RJ2983 (37118) 10/30/80
Computer Science

Research Report

ON THE DEVELOPMENT OF THE ALGEBRA OF FUNCTIONAL PROGRAMS

John H. Williams

IBM Research Laboratory
San Jose, California 95193

IBM Research Division
Yorktown Heights, New York · San Jose, California · Zurich, Switzerland

B-9
10

Copies may be requested from:
IBM Thomas J. Watson Research Center
Distribution Services
Post Office Box 218
Yorktown Heights, New York 10598

RJ2983 (37118) 10/30/80
Computer Science

ON THE DEVELOPMENT OF THE ALGEBRA OF FUNCTIONAL PROGRAMS

John H. Williams

IBM Research Laboratory
San Jose, California 95193

Abstract: In this paper we further the development of the algebraic approach to reasoning about functional programs that was introduced by John Backus in his Turing Award Lecture. We give precise definitions for the foundations on which the algebra is based, and we prove some new expansion theorems that broaden the class of functions for which this approach is applicable. In particular we define the class of "overrun-tolerant" forms -- nonlinear forms that include some of the familiar divide and conquer program schemes, we prove an expansion theorem for such forms, and we use that theorem to show how to derive expansions for some programs defined by nonlinear forms.

ON THE DEVELOPMENT OF THE ALGEBRA OF FUNCTIONAL PROGRAMS

John H. Williams

IBM Research Laboratory
5600 Cottle Road
San Jose, California 95193

Section 1. Introduction

In [Ba 78] Backus proposes an alternative to the conventional style of programming, which he calls functional programming. One of the advantages of this functional approach is that it allows one to develop an "algebra" of functional programs, *i.e.* a means for reasoning about programs by manipulating (or transforming) the programs themselves. This algebra is a collection of laws and theorems, based on functional identities, that allows the programmer to reason about his programs by transformations. Backus has provided algebraic methods for dealing with recursive definitions of a certain type, the "linear" forms.

In this paper we further the development of this algebraic approach by precisely defining the foundations on which it is based and by proving some new expansion theorems that broaden the class of functions for which this approach is applicable. In particular we define the class of "overrun-tolerant" forms -- nonlinear forms that include some of the familiar divide and conquer program schemes, we develop and prove an expansion theorem for such forms, and we use that theorem to show how to derive expansions for some programs defined by nonlinear forms.

In Section 2 we describe the algebra of functional programs and we discuss some of the advantages of this approach. In Section 3 we formally define some extensions to the set of functional programming primitives that will enable us to precisely define what is meant by the solution to a recursive functional equation and to formally justify the correctness of this algebraic approach. In Section 4 we consider the problem of finding expansion theorems for recursively defined functional programs. We then prove an expansion theorem for a particular, nonlinear class of programs, the overrun-tolerant programs, and we use the expansion to reason about a program defined by a nonlinear functional form. Finally in Section 5 we suggest a generalization that will enable us to find expansions for some highly nonlinear forms such as the defining equation for McCarthy's "91 function", and we pose some problems for further research. In the remainder of this section we review the basic notions of functional programming and define the primitive functions and combining forms that will be used in the paper. A complete development and description of the functional style of programming can be found in [Ba 78]. The reader who is completely unfamiliar with these notions will want to refer to that paper.

A *program* in the functional programming style is simply an expression representing a *function* that maps *objects* to *objects*. This will completely specify the concept of *program* when we recall that:

1) Given a set A of atoms, the set O of objects is recursively defined by:

   a) $\perp$ is in O ("bottom" denotes the undefined object),

b) A is contained in O (all atoms are objects),

c) if $x_1, \ldots, x_n$ are in O and $x_i \neq \perp$, then $\langle x_1, \ldots, x_n \rangle$ is in O, (all sequences of non-$\perp$ objects are themselves objects),

d) $\langle x_1, \ldots, x_n \rangle = \perp$ if for some i, $x_i = \perp$ (the sequence constructor is $\perp$-preserving).

2) Given a set P of primitive functions and a set Γ of *combining forms* (functionals that map functions into functions), the set of *functions* **F** is defined recursively by:

a) P is contained in **F** (all primitives are functions),

b) if $f_1, \ldots, f_n$ are in **F** and **C** in Γ is a functional that takes n arguments, then $\mathbf{C}(f_1, \ldots, f_n)$ is in **F**,

c) the function f defined by the equation $f = Ef$ is in **F** if E is a functional form constructed by applying combining forms chosen from Γ to functions chosen from **F** together with the function symbol f. The interpretation of such an equation is that the meaning of f applied to an object is taken to be the meaning of Ef applied to that object.

3) The application of a function f in **F** to an object x in O is denoted $f{:}x$.

4) The sets P and Γ will be chosen such that all functions f in **F** are *strict*, *i.e.* for all f, $f{:}\perp = \perp$.

In this paper we will assume that the set of atoms consists of the integers together with all finite, nonempty strings of capital letters. The following are all examples of atoms:

1, AB, T, -7, ABCDE

Given this choice of atoms, some examples of objects are:

$\perp$, 1, <A, B>, <A, <1, B>>, <>

A typical choice of a set of primitive functions will contain some functions for producing and decomposing sequences, some arithmetic functions, and some predicates. The following is a list of some of the primitives used in this paper:

function	description and examples
1, 2, ...	selector functions: the selector i applied to an object x produces the i^{th} element of x if x is a sequence of at least i elements and produces $\perp$ otherwise. (Note that we are using a single graphic to denote both the *object* 1 and the *selector function* 1. Which meaning is intended will always be clear from the context.)
tl	the tail function: removes the first element of a sequence; produces $\perp$ when applied to a non-sequence or to the empty sequence.
id	the identity function: $\mathrm{id}{:}x = x$ for all x in O.

eq	test for equality: eq:x produces T if x is a pair of identical objects, produces F if x is a pair of non-identical objects, and produces $\perp$ otherwise; *e.g.*, eq:<<A, 3>, <A, 3>> = T, eq:<A, 3> = F, eq:<3, 3, 3> = $\perp$.
eq0	test for zero: eq0:0 = T.
gt	greater than: gt:<3, 2> = T ; gt:<3, A> = $\perp$; gt:<3, 2, 1> = $\perp$.
ge	greater than or equal to: ge:<3, 3> = T.
+, −, ×	arithmetic operations: +:<3, 4> = 7 ; +:<3> = $\perp$; +:<3, A> = $\perp$.
a	add one: a:7 = 8 ; a:A = $\perp$.
s	subtract one: s:−3 = −4.

Of the several combining forms described in [Ba 78] we will use the following:

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

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

Conditional $(p \rightarrow f; g):x = f:x$ if $p:x = T$,
$g:x$ if $p:x = F$,
$\perp$ otherwise.

Constant $\bar{x}:y = x$ if $y \neq \perp$,
$\perp$ otherwise.

Insert Left $(/f):\langle x_1 \rangle = x_1$

$(/f):\langle x_1, \ldots, x_n \rangle = f:\langle f:\langle x_1, \ldots, x_{n-1} \rangle, x_n \rangle$

Note that this definition of insert differs from that of [Ba 78] where it was defined to associate to the right. As in that paper we will adopt some informal precedence rules and frequently omit parentheses when no confusion can arise. For example,

$$p \rightarrow q; r \rightarrow s; t = p \rightarrow q; (r \rightarrow s; t)$$

$$/f \circ g = (/f) \circ g$$

$$\text{eq0} \circ a \rightarrow b; c \circ d = (\text{eq0} \circ a) \rightarrow b; (c \circ d)$$

Section 2. The Algebra of Functional Programs

In this section we will discuss the use of the algebraic approach to reasoning about programs. This approach differs from the conventional methods for reasoning about programs in that it is based neither on the notion of an inductive proof rule such as computational induction nor on fixed point arguments such as recursion induction. Instead it is based on a

collection of algebraic identities: statements that assert the equality of general classes of functional programs. These identities are then used to show directly the equality of the functions computed by different programs.

It happens that the notation of functional programming is also well-suited to the techniques of recursion induction, since if f is defined by the equation

$$f = Ef$$

we can frequently show that some program g satisfies that equation and therefore that $f \leq g$ by using the algebraic identities to show that $Eg = g$. However, in this paper we will concentrate on the conceptually simpler technique of reasoning by algebraically transforming the programs themselves, instead of manipulating program descriptions such as recursive defining equations.

The algebraic statements we will be using can be grouped into three major categories: algebraic laws, derived theorems, and expansion theorems. The algebraic laws are like axioms in that they require no proof in the algebra; rather they are identities that follow from the definitions of the particular combining forms and primitives. (Alternatively, these laws might be viewed as axiomatic specifications for the behavior of the primitives and forms rather than mere consequences of the (operational) definition of the primitives and forms, but such a consideration is outside the scope of this paper and of no consequence to the development of the algebra.) The algebraic laws, however obtained, are taken as the axioms of the algebra of programs. Whereas functional programs are variable-free, these algebraic laws have variables that range over functional programs. For example, the law

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

asserts that for all programs f, g, and h the composition on the left is equal to the construction on the right. Thus these laws with free program variables are similar to the axiom schemata of mathematical logic.

The second category of statements, the derived rules, are those identities that follow from, *i.e.* can be proved directly from, the algebraic laws will give an example below.

Finally, the expansion theorems are algebraic statements that give nonrecursive forms for programs that are defined by recursive equations. For example in [BA 79] Backus proves the following:

Theorem (*Linear Expansion*). If

$f = p \rightarrow q; Hf$,

H is $\overline{\perp}$-preserving, and

H is "linear", *i.e.* there is an H_1 such that for all a, b, and c,
$H(a \rightarrow b; c) = H_1 a \rightarrow Hb; Hc$,

then $f = p \rightarrow q; \ldots; H_1{}^n p \rightarrow H^n q; \ldots$.

We will discuss the interpretation and the formal treatment of this apparently infinite object in the following section. In the remainder of this section we give the flavor of this algebraic approach by considering an example.

We will use the following algebraic laws without further motivation; arguments justifying their adoption as axioms can be found in [Ba 78].

(A1) $h \circ (p \rightarrow f;g) = p \rightarrow h \circ f;\ h \circ g$

(A2) $(p \rightarrow f;g) \circ h = p \circ h \rightarrow f \circ h;\ g \circ h$

(A3) $/f \circ [g_1, \ldots, g_{n+1}] = f \circ [\ /f \circ [g_1, \ldots, g_n]\ ,\ g_{n+1}]$

(A4) $/f \circ [g] = g$

(A5) $[a,b] \circ c = [a \circ c\ ,\ b \circ c]$

(A6) $1 \circ [a,b] = a$, in the domain of definition of b

(A7) $2 \circ [a,b] = b$, in the domain of definition of a

(A8) $a \rightarrow (a \rightarrow b;\ c);\ d = a \rightarrow b;\ d$

These are but a few of the many algebraic laws that can be inferred from the definitions of the combining forms of [BA 78]. Indeed, one of the primary criteria for the inclusion of a combining form in a functional programming language is the extent to which its interaction with the other forms exhibits nice algebraic properties. Consequently, the set of generally applicable laws that can be obtained for functional programming languages is relatively rich compared to those that can be obtained for conventional languages.

This rich set of axioms in turn generates many general theorems about functional program equivalence. As an example, consider the following theorem, which relates the powers of construction to the combining form insert. (The n^{th} power of a function x, x^n, is defined to be $x \circ x^{n-1}$ for $n > 0$, and id for $n = 0$.) This theorem can be used to rewrite any of a large class of recursively defined programs that accomplish iteration by means of tail recursion, regardless of the particular types of data objects or primitive functions involved. We will state and prove this theorem and use it to transform a sample iterative program.

Theorem 1. For $n \geq 1$ and all programs f, a, b and c,

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

Proof: (By induction on n)

Basis: $[f\ ,\ a \circ 2] \circ [b,c] = [f \circ [b,c],\ a \circ 2 \circ [b,c]]$ (by (A5))

$= [f \circ [b,c],\ a \circ c]$ (by (A7) if b is defined, and by $\bot = \bot$ otherwise)

$= [f \circ [/f \circ [b],\ c],\ a \circ c]$ (by (A4))

$= [/f \circ [b,c],\ a \circ c]$ (by (A3))

Inductive step: For ease of notation define E to be:

$$E = [/f \circ [b, c, a \circ c, \ldots, a^{n-1} \circ c],\ a^n \circ c]$$

Then

$[f, a \circ 2]^{n+1} \circ [b,c] = [f, a \circ 2] \circ [f, a \circ 2]^n \circ [b,c]$ (by def of x^n)

$= [f,a\circ 2] \circ E$ (by induction)

$= [f\circ E, a\circ 2\circ E]$ (by (A5))

$= [f\circ E, a\circ a^n\circ c]$ (by (A7) if b is defined, and by $\bar{\perp} = \bar{\perp}$ otherwise)

$= [f\circ E, a^{n+1}\circ c]$ (by def of x^n)

$= [/f\circ[b,c,a\circ c,\ldots,a^n\circ c],\ a^{n+1}\circ c]$ (by (A3)) QED

We stress that the validity of this equation is independent of the properties of the particular programs f, a, b, and c. The theorem follows directly from the properties of the combining forms construction, insert, and composition.

Consider now the problem [Di 76] of finding the index of a maximum of a function g on the interval 1:N, *i.e.* of computing k such that $1 \le k \le N$ and $g(k) \ge g(j)$ for $1 \le j \le N$. Since the functional programming style has no nondeterministic control mechanisms such as the **do** construct of [Di 76], we will also require that k be as large as possible. Then a possible solution in the language of guarded commands is:

```
{N ≥ 1  and g  defined on  (1,...,N)}

k, j :=  N, N-1;
do j > 0  →
    if  g(k) ≥ g(j)  →  j := j-1
      □g(k) < g(j)  →  k, j := j, j-1
    fi
od {for 1 ≤ i ≤ N, g(k) ≥ g(i) }
```

with loop invariant

$0 \le j < k \le N$ *and* (for $j < i \le N$, $g(k) \ge g(i)$) .

While such an iterative approach is not in the functional style, we can transliterate this program into the language of functional programs by building a sequence to represent the "state" (k, j) and by using tail recursion to mimic the loop. Thus (assuming that g is a program to evaluate the given function) we can express the above solution as the composition

(1) iter ∘ [id,s]

where iter is defined to be the solution of the recursive definition

(2) **def** iter = eq0∘2 → 1; iter∘[2max, s∘2],

and 2max is defined (for notational convenience) to be

(3) **def** 2max = (ge∘[g∘1, g∘2] → 1;2)

Now the recursive definition of iter is of the form

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

where p = eq0∘2, q = 1, and $Hf = f\circ r$, and where r denotes the function

$$r = [2max, s \circ 2].$$

Moreover, the form $Hf = f \circ r$ is linear in f with $H_1 f = Hf$ and is $\overline{\perp}$-preserving, so according to the linear expansion theorem, f can be expressed as the infinite conditional

(4) $\quad f = p \rightarrow q; \ldots ; H_1{}^n p \rightarrow H^n q; \ldots$

or

(5) $\quad f = p \rightarrow q ; \ldots ; p \circ r^n \rightarrow q \circ r^n; \ldots .$

Therefore, our program (1) can be rewritten as

(6) $\quad (p \rightarrow q; \ldots ; p \circ r^n \rightarrow q \circ r^n; \ldots) \circ [id,s].$

Now using the abbreviation

(7) $\quad G = [2max, s \circ 2]^n \circ [id,s] ,$

we can write the general term of (6) as

(8) $\quad eq0 \circ 2 \circ G \rightarrow 1 \circ G .$

Using Theorem 1 with f = 2max, a = s, b = id, and c = s, we can rewrite G as

(9) $\quad [/2max \circ [id,s,s^2, \ldots , s^n], s^{n+1}].$

Therefore,

$$eq0 \circ 2 \circ G = eq0 \circ s^{n+1} ,$$

and $1 \circ G = /2max \circ [id,s,\ldots,s^n] ,$

so that the general term of (6) can be rewritten

(10) $\quad eq0 \circ s^{n+1} \rightarrow /2max \circ [id,s, \ldots ,s^n],$ or

(11) $\quad eq(n+1) \rightarrow /2max \circ reviota,$

where for each n, eq(n+1) is a predicate that is true if its argument is the integer n+1 and reviota is a primitive that, when applied to an integer m, yields the sequence <m, m-1, ... , 1>. Therefore, we can rewrite the infinite conditional (6) simply as

(12) $\quad /2max \circ reviota$

since by (11) all of the consequents of (6) are identical, and (12) is defined if (6) is. Thus we have started with the transliteration (1) and transformed it into the more representative functional program (12) by using the algebraic approach to reasoning about functional programs.

This example illustrates some of the advantages of this approach over the more conventional methods of reasoning about programs. First, we did not have to map program (1) into some other domain of discourse such as mathematical functions [Ma 73] or predicate transformers [Di 76] in order to reason about it. The language we used to reason about the

program was essentially the language of functional programs. We say "essentially" because we have used some concepts in our algebraic treatment that are not strictly part of the formal style of functional programming. For example, the infinite conditionals of the above argument can not be expressed in the FP system of [Ba 78], nor can the predicate = , which is the main connective of the algebraic laws themselves, since it asserts the strong equality of two functions in the FP system and therefore can not be expressed as an FP program. We will treat these issues in more detail in the next section, but, modulo these extensions, the language of discourse is the same as the language of programs, and the programmer who wants to reason about (*i.e.* to understand) his programs has a single language to master.

Another advantage of this approach is the relative generality of the provable theorems. For example, Theorem 1 can be applied regardless of the particular programs chosen for f, a, b, and c. Because the combining forms of conventional languages (blocks, repetitive loops, and subroutines) have comparatively few of the mathematical properties enjoyed by functional combining forms, proofs of correctness and equivalence for conventional programs typically are specific to one particular program and are not easily generalized.

Further development of this algebraic approach will require efforts of two essentially different kinds. One is the development of a rich set of generally applicable and relatively high-level theorems built up from the algebraic laws and other theorems such as our Theorem 1. These will enable the programmer to reason about his programs without having to resort to first principles every time. The second and more difficult task is the proof of more generally applicable expansion theorems. When a programmer writes a recursively defined functional program, the function being defined is not the functional equation $f = Ef$ *per se*; rather it is the solution to that equation. Consequently, that solution must be expressed as a program in the language of functional programs before the algebraic laws and theorems can be used to manipulate it. (Note that the algebra can be used to transform the right hand side, Ef, of the functional equation, but the right hand side is not in general the solution since it may include instances of the function variable f.)

If a program is defined by a recursive equation

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

we know ([BA 78] and the next section of this paper) that it can be expressed as the "infinite" object:

(14) $\quad p \rightarrow q;\ H(p \rightarrow q; H(p \rightarrow q; ...)).$

Now the form of (14) is intractable. In general there is not much we can do to manipulate or transform it since the algebraic laws deal only with particular combining forms, not general forms H. We need to be able to deal with the object $H(p \rightarrow q; ...)$. If form Hf is linear in f, then by the linear expansion theorem given earlier,

(15) $\quad H(p \rightarrow q; ...) = H_1p \rightarrow Hq; H(...)$,

and we have transformed the object into something that our laws and theorems can manipulate since its main connective is now the familiar conditional instead of the unknown H, and further substitution for the variable f will not affect the part $H_1p \rightarrow Hq$ of the conditional. Indeed, the underlying motivation for the definition of linear forms was the desire to solve equations such as (13).

If we consider a nonlinear form such as

(16) $Hf = h \circ [f \circ i, f \circ j]$,

we might begin to substitute for f in $H(p \rightarrow q; Hf)$ as above, for example:

$$H(p \rightarrow q; Hf) = h \circ [(p \rightarrow q; Hf) \circ i\ ,\ (p \rightarrow q; Hf) \circ j]$$

$$= p \circ i \rightarrow h \circ [q \circ i, (p \rightarrow q; Hf) \circ j]; h \circ [Hf \circ i, (p \rightarrow q; Hf) \circ j]$$

$$= p \circ i \rightarrow (p \circ j \rightarrow h \circ [q \circ i, q \circ j]; h \circ [q \circ i, Hf \circ j]);$$
$$p \circ j \rightarrow h \circ [Hf \circ i, q \circ j]; h \circ [Hf \circ i, Hf \circ j]$$

(17) $$= p \circ i \rightarrow (p \circ j \rightarrow Hq; h \circ [q \circ i, Hf \circ j]); p \circ j \rightarrow h \circ [Hf \circ i, q \circ j]; H^2 f$$

Now (17) has a conditional as the main connective, but the "then" part of the conditional is dependent on f. That is, further substitution for f will result in an infinitely growing "two-dimensional" object that is not readily amenable to manipulation using the algebraic laws and theorems.

If the particular properties of p, q, h, i, and j are such that the cross terms of (17),

$h \circ [q \circ i, Hf \circ j]$ and $h \circ [Hf \circ i, q \circ j]$,

will disappear, then form (16) will admit of a linear-like expansion for $H(p \rightarrow q; Hf)$. Letting $H_1 p = \& \circ [p \circ i, p \circ j]$ we could then write

(18) $H(p \rightarrow q; Hf) = H_1 p \rightarrow Hq; H^2 f$.

Observe that even in this case, form (16) might not be linear. Linearity requires that

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

for all functions a, b, and c. In general that is more restrictive than requiring for particular functions p and q that

$H(p \rightarrow q; Hf) = H_1 p \rightarrow Hq; H^2 f$.

A characterization of linear forms can be found in [Ba 79].

In this paper we will be concerned with proving expansion theorems for programs defined by nonlinear forms. Roughly speaking, our general approach will be to relax the requirement that a form H distribute over all conditionals $a \rightarrow b;c$. Rather we will require that it distribute over just those conditionals that are encountered in building up the solution to $f = p \rightarrow q; Hf$ as an infinite conditional.

Section 3. The Foundations of the Algebraic Technique

In this section we give more precise definitions for some of the concepts that were treated informally in the previous section. In particular we consider carefully what is meant by the solution to a functional equation $f = Ef$, and we define some extended, non-strict primitives for reasoning about functional programs.

We have said that one of the ways of building new functional programs is by a recursive definition, $f = Ef$, where E is a functional form built up from primitive functions, already defined functions, and the function letter f using given combining forms. The semantics of a function so defined is given operationally in [Ba 78] by the meaning function μ, which describes how such a function gets applied to arbitrary objects. In order that we may reason about such a function using the algebra of functional programs, we need some nonrecursive representation of the function to which we can apply our algebraic transformations. That is, we need to have some concrete representation of the solution to the functional equation, and we need to be sure that that representation is in fact the function that μ defines operationally.

In [Ba 78] Backus defines "the solution of $f = Ef$" to be the least fixed point of E. Now in his Ph.D. dissertation [Ca 72] Cadiou gives two possible definitions for the least defined fixed point of a functional equation, the least *strong* fixed point and the least *weak* fixed point. The former results from allowing the solution space to be all of $D^+ \rightarrow D^+$ (where D^+ is some domain augmented by the undefined element $\perp$), while the latter results from restricting the solution space to strict functions, functions that produce $\perp$ whenever one of their arguments is $\perp$. These are different in general, because the base functions (in terms of which the recursive function is being defined) are not necessarily strict. Moreover, he analyzes various computation rules to determine which rules correctly compute the least weak and least strong fixed points, and he shows that there are recursive definitions for which a leftmost innermost computation rule will not correctly compute *either* the weak or strong least fixed point.

Examination of the definition of μ in [Ba 78] shows that it is a variation of a leftmost innermost rule of computation, so it would appear that defining the solution of $f = Ef$ to be either of the two least fixed points would result in a definition that is inconsistent with the function computed by μ. This would be the case if, like Cadiou and Manna [Ma 73], we were to assume only that the primitives and combining forms were monotonic and continuous. In fact we know more; we know that all functions, whether primitive or defined, are *strict*, and we know that the sequence constructor is $\perp$-preserving, *i.e.* $< \ldots , \perp, \ldots > = \perp$. Under these additional assumptions, the least weak and strong fixed points become identical, and the various computation rules, outermost, leftmost outermost, innermost, and leftmost innermost, are all safe rules for computing the least fixed point. Thus the formal semantics is simplified considerably.

It is interesting to note that in Manna's treatment this simplifying restriction is not possible. Since there is only one combining form (composition) for creating new functionals, conditional must necessarily be realized as a (primitive) function. Now if all functions were required to be strict, then any nontrivial functional equation $f = Ef$ whose right hand side contained an occurrence of f would necessarily have the everywhere undefined function $\overline{\perp}$ as its least fixed point. Therefore he is forced to allow non-strict primitives in his treatment. In fact, he introduces only two, the conditional and the constant functions. All his other primitive functions are strict, *i.e.* are natural extensions of partial functions. The functional programming approach of allowing combining forms other than composition permits the restriction of all functions, both primitive and defined, to strict functions since the conditional can be realized as a combining form or *functional* rather than a function. This functional is non-strict, but the function that results from applying the conditional functional to any three functions *is* strict. For example the function zerofilter, which is defined only on the object 0,

def zerofilter = eq0 → id; $\overline{\perp}$

results from the application of the conditional combining form to the three (strict) functions eq0, id , and $\overline{\perp}$ and is itself a strict function, *i.e.*

zerofilter:$\perp$ = $\perp$

But conditional is not a strict *functional*, *i.e.* conditional applied to three functions, one of which is $\overline{\perp}$, doesn't necessarily produce $\overline{\perp}$, as the definition of zerofilter shows.

To illustrate further these distinctions, consider the following example from [Ca 72]. Let

(1) $f = Ef$ where $Ef = \text{eq0} \rightarrow \overline{0}; \times \circ [f \circ s , f \circ a]$

Suppose we relax the condition that $\times$ be strict and choose for $\times$ the *monotonic* extension:

$$\times :< 0, \perp > \; = \; 0 \; = \; \times :< \perp, 0 >$$

Then the least weak fixed point f_w of E is

$$\lim f_i \text{ where } f_i = E^i(\overline{\perp})$$

Now $f_0 = \overline{\perp}$

$$\begin{aligned} f_1 &= \text{eq0} \rightarrow \overline{0}; \times \circ [\overline{\perp} \circ s , \overline{\perp} \circ a] \\ &= \text{eq0} \rightarrow \overline{0}; \overline{\perp} \end{aligned}$$

$$\begin{aligned} f_2 &= \text{eq0} \rightarrow \overline{0}; \times \circ [(\text{eq0} \rightarrow \overline{0}; \overline{\perp}) \circ s , (\text{eq0} \rightarrow \overline{0}; \overline{\perp}) \circ a] \\ &= \text{eq0} \rightarrow \overline{0}; \times \circ [(\text{eq1} \rightarrow \overline{0}; \overline{\perp}) , (\text{eq-1} \rightarrow \overline{0}; \overline{\perp})] \\ &= \text{eq0} \rightarrow \overline{0}; \text{eq1 } \textit{or } \text{eq-1} \rightarrow \overline{0}; \overline{\perp} \\ &= \leq 1 \circ \text{abs} \rightarrow \overline{0}; \overline{\perp} \quad \text{(where abs is the absolute value primitive)} \end{aligned}$$

and, in general,

$$f_{i+1} = \leq i \circ \text{abs} \rightarrow \overline{0}; \overline{\perp}$$

so that the least weak fixed point is

$$f_w = \overline{0} \; .$$

However, computation according to μ, *i.e.* using a leftmost innermost rule of computation (LI), yields:

$$f_{LI} = \text{ge0} \rightarrow \overline{0}; \overline{\perp} \; .$$

But f_{LI} is not even a fixed point of (1) since

$$\begin{aligned} E(f_{LI}) &= \text{eq0} \rightarrow \overline{0}; \times \circ [f_{LI} \circ s , f_{LI} \circ a] \\ &= \text{eq0} \rightarrow \overline{0}; \times \circ [(\text{ge1} \rightarrow \overline{0}; \overline{\perp}) , (\text{ge-1} \rightarrow \overline{0}; \overline{\perp})] \\ &= \text{eq0} \rightarrow \overline{0}; \text{ge-1} \rightarrow \overline{0}; \overline{\perp} \; , \\ &= \text{ge-1} \rightarrow \overline{0}; \overline{\perp} \; . \end{aligned}$$

Note that if $\times$ is strict, *i.e.* if

$$\times :< x, \perp > \; = \; \perp \; = \; \times :< \perp, x >$$

for all x, including 0, then the least fixed point is

$$f_s = f_w = \text{eq0} \rightarrow \overline{0}; \overline{\perp} \; ,$$

which is the function computed by all of the computation rules mentioned above.

In summary, this simplification of the fixed point semantics is the result of requiring that all functions and the sequence constructor be strict and of treating conditional as a combining form or functional rather than a function. It is interesting to note that in his development of recursive functions Kleene also restricts primitives to strict functions, but he treats conditional as a separate, distinguished function instead of a functional [Kl 52].

It might be interesting to investigate a functional programming system in which constant functions were taken to be non-strict, *i.e.*

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

even when $y = \perp$, to see what semantic complications would be introduced by this seemingly harmless, non-strict extension.

To facilitate reasoning about programs within the language of functional programs, we now proceed to extend the language by adding some functions that will not be realizable in any actual programming system; in fact, some of these functions will not even be computable. In doing so we are violating, in the strict sense, our claim of not needing to use some auxiliary domain of discourse in order to be able to reason about functional programs. However, we feel that while these extensions can't be realized in an actual, implemented language, they nonetheless are in the functional style, and their inclusion does not destroy the advantages discussed in the previous section.

The first extension we consider is the "infinite conditional." In the previous section we wrote expressions such as

(2) $\quad p_0 \rightarrow q_0; \ldots ; p_i \rightarrow q_i; \ldots$

where there was some process for determining an arbitrary predicate/consequent pair $p_i \rightarrow q_i$. The meaning of (2) can be taken to be the limit of a chain of partial functions as follows:

Definition If $f_i = p_0 \rightarrow q_0; \ldots ; p_i \rightarrow q_i; \bar{\perp}$,
then $p_0 \rightarrow q_0; \ldots ; p_i \rightarrow q_i ; \ldots = \lim f_i$.

Using this definition, the algebraic laws for finite conditionals can be extended to such infinite conditionals, since the laws are just expressions of the mutually distributive properties of the combining forms, and each of the combining forms is monotonic and continuous. For example, the law

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

can be extended to infinite conditionals as follows. Taking f_i as above and $Hf = f \circ h$, then since H is continuous, we know that $H(\lim f_i) = \lim Hf_i$, *i.e.*

(3) $\quad (p_0 \rightarrow q_0; \ldots ; p_i \rightarrow q_i; \ldots) \circ h = p_0 \circ h \rightarrow q_0 \circ h; \ldots ; p_i \circ h \rightarrow q_i \circ h; \ldots$.

A similar argument exhibits the validity of extending the other laws involving conditionals to include infinite conditional expressions.

The next extension we consider is the strong equality predicate over functions, "=". Since this is noncomputable, it can never be realized in any actual implementation, but it is a natural extension to make to the language. In fact it is the main connective of the algebraic laws themselves. To define it we first define a total, non-strict extension to the eq predicate:

Definition $x = y$ is true if the object x is the same as the object y and false otherwise.

For example, $\perp = \perp$ and $+:< 1, T > = \perp$ are both true.

We then extend the definition of $=$ to functions by writing:

Definition $f = g$ if $f:x = g:x$ for all objects x.

In the previous section we have seen that it is useful to be able to express the equality of two functions on some restricted domain. To achieve this we first consider the total, non-strict primitive $=>$ defined on pairs of objects by:

Definition $x => y$ is false if $x = T$ and $y \neq T$ and is true otherwise.

Note that the "truth table" for $=>$ under this definition is (using v to represent any object other than T or F):

x \ y	T	F	v
T	T	F	F
F	T	T	T
v	T	T	T

which differs from the approaches to three-valued logic taken by Kleene [Kl 52] and McCarthy [Mc 63].

As with $=$, this definition is extended to functions by:

Definition $f => g$ if $f:x => g:x$ for all objects x.

We will find it useful to define two other extended primitives:

Definition defined:x is true if $x \neq \perp$ and is false otherwise.

Definition boolean:x is true if $x = T$ or $x = F$ and false otherwise.

As an example of the use of these extended primitives, it is convenient to express the fact "on objects for which the function b is defined, the function $1 \circ [a,b]$ is equal to the function a" by the statement

(4) $\quad \text{defined} \circ b => 1 \circ [a,b] = a$

Note that any statement such as (4) is understood to have an implicit universal quantifier according to the conventions extending $=>$ and $=$ to functions, and that the application of an expression such as $f = g$ to an object x is understood to denote $f:x = g:x$. That is, (4) is to be interpreted as:

$$\forall x \{ \text{defined} \circ b:x => (1 \circ [a.b]:x = a:x) \}$$

Similarly the equation

$$f=g \;=>\; Ef=Eg$$

asserts that

$$\forall x \{(f{:}x = g{:}x) \Rightarrow ((Ef){:}x = (Eg){:}x)\}$$

not that

$$\forall x\{f{:}x = g{:}x\} \Rightarrow \forall x\{(Ef){:}x = (Eg){:}x\} \ .$$

In the next section we will use these extended primitives to describe the development of an expansion theorem for a class of nonlinear recursive definitions.

Section 4. An Expansion Theorem for Overrun-tolerant Forms

In the previous sections we have seen that the algebra provides a convenient mechanism for reasoning about programs. However, to be able to use that mechanism to reason about a program defined by a functional equation requires either that we proceed indirectly by reasoning about the defining equation itself or that we produce some concrete representation of the program in the (extended) domain of functional programs to which we can then apply the laws and theorems directly. Recall that for the purposes of this paper we have chosen to restrict our attention to the latter approach. Now Backus [Ba 79] has identified a large class of forms, the linear forms, for which such a concrete object is always obtainable. However, for nonlinear forms we cannot use the algebraic laws until we find an appropriate expansion.

Unfortunately, many commonly occurring recursive definitions use nonlinear combining forms to which the linear expansion theorem is not applicable. For example, the Fibonacci number program can be defined as the solution to

(1) $\quad \text{fib} = \leq 1 \rightarrow \text{id}; + \circ [\text{fib} \circ \text{s}, \text{fib} \circ \text{s}^2]$

where s denotes subtraction by 1. The general form of the functional used in the else clause,

(2) $\quad Hf = \text{h} \circ [f \circ \text{i}, f \circ \text{j}],$

is not always linear, *i.e.* it is not the case that for all h, i, and j there exists an $H_1 f$ such that

$$H(\text{a} \rightarrow \text{b}; \text{c}) = H_1\text{a} \rightarrow H\text{b}; H\text{c} \quad \text{for all a, b, c.}$$

For example, even in the simple case $Hf = \text{id} \circ [f \circ \text{id}, f \circ \text{a}]$ no such H_1 can exist. Just consider

$$H(\text{even} \rightarrow \bar{0}; \bar{1}) = \text{even} \rightarrow [\bar{0},\bar{1}]; [\bar{1},\bar{0}].$$

Now clearly no H_1 can be found that will always make

$$\text{even} \rightarrow [\bar{0},\bar{1}]; [\bar{1}.\bar{0}] \quad \text{equal to} \quad H_1(\text{even}) \rightarrow [\bar{0}.\bar{0}]; [\bar{1},\bar{1}] \ .$$

In fact, on the domain of integers the two functions are *never* equal.

The algebraic approach will not be very useful if we have to develop a new expansion for each equation involving a nonlinear form. Therefore we are interested in discovering extensions of the class of linear forms for which we can prove expansion theorems. In the remainder of this section we will develop and prove an expansion theorem for a particular class of forms, which we call *overrun-tolerant.*

The notion of overrun-tolerant is based on the definition of bilinear forms in [Ba 79]:

Definition: $G(f,g)$ is *bilinear* if there exist $G_1 f$ and $G_2 g$ such that for all a, b and c:

(3) $\quad G(a \rightarrow b;c\ ,\ g) \;=\; G_1 a \rightarrow G(b,g);\ G(c,g)$

(4) $\quad G(f,\ a \rightarrow b;\ c) \;=\; G_2 a \rightarrow G(f,b);\ G(f,c).$

G_1 and G_2 are called "predicate transformers". In the sequel, if G is bilinear, then G_1 and G_2 will always denote a corresponding pair of predicate transformers.

Intuitively, $G(f,g)$ is bilinear if it is linear in each of its arguments and if the associated predicate transformers are independent of g and f, respectively. For example, the two-variable form

$$G(f,g) \;=\; h \circ [f \circ i,\ g \circ j]$$

is bilinear with $G_1 f = f \circ i$ and $G_2 g = g \circ j$.

The form we are interested in, (2), is the form in one variable obtained by identifying the two arguments of G, *i.e.*

(5) $\quad Hf \;=\; Gff \;=\; h \circ [f \circ i,\ f \circ j]$

Suppose, now, that we are given the equation

(6) $\quad f \;=\; Ef \qquad$ where $\quad Ef \;=\; p \rightarrow q;\ Hf$

We will derive the first three functions in the chain of approximations to f:

$$\overline{\perp} \sqsubseteq E(\overline{\perp}) \sqsubseteq E^2(\overline{\perp}) \sqsubseteq \ \ldots\ \sqsubseteq E^n(\overline{\perp}) \sqsubseteq \ \ldots\ .$$

$$f_0 \;=\; \overline{\perp}$$

$$f_1 \;=\; E(\overline{\perp}) \;=\; p \rightarrow q;\ H(\overline{\perp}) \;=\; p \rightarrow q;\ \overline{\perp}$$

$$\begin{aligned} f_2 \;&=\; E^2(\overline{\perp}) \\ &=\; E(p \rightarrow q;\ \overline{\perp}) \\ &=\; p \rightarrow q;\ H(p \rightarrow q;\ \overline{\perp}) \\ &=\; p \rightarrow q;\ G(p \rightarrow q;\ \overline{\perp}\ ,\ p \rightarrow q;\ \overline{\perp}) \end{aligned}$$

Applying (3) and (4) and simplifying yields

(7) $\quad f_2 \;=\; p \rightarrow q;\ G_1 p \rightarrow (G_2 p \rightarrow Hq;\ \overline{\perp});\ \overline{\perp}$

At this point we see that the f_n are not going to "grow" in a nice, linear fashion. In particular, the second predicate/consequent pair of f_2,

$$G_1p \rightarrow (G_2p \rightarrow Hq; \overline{\perp}) ,$$

is not yet completely specified. It will become increasingly more defined in successive approximations, the next of which is shown without the intermediate steps in its derivation:

$$(8)\quad f_3 = p \rightarrow q; G_1p \rightarrow (G_2p \rightarrow Hq; G_2G_1p \rightarrow (G_2^2p \rightarrow G(q,Hq); \overline{\perp}); \overline{\perp}); \\ G_1^2p \rightarrow (G_1G_2p \rightarrow (G_2p \rightarrow G(Hq,q); G_2G_1p \rightarrow (G_2^2p \rightarrow H^2q; \overline{\perp}); \overline{\perp}); \overline{\perp}); \overline{\perp}$$

Notice that f_3 is obtained from f_2 by replacing the $\overline{\perp}$s with partial functions, thus making f_3 more defined than f_2. For the particular case we are considering, the second term in the sequence of approximating functions goes from

$$\overline{\perp} \quad (\text{in } f_1) \text{ to}$$
$$G_1p \rightarrow (G_2p \rightarrow Hq; \overline{\perp}) \quad (\text{in } f_2) \text{ to}$$
$$G_1p \rightarrow (G_2p \rightarrow Hq; G_2G_1p \rightarrow (G_2^2p \rightarrow G(q,Hq); \overline{\perp}); \overline{\perp}) \quad (\text{in } f_3) .$$

This term will become increasingly more defined in successive approximations but will be *completely* defined only in the limit of the chain. Thus, unlike the situation for a linear form in which the nth term of the solution is completely specified by the nth approximation, the nth term in the solution to (6) is itself a limit of a chain of partially defined functions. This severely complicates any attempt to produce a closed form expression for the nth term.

We will see that imposing some additional conditions on p, q and H in the equation $f = p \rightarrow q; Hf$ will allow us to prove an expansion theorem very similar to the theorem for linear forms. Consider again the Fibonacci program defined by

$$(9)\quad fib = \leq 1 \rightarrow id; + \circ [fib \circ s, fib \circ s^2].$$

We can write (9) as

$$(10)\quad fib = \leq 1 \rightarrow id; + \circ [fib \circ i, fib \circ j]$$

where $i = \leq 1 \rightarrow \bar{0}; s$ and
$j = \leq 1 \rightarrow id; s^2$,

since $\leq 1 \rightarrow id; + \circ [fib \circ i, fib \circ j]$

$$= \leq 1 \rightarrow id; \leq 1 \rightarrow + \circ [fib \circ \bar{0}, fib \circ id]; + \circ [fib \circ s, fib \circ s^2]$$

$$= \leq 1 \rightarrow id; + \circ [fib \circ s, fib \circ s^2] .$$

Now (10) has the form $f = p \rightarrow q; Hf$ where

$$p = \leq 1,$$
$$q = id,$$
$$Hf = + \circ [f \circ i, f \circ j] .$$

Moreover, p, q, and H satisfy the two conditions

$$(c1)\quad p \Rightarrow q = Hq, \quad \text{and}$$

$$(c2)\quad p \Rightarrow \& \circ [p \circ i, p \circ j].$$

To verify (c1), if p(*i.e.* ≤ 1) is true, then

$$\begin{aligned} Hq &= +\circ[q\circ i,\ q\circ j] \\ &= \leq 1 \rightarrow +\circ[q\circ \bar{0},\ q\circ id];\ +\circ[q\circ s,\ q\circ s^2] \\ &= +\circ[q\circ \bar{0},\ q\circ id] \quad \text{(since } \leq 1 \text{ is true)} \\ &= +\circ[id\circ \bar{0},\ id\circ id] \quad \text{(since } q = id) \\ &= id \end{aligned}$$

Similarly, to verify (c2) we have that

$$\begin{aligned} \&\circ[\leq 1\circ i, \leq 1\circ j\,] &= \leq 1 \rightarrow \&\circ[\leq 1\circ \bar{0},\ \leq 1\circ id];\ \&\circ[\leq 1\circ s,\ \leq 1\circ s^2] \\ &= \&\circ[\leq 1\circ \bar{0},\ \leq 1\circ id] \quad \text{(since } \leq 1 \text{ is true)} \\ &= \leq 1\circ id \quad \text{(since } \leq 1\circ \bar{0} = \bar{T}) \\ &= \bar{T} \quad \text{(since } \leq 1 \text{ is true)} \end{aligned}$$

We will use these two conditions to motivate the following expansion theorem:

Theorem (*Overrun-tolerant expansion*) Given the functional equation $f = p \rightarrow q;\ Hf$ and a predicate transformer H_1 such that

(a) there exists a bilinear and strict form
Gfg such that $H_1 f = G_1 f\ \&\ G_2 f$ and $Hf = Gff$,
(b) $p \Rightarrow (q = Hq)$, and
(c) $H_1{}^n p \Rightarrow H_1{}^{n+1} p$ for $n \geq 0$,

then $f = p \rightarrow q;\ ...;\ H_1{}^n p \rightarrow H^n q;\ ...$.

Intuitively, condition (b) means that we can safely "overapply" the recursive expression; *i.e.* if p is true then the function can be correctly computed by applying either q or Hq. Condition (c) guarantees that once some $H_1{}^n p$ becomes true, all successive $H_1{}^m p$, $m > n$, will be true as well, so that overapplying the recursive definition will not result in an infinite computation.

Our proof of the theorem will use the following four technical lemmas. In their presentation we will use the abbreviation $f\ \&\ g$ as shorthand for $\&\circ[f,g]$ and $\neg f$ for $not\circ f$.

Lemma A. If Gfg is bilinear and strict,
then $G(p \rightarrow q;\bar{\perp}\ ,\ p \rightarrow q;\bar{\perp}) = G_1 p\ \&\ G_2 p \rightarrow Gqq;\ \bar{\perp}$.

Proof: $G(p \rightarrow q;\bar{\perp}\ ,\ p \rightarrow q;\bar{\perp})$

$$\begin{aligned} &= G_1 p\ \&\ G_2 p \rightarrow Gqq; \\ &\quad\ G_1 p\ \&\ \neg G_2 p \rightarrow G q\bar{\perp}; \\ &\quad\ \neg G_1 p\ \&\ G_2 p \rightarrow G\bar{\perp} q; \\ &\quad\ \neg G_1 p\ \&\ \neg G_2 p \rightarrow G\bar{\perp}\bar{\perp} \qquad \text{since } G \text{ is bilinear} \\ &= G_1 p\ \&\ G_2 p \rightarrow Gqq;\ \bar{\perp} \qquad \text{since } G \text{ is strict.} \qquad \text{QED} \end{aligned}$$

Lemma B. If Ef is linear with predicate transformer E_1, and if $a \Rightarrow (b = c)$,
then $E_1 a \Rightarrow (Eb = Ec)$.

Proof: Since $a \Rightarrow (b = c)$, we know that $a \rightarrow b;c\ =\ a \rightarrow c;c$
Therefore
$E(a \rightarrow b;\ c)\ =\ E(a \rightarrow c;\ c)$ and since E is linear,
$E_1 a \rightarrow Eb;\ Ec\ =\ E_1 a \rightarrow Ec;\ Ec$.

So if E_1a is true for an object, Eb applied to that object is the same as Ec applied to the object. QED

Lemma C. If Gfg is bilinear and $a \Rightarrow (b = c)$, then $G(a \rightarrow b;c\ ,\ a \rightarrow b;c) = G_1a\ \&\ G_2a \rightarrow Gbb;\ Gcc$.

Proof: $G(a \rightarrow b;c\ ,\ a \rightarrow b;c)$

(11a) $= G_1a\ \&\ G_2a \rightarrow Gbb;$
(11b) $G_1a\ \&\ \neg G_2a \rightarrow Gbc;$
(11c) $\neg G_1a\ \&\ G_2a \rightarrow Gcb;$
(11d) $\neg G_1a\ \&\ \neg G_2a \rightarrow Gcc$ since G is bilinear.

Now Gfc is linear in f with predicate transformer G_1, so, by Lemma B,

$$G_1a \Rightarrow (Gbc = Gcc).$$

Therefore Gbc can be replaced by Gcc in (11b). Similarly, since Gcg is linear in g with predicate transformer G_2, Gcb can be replaced by Gcc in (11c). QED

Lemma D. If Gfg is bilinear and $a \Rightarrow (b = c)$, then $G_1a\ \&\ G_2a \Rightarrow (Gbb = Gcc)$.

Proof: Since $a \Rightarrow (b = c)$, $a \rightarrow b;c = a \rightarrow c;c$. Therefore

(12) $G(a \rightarrow b;c,\ a \rightarrow b;c) = G(a \rightarrow c;c,\ a \rightarrow c;c)$.

Since $a \Rightarrow (b = c)$, we have, by Lemma C,

(13) $G(a \rightarrow b;c\ ,\ a \rightarrow b;c) = G_1a\ \&\ G_2a \rightarrow Gbb;\ Gcc$.

Similarly, since $a \Rightarrow (c = c)$, Lemma C yields

(14) $G(a \rightarrow c;c\ ,\ a \rightarrow c;c) = G_1a\ \&\ G_2a \rightarrow Gcc;\ Gcc$.

Therefore, substituting (13) and (14) in (12) yields

$$G_1a\ \&\ G_2a \rightarrow Gbb;\ Gcc = G_1a\ \&\ G_2a \rightarrow Gcc;\ Gcc,$$

so that $G_1a\ \&\ G_2a \Rightarrow (Gbb = Gcc)$. QED

Corollary. If $Hf = Gff$ where G is bilinear, $H_1a = G_1a\ \&\ G_2a$, and $p \Rightarrow (q = Hq)$, then $H_1{}^np \Rightarrow (H^nq = H^{n+1}q)$.

Proof: By n-fold application of Lemma D. QED

Our proof of the overrun tolerant expansion theorem now proceeds as follows:

Proof: We know from Section 3 that $f = \lim E^i(\overline{1})$ where $Ef = p \rightarrow q;\ Hf$. Therefore, if we can show that for all $i \geq 1$ the ith approximation in this chain, f_i, is given by

(15) $f_i = p \rightarrow q;\ ...;\ H_1{}^{i-1}p \rightarrow H^{i-1}q;\ \overline{1}$,

the theorem will be proved. We will prove (15) by induction on i, using the following inductive hypotheses:

$$\text{(A) } H^n f_i = H_1{}^n p \to H^n q; \ldots; H_1{}^{n+i-1} p \to H^{n+i-1} q; \overline{\perp} \ ,$$

and $\text{(B) } H_1{}^n p \Rightarrow (H^n q = H^{n+1} f_i)$.

Basis: i = 1. To establish (A) we see that

$H^n f_1 = H^n(p \to q; \overline{\perp})$ by definition of f_1

$= H_1{}^n p \to H^n q; \overline{\perp}$ by n applications of Lemma A.

To establish (B), if we assume $H_1{}^n p$, then

$H^{n+1} f_1 = H_1{}^{n+1} p \to H^{n+1} q; \overline{\perp}$ by (A) for i = 1,

$= H^{n+1} q$ by hypothesis (c), since $H_1{}^n p$ is true,

$= H^n q$ by Corollary to Lemma D.

Induction step: Assume (A) and (B) true for values $\leq$ i. Then to establish (A) for i+1,

$H^n f_{i+1} = H^n(p \to q; H f_i)$ by definition of f_{i+1},

$= H_1{}^n p \to H^n q; H^{n+1} f_i,$ by (B) and n applications of Lemma C,

$= H_1{}^n p \to H^n q; H_1{}^{n+1} p \to H^{n+1} q; \ldots ; H_1{}^{n+1+i-1} p \to H^{n+1+i-1} q; \overline{\perp}$
by using (A) to rewrite $H^{n+1} f_i$,

$= H_1{}^n p \to H^n q; \ldots; H_1{}^{n+(i+1)-1} p \to H^{n+(i+1)-1} q; \overline{\perp}$
by rewriting the exponents.

Similarly, to establish (B), if we assume $H_1{}^n p$, then

$H^{n+1} f_{i+1} = H^{n+1}(p \to q; H_i)$ by definition of f_{i+1} ,

$= H_1{}^{n+1} p \to H^{n+1} q; \ldots$ as above,

$= H^{n+1} q$ by $H_1{}^n p$ and hypothesis (c),

$= H^n q$ by Corollary to Lemma D.

QED

To illustrate the use of this expansion theorem, consider the following function, called "fusc" for ob*fusc*ate, from [Di 78], where we use d to denote the primitive "divide by 2":

(16) **def** fusc = $\leq 1 \to$ id; even $\to$ fusc $\circ$ d; + $\circ$ [fusc $\circ$ d $\circ$ s, fusc $\circ$ d $\circ$ a]

We can use the overrun-tolerant expansion theorem to obtain a representation for this function as follows. We can rewrite (16) as

(17) **def** fusc = $\leq 1 \to$ id; + $\circ$ [fusc $\circ$ i, fusc $\circ$ j]

where i and j are defined as:

$$\textbf{def } i = \text{even} \rightarrow \bar{0};\ d \circ s$$
$$\textbf{def } j = \text{even} \rightarrow d;\ d \circ a$$

since $+ \circ [\text{fusc} \circ i, \text{fusc} \circ j] = + \circ [\text{fusc} \circ (\text{even} \rightarrow \bar{0};\ d \circ s), \text{fusc} \circ (\text{even} \rightarrow d;\ d \circ a)]$

$$= \text{even} \rightarrow + \circ [\text{fusc} \circ \bar{0}, \text{fusc} \circ d];\ + \circ [\text{fusc} \circ d \circ s, \text{fusc} \circ d \circ a]$$

$$= \text{even} \rightarrow \text{fusc} \circ d;\ + \circ [\text{fusc} \circ d \circ s, \text{fusc} \circ d \circ a]$$

Now if we let

$$Gfg = + \circ [f \circ i, g \circ j] ,$$
$$Hf = Gff \text{ , and}$$
$$H_1 p = \& \circ [G_1 p, G_2 p] = \& \circ [p \circ i, p \circ j] ,$$

then the hypotheses of the theorem are satisfied; *i.e.*

a) Gfg is bilinear,
b) $\leq 1 \Rightarrow H(\text{id}) = \text{id}$, and
c) $H_1{}^n(\leq 1) \Rightarrow H_1{}^{n+1}(\leq 1)$

so that

$$\text{fusc} = \leq 1 \rightarrow \text{id};\ \ldots\ ;\ H_1{}^n(\leq 1) \rightarrow H^n(\text{id});\ \ldots \ .$$

Now $H_1(\leq 1) = \& \circ [\leq 1 \circ i, \leq 1 \circ j]$

$$= \text{even} \rightarrow \& \circ [\overline{T}, \leq 1 \circ d];\ \& \circ [\leq 1 \circ d \circ s, \leq 1 \circ d \circ a]$$

$$= \text{even} \rightarrow \leq 2;\ \leq 1$$

$$= \leq 2$$

and in general,

$$H_1{}^n(\leq 1) = \leq 2^n \ .$$

Similarly,

$$H(\text{id}) = + \circ [i, j] ,$$

$$H^2(\text{id}) = + \circ [+ \circ [i, j] \circ i, + \circ [i, j] \circ j]$$

$$= / + \circ [ii, ji, ij, jj]$$

and if we let IJ^n stand for the construction of the 2^n functions,

$$IJ^n = [i...i, ji...i, \ldots , ij...j, j...j]$$

then in general,

$$H^n(\text{id}) = / + \circ IJ^n \ .$$

From this general term we can observe that if $H_1^n(\leq 1)$ is true, *i.e.* if $\leq 2^n$, then $H^n(id)$ will consist of the sum of 2^n terms each of which is either 0 or 1. By studying the function IJ^n more closely, we could discover that for all n and m, there is an integer k such that $IJ^n{:}m$ will be a sequence of 2^n numbers, each of which is either 0, k, or $k-1$, and thereby discover the iterative algorithm of Dijkstra [Di 78].

Note that this analysis is not an automatic byproduct of the expansion theorem. The expansion theorem merely gives us a convenient, nonrecursive expression for the general term of the infinite conditional. How we use that general term to reason about the function is still left to our creativity and imagination. But it does allow us to apply that creativity within the language of functional programs, without the necessity for reasoning about some auxiliary domain. In general, the usefulness of an expansion will be limited by the complexity of the forms H_1 and H. If H_1^n and H^n are very complex, it will do little good to know that the general term is $H_1^n p \rightarrow H^n q$.

Section 5. Suggestions for Further Work

As we observed in Section 2, when a function is defined by the recursive equation, $f = p \rightarrow q; Hf$, linearity of Hf is a stronger condition than is actually needed to prove the linear expansion theorem. That is, we needn't require that H distribute over all conditionals $a \rightarrow b;c$, but only that it distribute over the approximations to the least fixed point of $f = Ef$, *i.e.* $E^i(\overline{\perp})$ for $i \geq 1$.

At the same time we can make a further generalization. Just as linearity modified the notion of distributivity by allowing a predicate transformer H_1, we can further relax the notion by allowing a "consequent transformer" as well. This suggests the following

Definition Hf is *p,q-distributive* if there exist H_1 and H_2 such that for all $n \geq 0$,

(1) $H(p \rightarrow q; \ldots ; H_1^n p \rightarrow H_2^n q; \overline{\perp}) = H_1 p \rightarrow H_2 q; \ldots ; H_1^{n+1} p \rightarrow H_2^{n+1} q; H\overline{\perp}$.

Given this definition we have immediately the following

Theorem If $f = p \rightarrow q; Hf$, and Hf is p,q-distributive and $\overline{\perp}$-preserving,

then $f = p \rightarrow q; \ldots ; H_1^n p \rightarrow H_2^n q; \ldots$.

Proof: Each $E^i(\overline{\perp})$ can be rewritten as

$$p \rightarrow q; \ldots ; H_1^{i-1} p \rightarrow H_2^{i-1} q; \overline{\perp} .$$

For example, $E^3(\overline{\perp}) = p \rightarrow q; H(p \rightarrow q; H(p \rightarrow q; H\overline{\perp})))$

$= p \rightarrow q; H(p \rightarrow q; H(p \rightarrow q; \overline{\perp}))$ since $H\overline{\perp} = \overline{\perp}$,

$= p \rightarrow q; H(p \rightarrow q; H_1 p \rightarrow H_2 q; H\overline{\perp})$ since H is p,q-distributive,

$= p \rightarrow q; H(p \rightarrow q; H_1 p \rightarrow H_2 q; \overline{\perp})$ since $H\overline{\perp} = \overline{\perp}$,

$= p \rightarrow q; H_1 p \rightarrow H_2 q; H_1^2 p \rightarrow H_2^2 q; \overline{\perp}$ since H is p,q-distributive.

Then, since $f = \lim E^i(\overline{\perp})$, we have that $f = p \to q; \ldots ; H_1{}^n p \to H_2{}^n q; \ldots$. QED

To illustrate the use of this expansion theorem, consider the following example of Morris [Mo 71] where h is some arbitrary function:

(2) **def** $f = p \to id; f^2 \circ h$

Now $Hf = f^2 \circ h$ is p,id-distributive with

$$H_1 f = H_2 f = f \circ h,$$

since for all n,

$H(p \to q; \ldots ; H_1{}^n p \to H_2{}^n q; \overline{\perp})$

$= H(p \to id; p \circ h \to h; \ldots ; p \circ h^n \to h^n; \overline{\perp})$ by def. of q, H_1, and H_2

$= (p \to id; \ldots ; p \circ h^n \to h^n; \overline{\perp}) \circ (p \to id; \ldots ; p \circ h^n \to h^n; \overline{\perp}) \circ h$ by definition of H

$= (p \to id; \ldots ; p \circ h^n \to h^n; \overline{\perp}) \circ (p \circ h \to h; \ldots ; p \circ h^{n+1} \to h^{n+1}; \overline{\perp})$ by (A2)

$= p \circ h \to (p \circ h \to h; \ldots ; p \circ h^{n+1} \to h^{n+1}; \overline{\perp}); \ldots ; p \circ h^{n+1} \to (p \circ h^{n+1} \to h^{n+1}; \ldots); \overline{\perp}$ by (A1)

$= p \circ h \to h; \ldots ; p \circ h^{n+1} \to h^{n+1}; \overline{\perp}$ by n+1 applications of (A8)

$= H_1 p \to H_2 q; \ldots ; H_1{}^{n+1} p \to H_2{}^{n+1} q; \overline{\perp}$.

Therefore, the solution to (2) may be written as

$$f = p \to id; p \circ h \to h; \ldots ; p \circ h^n \to h^n; \ldots .$$

In [Ba 78], Backus arrived at this expansion by *ad hoc* methods, and he used a case analysis at the object level to show that f is idempotent. However, using this expansion, we can show that $f = f^2$ directly:

$f^2 = (p \to id; \ldots ; p \circ h^n \to h^n; \ldots) \circ (p \to id; \ldots ; p \circ h^n \to h^n; \ldots)$

$= p \to (p \to id; \ldots ; p \circ h^n \to h^n; \ldots) ; \ldots ; p \circ h^n \to (p \circ h^n \to h^n; \ldots) ; \ldots$ by (A1)

$= p \to id; \ldots ; p \circ h^n \to h^n; \ldots$ by (A8)

$= f$

The proof of the above theorem required only that H distribute over $E^i(\overline{\perp})$ when p is false. Therefore we can weaken the condition for p,q-distributivity to read: for all $n \geq 0$,

(3) $\neg p \Rightarrow H(p \to q; \ldots ; H_1{}^n p \to H_2{}^n q; \overline{\perp}) = H_1 p \to H_2 q; \ldots ; H_1{}^{n+1} p \to H_2{}^{n+1} q; H\overline{\perp}$

and still prove the expansion theorem.

It would be interesting to characterize the class of triples p, q, H such that H is p,q-distributive. Some very nonlinear forms can be expanded using this technique. For example, McCarthy's "91" function [Ma 73] given by the equation

(4) **def** $f = >100 \rightarrow s10; f \circ f \circ a11$

where $s10 = -\circ[id, \overline{10}]$ and $a11 = +\circ[id, \overline{11}]$ can be shown to be (>100, s10)-distributive with

$$H_1 p = p \circ \overline{91} \rightarrow p \circ a11; p \circ a1 \quad \text{and}$$

$$H_2 q = \overline{91} \quad .$$

The complicated nature of the predicate transformer H_1 reflects the peculiar nature of the successive approximations to this function. Each successive approximation is defined for one more argument value until the twelfth approximation, at which point each one is defined for eleven more values than its predecessor; *i.e.*

$$\begin{aligned} &\vdots \\ f_{10} &= >100 \rightarrow s10; >91 \rightarrow \overline{91}; \overline{\perp} \\ f_{11} &= >100 \rightarrow s10; >90 \rightarrow \overline{91}; \overline{\perp} \\ f_{12} &= >100 \rightarrow s10; >79 \rightarrow \overline{91}; \overline{\perp} \\ f_{13} &= >100 \rightarrow s10; >68 \rightarrow \overline{91}; \overline{\perp} \\ &\vdots \end{aligned}$$

Thus the solution to (4) can be written

$$\begin{aligned} f &= >100 \rightarrow s10; \ldots ; H_1^n(>100) \rightarrow H_2^n(s10); \ldots \\ &= >100 \rightarrow s10; \ldots ; H_1^n(>100) \rightarrow \overline{91}: \ldots \\ &= >100 \rightarrow s10; \overline{91} \end{aligned}$$

since $\neg >100$ implies that for some value of i, $H_1^i(>100)$ will be true and that for all values of i, $H_1^n(>100)$ will be defined. Thus the (>100, s10)-distributivity of $Hf = f^2 \circ a11$ enables us to write the solution to (4) as a nonrecursive conditional, which we can use to reason about the function.

Of course, as we noted in the last section, the existence of an expansion in terms of H_1 and H_2 does not guarantee that we will be able to understand (*i.e.* to reason about) that function. First, we must discover the particular forms H_1 and H_2, and for this we will need some general, constructive methods that work for significant classes of recursive definitions. Second, the forms so produced will have to be manageable. It will do little good to find forms H_1 and H_2 if they are so complex as to make H_1^n and H_2^n inscrutable. But, these considerations notwithstanding, it would be interesting to characterize the functions p and q and the forms H for which there exist H_1 and H_2 such that (3) holds.

Acknowledgments It is a pleasure to acknowledge my indebtedness to John Backus, the *sine qua non* of this functional programming style. I have benefited greatly from countless hours of stimulating discussions with him. I am also grateful to David Gries, Steven Muchnick, and

Donald Stanat, each of whom read earlier drafts of this paper and made many helpful suggestions.

References

[Ba 78] Backus, J.W. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." *CACM* August, 1978

[Ba 79] Backus, J.W. "On extending the concept of program and solving linear functional equations." IBM Report, August, 1979

[Ca 72] Cadiou, J.M. "Recursive definitions of partial functions and their computations." Stanford Artificial Intelligence Memo 163, Computer Science Department, Stanford University, March, 1972

[Di 76] Dijkstra, E.W. *A Discipline of Programming*, Prentice Hall, 1976

[Di 78] Dijkstra, E.W. "Lecture notes on program development." International Summer School on Program Construction, Marktoberdorf, Germany , August, 1978.

[Kl 52] Kleene, S.C. *Introduction to Metamathematics*, Van Nostrand, New York, 1952

[Ma 73] Manna, Z., Ness, S., and Vuillemin, J. "Inductive methods for proving properties of programs." *CACM* August, 1973

[Mc 63] McCarthy, J. "Predicate Calculus with Undefined as a Truth Value." Stanford Artificial Intelligence Memo 1, Computer Science Department, Stanford University, March, 1963

[Mo 71] Morris, J.H. "Another recursion induction principle." *CACM* May, 1971