (a) and (g) we get $e' \circ \bar{y} = \bar{y}$, hence $e':y = y$. Thus $p'_E:y = T$, as required. Q.E.D.

(D) Extended definitions

Let $Ex_1 \ldots x_n$ be distributive and invertible. Let $Ffx_1 \ldots x_n$ be distributive in $x_1 \ldots x_n$. Then

(7) xdef $f \circ Ex_1 \ldots x_n = Ffx_1 \ldots x_n$

is an extended definition of the function symbol f. It is understood that the function f defined by (7) is the least function that satisfies, for all functions $f_1 \ldots f_n$,

(8) $f \circ Ef_1 \ldots f_n = \bar{T} \circ Ef_1 \ldots f_n \rightarrow Ff f_1 \ldots f_n \; ; \; \bar{I}$,

and that satisfies also the following condition:

(9) For all objects z, $f{:}z \neq \perp$ implies that there are functions $f_1 \ldots f_n$ and an object y such that $z = (Ef_1 \ldots f_n){:}y$.

Together, the requirement that f satisfy (8) and (9) guarantees that f satisfies assumptions 1 and 2 given in the introductory discussion.

Example of an extended definition

xdef $g \circ [x,y] = eq0 \circ x \rightarrow y; \; g \circ [sb \circ x, mult \circ [y,x]]$

(where sb is subtract 1 and mult is multiply) defines an iterative function such that $g \circ [id, \bar{1}]$ is the factorial function. The proper definition of g is

Def $g = eq \circ [[1,2], id] \rightarrow (eq0 \circ 1 \rightarrow 2; \; g \circ [sb \circ 1, mult \circ [2,1]]); \; \bar{I}$.

The extended definition theorem

It is convenient to prove the following lemmas before we come to the main theorem.

Lemma 1 (qualified substitution) Given:

(a) $Ex_1 \ldots x_n v_1 \ldots v_m$ distributive in $v_1 \ldots v_m$.

(b) $p \twoheadrightarrow g_i = g_i'$ for some i in $[1 \ldots m]$.

Then, for all $f_1 \ldots f_n$, $g_1 \ldots g_m$,

(c) $p \twoheadrightarrow Ef_1 \ldots f_n g_1 \ldots g_i \ldots g_n = Ef_1 \ldots f_n g_1 \ldots g_i' \ldots g_n$.

Proof Let $f_1 \ldots f_n$, $g_1 \ldots g_m$ be given and let (b) hold for g_i and g_i'. Let x be any object such that $p{:}x = T$. Then $g_i \circ \bar{x} = g_i' \circ \bar{x}$ by (b). And

$$(Ef_1\ldots f_n g_1\ldots g_i\ldots g_n)\circ\bar{x} = Ef_1\ldots f_n g_1\circ\bar{x}\ldots g_i\circ\bar{x}\ldots g_n\circ\bar{x}$$
$$= Ef_1\ldots f_n g_1\circ\bar{x}\ldots g_i'\circ\bar{x}\ldots g_n\circ\bar{x} = (Ef_1\ldots f_n g_1\ldots g_i'\ldots g_n)\circ\bar{x}$$

by the distributivity of E. Thus (c) holds for any x for which p:x=T, as required. Q.E.D.

Lemma 2 Given:

(a) $Ex_1\ldots x_n$ distributive and invertible by $s_1\ldots s_n$.

(b) $Ffx_1\ldots x_n$ distributive in $x_1\ldots x_n$.

Then, for all f, $f_1\ldots f_n$,

(c) $\bar{T}\circ Ef_1\ldots f_n \twoheadrightarrow (Ffs_1\ldots s_n)\circ(Ef_1\ldots f_n) = Fff_1\ldots f_n$, and

(d) $\bar{T}\circ Ef_1\ldots f_n \twoheadrightarrow (Es_1\ldots s_n)\circ(Ef_1\ldots f_n) = Ef_1\ldots f_n$.

Proof Let $f_1\ldots f_n$ be given. Let $e = Ef_1\ldots f_n$. Then, since E is invertible,

(e) $\bar{T}\circ e \twoheadrightarrow s_i\circ e = f_i$.

And since F is distributive in $_{1\cdots\ n}$,

(f) $(Ffs_1\ldots s_n)\circ e = Ffs_1\circ e\ldots s_n\circ e$.

Then, applying lemma 1 to (f) and (e) n times gives

(g) $\bar{T}\circ e \twoheadrightarrow (Ffs_1\ldots s_n)\circ e = Fff_1\ldots f_n$.

This establishes (c); and (d) is a special case of (c) since E is distributive. Q.E.D.

Lemma 3 Given: $\bar{T}\circ f \twoheadrightarrow g\circ f = w\circ f$. Then $g\circ f = w\circ f$.

Proof Let x be given. If f:x is defined, then $w\circ f:x = g\circ f:x$, by the hypothesis. If $f:x = \perp$, then both sides of the conclusion give $\perp$. Q.E.D.

Lemma 4 Given:

(a) $Ex_1\ldots x_n$ distributive and invertible by $s_1\ldots s_n$.

(b) p_E a range predicate for E: $p_E = eq\circ[Es_1\ldots s_n, id]$.

Then, for any functions $f_1\ldots f_n$,

(c) $p_E\circ Ef_1\ldots f_n = \bar{T}\circ Ef_1\ldots f_n$

Proof Let $f_1\ldots f_n$ be given. Let $e = Ef_1\ldots f_n$. Then

(d) $p_E\circ e = eq\circ[Es_1\ldots s_n, id]\circ e = eq\circ[(Es_1\ldots s_n)\circ e, e]$
$= eq\circ[Es_1\circ e\ldots s_n\circ e,\ e]$.

Let $Hx_1\ldots x_n y = eq\circ[Ex_1\ldots x_n, y]$. Then H is distributive and, by (d),

(e) $p_E\circ e = Hs_1\circ e\ldots s_n\circ e\ e$.

Since E is invertible,

(f) $\bar{T}\circ e \twoheadrightarrow s_i\circ e = f_i$.

Using lemma 1 n times with (f) and the right side of (e) gives

(g) $\bar{T}\circ e \twoheadrightarrow p_E\circ e = Hf_1\ldots f_n e = eq\circ[e,e] = \bar{T}\circ e$.

By lemma 3 we then have

(h) $p_E\circ e = \bar{T}\circ e$. Q.E.D.

EXTENDED DEFINITION THEOREM Given:

(a) $Ex_1\ldots x_n$ distributive and invertible by $s_1\ldots s_n$.

(b) $Ffx_1\ldots x_n$ distributive in $x_1\ldots x_n$.

(c) p_E a range predicate for E .

Then, for any function f, f satisfies (10) for all functions $f_1\ldots f_n$ and f satisfies (11) if and only if f satisfies (12), where

(10) $f\circ Ef_1\ldots f_n = \bar{T}\circ Ef_1\ldots f_n \rightarrow Fff_1\ldots f_n \;;\; \bar{I}$

(11) For all objects x, $f{:}x \neq \perp$ implies that there are functions $f_1\ldots f_n$ and an object y such that $x = (Ef_1\ldots f_n){:}y$.

(12) $f = p_E \rightarrow Ffs_1\ldots s_n \;;\; \bar{I}$.

Proof Only if: Let f satisfy (10) and (11). Let $Hf = Ffs_1\ldots s_n$. Let $e' = Es_1\ldots s_n$. Then, since f satisfies (10) for all $f_1\ldots f_n$, it satisfies (10) for $s_1\ldots s_n$. Thus f satisfies

(d) $f\circ e' = \bar{T}\circ e' \rightarrow Hf \;;\; \bar{I}$.

Now $\bar{I} = \bar{I}\circ e'$ and by lemma 2, $\bar{T}\circ e' \twoheadrightarrow Hf\circ e' = Hf$, therefore we can replace Hf by $Hf\circ e'$ in (d) since the qualification $\bar{T}\circ e'$ will always hold. Finally, by lemma 4, $\bar{T}\circ e' = p_E\circ e'$. We can thus rewrite (d) as

(e) $f\circ e' = p_E\circ e' \rightarrow Hf\circ e';\ \bar{I}\circ e'$

Now, since $p_E = eq\circ[e',id]$,

(f) $p_E \twoheadrightarrow e' = id$.

We want to use the qualified substitution lemma to replace e' by id in (e); to do so we need a distributive form. Let $Wx = p_E\circ x \rightarrow Hf\circ x;\bar{I}\circ x$. Then W is distributive and (e) becomes

(g) $f\circ e' = We'$.

Applying lemma 1 to both sides of (g) using (f) gives $p_E \twoheadrightarrow f\circ id = Wid$, that is, using the definition of W and the properties of id,

(h) $p_E \twoheadrightarrow f = p_E \rightarrow Hf;\ \bar{I}$.

We now assert that

(i) For all objects x, $f{:}x \neq \perp$ implies $p_E{:}x = T$.

To see that (i) holds, let $f{:}x \neq \perp$; then by (11) $x = (Ef_1\ldots f_n){:}y$ for some functions $f_1\ldots f_n$ and some object y. Since $f{:}x \neq \perp$, $x \neq \perp$ (f is strict), hence $(\bar{T}\circ Ef_1\ldots f_n){:}y = T$. Thus by lemma 2, $(e'\circ Ef_1\ldots f_n){:}y = (Ef_1\ldots f_n){:}y$. But $(e'\circ Ef_1\ldots f_n){:}y = e'{:}(Ef_1\ldots f_n : y) = e'{:}x$. And so

$e':x = x$ and $p_E:x = T$. Thus (i) holds as claimed.

Now the contrapositive of (i) is

(j) For all x, if $p_E:x \neq T$, then $f:x = \perp$.

Together (h) and (j) show that the unqualified equation

(k) $f = p_E \rightarrow Hf; \bar{I}$

holds for every object x, for if $p_E:x = T$, then (k) holds by (h); otherwise it holds by (j). This completes the only if part of the proof.

If Let f satisfy (12). We want to show it satisfies (10) for all $f_1 \ldots f_n$ and that it satisfies (11). Let $f_1 \ldots f_n$ be given. Let $e = Ef_1 \ldots f_n$. Let $Hf = Ffs_1 \ldots s_n$ as before. Then if we compose e with both sides of (12) and distribute it on the right side by law II.1, we have

(l) $f \circ e = p_E \circ e \rightarrow Hf \circ e; \bar{I} \circ e$

Since $\bar{I} \circ e = \bar{I}$, and by lemma 4 $p_E \circ e = \bar{T} \circ e$, (l) becomes

(m) $f \circ e = \bar{T} \circ e \rightarrow Hf \circ e; \bar{I}$.

By lemma 2

(n) $\bar{T} \circ e \twoheadrightarrow Hf \circ e = Fff_1 \ldots f_n$.

Since $Hf \circ e$ in (m) always meets the qualification of (n), we can use (n) to obtain the following from (m).

(o) $f \circ e = \bar{T} \circ e \rightarrow Fff_1 \ldots f_n; \bar{I}$.

Thus f satisfies (10) for arbitrary $f_1 \ldots f_n$. Now let x be such that $f:x \neq \perp$; then by (12) $p_E:x = T$ and hence $(Es_1 \ldots s_n):x = x$. Therefore there are functions $s_1 \ldots s_n$ and object x such that $x = (Es_1 \ldots s_n):x$, as required by (11), thus f satisfies (11).

Q.E.D.

Related Work

It is not possible to even mention here all of the many articles concerned with proving theorems about programs, or even, more narrowly, those concerned with transforming programs from one form to another without changing their meaning. I believe the complexities of conventional, von Neumann programming languages make it unlikely that proof or transformation methods designed for them will ever become practical for the programmer to use himself without the aid of automated systems. In any case the flavor of most work on proof techniques for von Neumann languages (much of the best of it is based on the axiomatic approach founded by Hoare [Hoar69]) is very different from the approach proposed here. Therefore we shall discuss only work directed toward proof or transformation of purely functional programs.

Much of this related work concerns proofs about recursively defined functions; its foundations lie principally in the work of Church [Chur41], Curry [Curr58], Kleene [Klee52], and Scott [Scot70]. Among the early work in the field is that of de Bakker [deB&S69], Burstall [Burs69], Floyd [Floy67], Manna and his associates [M&P70], McCarthy [McCa63], J. H. Morris, Jr. [Morr71], Naur [Naur66], and Park [Park69]. Much of this work is well summarized in [MNV73].

In contrast to most work on proof and transformation methods for recursively defined functions, the approach here attempts to provide orderly methods for proofs about non-recursively defined, as well as about recursively defined ones, within a unified algebraic system capable of expressing many useful general theorems about both kinds of functions.

Recently there has been a rapid increase in work on the relationship of functional languages and new computer architectures. In view of its rapid growth and small overlap with the work reported here, I shall leave discussion of it to a future paper.

F. Y. Raymond [Raym75][Raym77] has developed, independently and earlier, an interesting algebra of functions having many similarities to the present FP algebra. He has made a somewhat different use of his algebra than we have of ours (one of his results is a generalization of the well-known Bohm & Jacopini "factorization" theorem [B&J66]). Two minor differences: (a) his set of functionals seem to make it difficult to express functions of the form $f \circ [g]$ and (b) he includes in his set of functionals the operator S_1 that depends on a "contexte"; this differs again from the approach here. Nevertheless, his axioms have much the same consequences as the FP algebraic laws of [Back78]. His algebra is constructed from a more abstract basis, whereas the FP algebra, developed later, grew out of the FP calculus begun in [Back73]. The algebra sections of [Back78] were written without knowledge of Raymond's work.

There is an interesting and as yet not precisely determined relationship between the linear forms of this paper and the linear recursion schemas in [W&S72]. Here we have defined linear forms H in terms of the semantic properties of H, and then characterized such forms, incompletely, by their structure. They, on the other hand, characterize linear schemas structurally in a style fairly different from ours. Nevertheless, it seems that there is a one-one relationship between our

linear forms and their linear schemas. Our purpose is to formally solve linear equations in order to facilitate reasoning about the functions they define. Their purpose is to characterize those schemas that are flowchartable, and they show that many non-linear schemas as well as linear ones are flowchartable. Flowchartability of schemas involves quite different issues than solvability of equations in our sense; many flowchartable schemas denote infinitely-branching unwound processes that do not have tractable formal solutions; however, [Will80] gives useful formal solutions for two classes of non-linear equations.

Perhaps the most elegant work on transforming functional programs is that of Burstall and Darlington [B&D77]. By their "folding" and "unfolding" methods they are often able to prove two recursively defined functions equivalent in a simpler way than is possible with other methods. It appears that their methods and our algebraic approach complement each other nicely. Although they tend to reason at the object level, it is usually a simple matter to restate their arguments at the function level. The use of algebraic laws and theorems can then sometimes help in making precise the arguments required at certain steps in a transformation. If used with care, the algebraic approach may help clarify questions about the domain in which an equation holds. Since algebraic equations preserve termination properties (i.e., f=g means f and g are both defined or undefined together), it may be possible to develop an algebraic modification of the folding-unfolding method that preserves termination properties, which is not presently the case.

The algebraic approach may sometimes help with another difficulty in the folding-unfolding method. It is sometimes easier to define the final function (that is, the target of a set of transformations) than it is to invent just the right definitions and instantiations (relating the original and final functions) that form the basis of the method. In such a case it may be possible to use the linear expansion theorem or a non-linear extension of it [Will80] to solve the equations for the original and final functions and then use the solutions to prove the desired relationship between them. This approach may help in another way: it may make clear that the proposed final function is not equivalent to the original in certain special cases. The identification of these cases is often the most troublesome part; after the adjustment of the proposed final function, the proof of equivalence is then often not difficult, whereas getting the right proposal is hard.

In [Burg73] Burge proposes some interesting multi-parameter functionals in the lambda style that are useful for building programs at the function level and that go beyond the usual functionals employed in the lambda style (e.g., "map" "maplist"). Such functionals are particularly useful in the area of "streams", where Burge uses stream functions in place of objects and hence must combine functions instead of objects.

In addition to the work discussed above, there is a great amount of work on automating proofs about functional programs using various theorem-proving or proof-checking techniques. Important efforts of this kind include the work of Milner and his colleagues on LCF [Miln76], that of Guttag, Horowitz and Musser [GH&M78] on theorem proving in equational systems using rewrite rules, and that of Boyer and Moore on proving theorems about LISP functions [B&M79]. In addition there are many extensive efforts to create better languages and better foundations that will simplify the problems of reasoning about programs. Important among the latter are the work of the ADJ group on initial algebra semantics [GTW&W77], that of Burstall and Goguen on the use of theories for making specifications [B&G77],[B&G80], that of Arbib and Manes on partially additive semantics and canonical fixed points [A&M79],[A&M80], and that of Turner on SASL [Turn79],[Turn79a].

All of the last-mentioned work is too diverse and it is still too early to be able to say much at this point about its relationship with the present work. A few remarks are possible, however. One point is that much of this related work is based on theories that are stronger than the kind of limited equational theory we are trying to develop and hence can prove theorems that cannot be proved in our algebra (e.g., [Miln76],[B&M79]). However, it is likely that equational theorem provers such as those discussed in [GHM78] and [Muss78] will be of help in proving theorems in our algebra.

There seems to be a close relationship between the partially additive functions of [A&M79] and the functions on sets proposed by Martin [Mart80] that develop an FP-like style in a new domain. We hope to study this relationship in the future.

There is also the question of the possible value of looking again at some of the above related work that presently uses an object level approach from a viewpoint that emphasizes the function level more. In particular, the work on data types in [GHM78] and elsewhere and the FP algebra of programs seem to complement each nicely and have the pos-

sibility of together producing some simplification in the work on data types.

Acknowledgments

It is a pleasure to acknowledge my debt to John H. Williams for many helpful discussions and for his pointing out many mistakes in early drafts. I have mentioned in the text my debt to Williams and to Michael Arnold for specific contributions to the work on linear forms and equations.

References

[A&M79] ARBIB, M.A. and MANES, E.G. The pattern-of-calls expansion is the canonical fixpoint for recursive definitions. Tech. Rep. Math. and Statistics, Univ. of Mass., Amherst, Mass., Jan. 1979.

[A&M80] ARBIB, M.A. and MANES, E.G. Partially additive categories and flow-diagram semantics. J. Algebra 62, 1 , Jan. 1980.

[Arno80] ARNOLD, M. (Private communication), May, 1980.

[B&D77] BURSTALL, R.M. and DARLINGTON, J. A transformation system for developing recursive programs. JACM 24,1, Jan. 1977.

[B&G77] BURSTALL, R.M. and GOGUEN, J.A. Putting theories together to make specifications. Proc. International Joint Conference on Artificial Intelligence, Cambridge, Mass. 1977.

[B&G80] BURSTALL, R.M. and GOGUEN, J.A. The semantics of Clear, a specification language. Proc. 1979 Copenhagen Winter School on Abstract Software Specifications, Copenhagen. Feb. 1980.

[B&J66] BOHM, C. and JACOPINI, G. Flow diagrams, Turing machines, and languages with only two formation rules. CACM 9,5, May 1966.

[B&M79] BOYER, R.S. and MOORE, J.S. A computational logic. Academic Press, New York, 1979.

[Back73] BACKUS, J. Programming language semantics and closed applicative languages. Conf. Record, ACM Symp. on Principles of Programming Languages, Boston, Mass. Oct. 1973.

[Back78] BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. CACM 21,8, Aug. 1978.

[Back79] BACKUS, J. On extending the concept of "program" and solving linear functional equations. Draft paper distributed at Summer Workshop on Programming Methodology, Univ. of Calif, Santa Cruz, Aug. 1979.

[Burg73] BURGE, W.H. Even more structured programming. Tech. Rept. RC4604 IBM Thos. J. Watson Research Center, Yorktown Hts, NY, Oct. 1973.

[Burs69] BURSTALL, R.M. Proving properties of programs by structural induction. Computer J. 12,1, Feb. 1969.

[Cadi72] CADIOU, J.M. Recursive definitions of partial functions and their computations. PhD Thesis, Computer Science Dept., Stanford Univ., Stanford, Calif. Mar. 1972.

[Chur41] CHURCH, A. The calculi of lambda-conversion. Princeton Univ. Press, Princeton, NJ. 1941.

[Curr58] CURRY, H.B. and FEYS, R. Combinatory logic. North-Holland, Amsterdam, 1958 (second printing 1968)

[deB&S69] deBAKKER, J.W. and SCOTT, D. A theory of programs. (Unpublished memo.) Aug. 1969.

[GH&M78] GUTTAG, J.V., HOROWITZ, E., and MUSSER, D.R. Abstract data types and software validation. CACM 21,12, Dec. 1978.

[GTW&W77] GOGUEN, J.A., THATCHER, J.W., WAGNER, E.G., and WRIGHT, J.B. Initial algebra semantics and continuous algebras. JACM 24,1, Jan. 1977.

[Hoar69] HOARE, C.A.R. An axiomatic basis for computer programming. CACM 12,10, Oct. 1969.

[Klee52] KLEENE, S.C. Introduction to metamathematics. D. van Nostrand, Princeton, NJ 1952.

[Land64] LANDIN, P.J. The mechanical evaluation of expressions. Computer J. 6,4, 1964.

[MNV73] MANNA, Z., NESS, S., and VUILLEMIN, J. Inductive methods for proving properties of programs. CACM 16,8, Aug. 1973.

[M&P70] MANNA, Z. and PNUELI, A. Formalization of properties of functional programs. JACM 17,3, July 1970.

[Mart80] MARTIN, J.J. Typed functional programming systems. Tech. Rept. Comp. Sci. Dept., Virginia Polytechnic Inst. and State Univ. Blacksburg, VA. 1980.

[McCa63] MCCARTHY, J. A basis for a mathematical theory of computation. (in) Computer Programming and Formal Systems, P. Braffort and D. Hirshberg (eds.), North Holland, Amsterdam, 1963.

[Miln76] MILNER, R. Models of LCF. Mathematical Centre Tracts 82, Univ. of Edinburgh, Edinburgh, 1976.

[Morr71] MORRIS, J.H., Jr. Another recursion induction principle. CACM 14,5, May 1971.

[Muss78] MUSSER, D.R. Convergent sets of rewrite rules for abstract data types. Tech. Rept., Univ. of So. Calif. Information Scis. Inst., Marina del Rey, Calif. Dec. 1978.

[Naur66] NAUR, P. Proof of algorithms by general snapshots. BIT 6, 1966.

[Park69] PARK, D. Fixpoint induction and proofs of program properties. (in) Machine Intelligence 5, B. Meltzer and D. Michie, (eds.)

[Raym75] RAYMOND, F.H. Note sur l'algebre des fonctions. R.A.I.R.O. R-3, Dec. 1975.

[Raym77] RAYMOND, F.H. Note sur la suppression des étiquettes en programmation. R.A.I.R.O. 11,1, 1977.

[Scot70] SCOTT, D. Outline of a mathematical theory of computation. Proc. Fourth Annual Princeton Conf. on Information Sciences and Systems, Princeton Univ. 1970.

[Turn79] TURNER, D.A. SASL language manual. Tech. Rept., Univ. of Kent, Canterbury, Aug. 1979.

[Turn79a] TURNER, D.A. A new implementation technique for applicative languages. Software--Practice and Experience 9, 1979.

[W&S73] WALKER, S.A. and STRONG, H.R. Characterizations of flowchartable recursions. J. Computer and System Sciences 7,4, Aug. 1973.

[Will80] WILLIAMS, J.H. On the development of the algebra of functional programs. Tech. Rept. RJ2983, IBM Research Lab., San Jose, Calif. Oct. 1980.