NOTES ON THE FP STYLE OF FUNCTIONAL PROGRAMMING

John H. Williams

IBM Research Laboratory
San Jose, California

Section 1. Introduction

In these notes we consider three facets of the functional programming style introduced by John Backus in his 1977 Turing Award Lecture [Ba 78]: the particular programming methodology it encourages, the associated algebra of programs that can be used to reason about and understand FP programs, and the environment-free reduction system that can be used to realize such a style of programming. In section 2 we construct several programs in this FP style, and we point out some of the differences between it and the lambda style of functional programming. In section 3 we describe an informal but precise method for reasoning about FP programs -- the algebra of functional programs, and we apply this method to some sample programs. We will see that this algebraic approach allows us to reason about FP programs without having to map programs to some auxilliary domain of discourse. The language of program proofs will be just the language of FP programs.

Finally in section 4 we will describe the Formal Functional Programming system, FFP, of [Ba 78], and we will show how all of the notions of the informal FP style of programming can be realized in the pure reduction system FFP. In the remainder of this section we review the basic notions of functional programming and define the primitive functions and combining forms that are used in these notes. 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 these notes 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 we will be using:

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: 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.

apndl append to the left: apndl applied to a pair of objects $\langle e, y\rangle$ the second of which is a sequence, will produce a sequence whose first element is e and whose tail is y, otherwise it will produce $\perp$; apndl:<3, <4, 5>> = <3, 4, 5> ; apndl:<<1>, <3>> = <<1>, 3> ; apndl:<3, 4> = $\perp$.

apndr append to the right: apndr:<<3, 4>, 5> = <3, 4, 5>.

distl distribute from the left: distl applied to a pair of objects the second of which is a sequence will produce a sequence of pairs, otherwise it will produce $\perp$; distl:<1, <A, B>> = <<1, A>, <1, B>> ; distl:<1, <A>, <B, C>> = $\perp$; distl:<1, <>> = <>.

distr distribute from the right: ; distr:<<1, 2>, 3> = <<1, 3>, <2, 3>>.

iota the APL ι operator: iota applied to an integer n produces a sequence of n consecutive integers starting from 1; iota applied to a non-integer produces $\perp$; iota:3 = <1, 2, 3> ; iota:0 = <> ; iota:<3, 4> = $\perp$.

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.

Apply to all $\alpha f:x = \langle f:x_1, \ldots, f:x_n\rangle$ if $x = \langle x_1, \ldots, x_n\rangle$,
$\perp$ if x is not a sequence.

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

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

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

$$(/f):\langle x_1, \ldots, x_n\rangle = f:\langle x_1, (/f):\langle x_2, \ldots, x_n\rangle\rangle.$$

Note that this definition of (unsubscripted) / is consistent with 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,

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

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

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

We will see in the next section that treating Conditional as a combining form instead of as a function as is usually done in programming language definitions will profoundly affect the programming style that we will adopt.

Section 2. The FP Style of Functional Programming

Having completed our review of FP, we now consider some sample programs in this language. One of the most important characteristics of this language definition is the style of programming that it encourages. The FP style of functional programming differs from the lambda style in that the emphasis in the former is on the application of functionals (*i.e.* combining forms) to functions to produce new functions, whereas the emphasis in the latter is on the application of functions to objects to produce new objects.

For example, consider the problem of writing a program, length, whose specification is:

$$\text{length}:x = \begin{cases} n & \text{if } x = \langle x_1, \ldots, x_n\rangle, \\ \bot & \text{if } x \text{ is not a sequence.} \end{cases}$$

Programming in the lambda style we would write

1) length = λx.(**if** null(x) **then** 0 **else** 1+length(tail(x)))

The right hand side of 1) is formed by taking an object x and applying two functions to it getting the objects null(x) and tail(x), applying the function length to the object tail(x) to get the object length(tail(x)), and finally applying the function conditional to the objects null(x), 0, and 1+length(tail(x)). (Of course, the function conditional must be treated as a "special form" when it is actually applied.) Then to produce the desired function length, this object is abstracted with respect to x by applying the one function forming operation of the lambda style, $\lambda x.(...)$. (Note that even "functionals" such as maplist and mapcar are object forming operations. Given an object -- a list -- and a function, they produce an object.) We will see that the FP style uses several function forming operations -- the combining forms, and that this style exploits the potential concurrency inherent in those forms.

It is possible to transliterate the lambda style solution 1) to an FP language solution by moving the programming activity up one level; *i.e.* by beginning with functions and applying combining forms to them. Thus,

2) $\quad \text{length} \circ \text{tl}$

is the *function* that results from the application of composition to the *functions* length and tl;

3) $\quad [\bar{1}, \text{length} \circ \text{tl}]$

is the function that results from the application of construction to the functions $\bar{1}$ and 2); and so on until the combining form conditional is applied to produce the final result:

4) $\quad \textbf{def}\ \text{length} = \text{null} \rightarrow \bar{0}; + \circ [\bar{1}, \text{length} \circ \text{tl}]$

But a solution that better utilizes the available combining forms and is more in the FP style is the following nonrecursive, closed form program:

5) $\quad \textbf{def}\ \text{length} = /+ \circ \alpha\bar{1}$

The algorithm implemented by this program treats each element of the sequence as a 1 and simply adds them up -- essentially the way we would solve the problem manually. We will see that this solution has

the advantage that it allows us to reason about it directly, since 5) gives a direct representation of the program, whereas 4) gives only an indirect description of it -- an equation that the program must satisfy.

As a second example consider the problem of determining whether some particular object is a member of a sequence of objects, *i.e.*

$$\begin{aligned} \text{member:arg} = {} & \bot \quad \text{if arg is not of the form} \\ & \quad \langle x, \langle x_1, \ldots, x_n \rangle \rangle, \\ & \text{T} \quad \text{if } \exists i \text{ such that } x = x_i, \\ & \text{F} \quad \text{otherwise.} \end{aligned}$$

A recursive solution in the lambda style might look like

6) **def** member = null∘2 → $\overline{\text{F}}$; eq∘[1, 1∘2] → $\overline{\text{T}}$; member∘[1, tl∘2]

A closed form, nonrecursive alternative is

7) **def** member = /or ∘ αeq ∘ distl .

Again, we prefer this solution since it allows all of the tests for equality to be made concurrently and provides us with a direct representation rather than an indirect description.

It will be convenient to have a program un that, when given a pair of sequences, will concatenate them. *E.g.*,

un:<<A, B>, <C, D>> = <A, B, C, D>
un:<<A, B>,<>> = <A, B>
un:<<>, <>> = <> .

Exercise 1. Give a closed form definition of un in terms of the primitive functions apndl and apndr.

Although program 7) allows more concurrency than program 6), closer examination of 7) shows that while all of the tests for equality can be made together, the function /or is still unnecessarily overspecified, since the definition of / requires that the function or be inserted from the right, resulting in a right linear tree of application. For associative functions like + and or, we would prefer a combining form that allowed as much concurrency as possible. Therefore we will

define a new combining form called *tree*, which can be used as an alternative to insert:

$$\begin{aligned}(\text{tree } f):x \;=\; & x_1\ , \text{ if } x = \langle x_1 \rangle\ , \\ & f{:}\langle(\text{tree } f){:}\langle x_1,\ldots, x_m\rangle, (\text{tree } f){:}\langle x_{m+1},\ldots, x_n\rangle\rangle\ , \\ & \quad \text{if } x = \langle x_1, \ldots x_n\rangle \text{ and } m = \text{ceil}(n/2), \\ & \perp\ , \text{ if } x \text{ is not a sequence.}\end{aligned}$$

Now member can be rewritten as

def member = (tree or) ∘ αeq ∘ distl ,

and the application of the function (tree or) to the sequence of n T's and F's will take $\log(n)$ time. We will take as a model for the underlying machine architecture one that provides arbitrary concurrency, just as the usual RASP model provides an arbitrary amount of memory. Our goal will then be to write programs that exploit this concurrency as much as possible, through the use of the α and tree combining forms. Of course, the correctness of the programs is independent of the degree of concurrency provided by the underlying implementing hardware.

Consider now the problem of writing a program that when given a pair of sequences representing two sets, will compute the Cartesian product of the two sets. *E.g.*,

cart2:<<A,B>, <C,D>> = <<A,C>, <A,D>, <B,C>, <B,D>>
cart2:<<A>, <1, 2, 3>> = <<A, 1>, <A, 2>, <A, 3>>
cart2:<<>, <>> = <>

A possible program in the FP style is

def cart2 = (tree un) ∘ αdistl ∘ distr

For simplicity we will assume, as Backus did for /f, that when the function (tree f) is applied to an empty sequence, it will produce the right unit of f whenever that makes sense. *E.g.*, the right unit of un would be <>, so that

cart2:<<>, <A, B>>

= (tree un) ∘ αdistl ∘ distr : <<>, <A, B>>

= (tree un) ∘ αdistl : <>

= (tree un) : <>

= <> .

If we preferred, we could rewrite the definition of cart2 as

def cart2 = (null → [ ]; (tree un)) ∘ αdistl ∘ distr .

Suppose now that we want to write a program, cart, whose domain is not just pairs of sets but rather arbitrary n-tuples of sets. The simple extensions (tree cart2) and (/ cart2) are incorrect, since each will produce a sequence of pairs rather than a sequence of n-tuples.

Exercise 2. Evaluate (tree cart2):< <A,B>, <C,D>, <E,F> > .

A possible solution could be to proceed sequentially, say from right to left, by appending some "collecter" to the right of the sequence of input sets, and then to add each set to the collection in turn, *i.e.* to write a program of the form

$/p_1$ ∘ apndr ∘ [id, [[ ]]]

where p_1 is some program containing cart2. Again we prefer to avoid such a sequential solution if possible.

Exercise 3. Define two programs f_1 and f_2 such that

def cart = (tree f_1)∘f_2

will correctly define a program for computing general Cartesian products.

Given a program for general Cartesian products we could then write a program to compute the set of all subsets of an input set as

8) **def** power = α(tree un) ∘ cart ∘ α[[id], [ ]] ,

since *e.g.* power:<A, B> would then be computed as α[[id], [ ]] applied to <A, B> which yields

9) $< <<A>,<>>, <<B>,<>> >$,

followed by cart applied to 9) yielding

10) $<$ $<<A>,<B>>$,
$<<A>,<>>$,
$<<>,<B>>$,
$<<>,<>> >$,

followed by α(tree un) applied to 10) or

11) $< <A,B>, <A>, <B>, <> >$.

The FP style of looking for closed form, concurrent solutions can lead to some interesting solutions for what might have been considered inherently sequential problems; *e.g.*

def reverse = (tree un$\circ$[2, 1])$\circ\alpha$[id]

and

def last = (tree 2) .

Exercise 4. Find a program f_1 such that

def parsums = (tree f_1)

defines the APL sum-scan-reduction function, *e.g.* parsums:<1, 2, 3, 4> = <1, 3, 6, 10> .

Section 3. Using the Algebra of Functional Programs

In this section we 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 these notes 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 discussion 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. We 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

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

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

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

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

The meaning of an "infinite conditional" such as this is taken to be the limit of a chain of partial functions as follows:

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

In the remainder of this section we give the flavor of this algebraic approach by considering some examples. We will use the following algebraic laws without further motivation; arguments justifying their adoption as axioms can be found in [Ba 78].

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

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

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

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

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

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

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

(A8) $\quad a \to (a \to b; c); d = a \to b; d$

(A9) $\alpha f \circ apndl \circ [a, b] = apndl \circ [f \circ a, \alpha f \circ b]$

(A10) $/f \circ apndl \circ [a, b] = f \circ [a, /f \circ b]$

(A11) $\bar{a} \circ b = \bar{a}$, in the domain of definition of b

These are but a few of the many algebraic laws that can be inferred from the definitions of the combining forms. 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.

As a simple example of how these laws can be used to reason about a program, recall the definition of our program to compute the length of a sequence,

def length = $/+ \circ \alpha\bar{1}$.

We will prove that appending an element to a sequence increases its length by one. This is expressed in the algebra by the statement

$length \circ apndl \circ [a, b] = + \circ [\bar{1}, length \circ b]$ when a is defined .

That is, we need to show that for all functions a and b and for all objects for which a is defined, that the function $length \circ apndl \circ [a, b]$ is equal to the function $+ \circ [\bar{1}, length \circ b]$.

$length \circ apndl \circ [a, b]$

$= /+ \circ \alpha\bar{1} \circ apndl \circ [a, b]$, by def of length

$= /+ \circ apndl \circ [\bar{1} \circ a, \alpha\bar{1} \circ b]$, by (A9)

$= /+ \circ apndl \circ [\bar{1}, \alpha\bar{1} \circ b]$, by (A11) since a is defined

$= + \circ [\bar{1}, /+ \circ \alpha\bar{1} \circ b]$, by (A10)

$= + \circ [\bar{1}, length \circ b]$, by def of length.

These laws can be used not only to prove specific properties of programs like length, but also to prove many general theorems about functional program equivalence. As an example, we consider the following theorem from [Wi 80], which relates the powers of construction to the combining form insert left. (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,

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

Proof: (By induction on n)

Basis: $n = 1$.

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

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

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

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

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

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

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 by $\bar{\perp} = \bar{\perp}$ otherwise)

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

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

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 $1 \leq k \leq N$ and $g(k) \geq g(j)$ for $1 \leq j \leq N$. 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

$$0 \leq j < k \leq N \textit{ and } (\text{for } j < i \leq N,\ g(k) \geq g(i)) .$$

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) $\quad \text{iter} \circ [\text{id},s]$

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

(2) **def** iter = $\text{eq0} \circ 2 \rightarrow 1;\ \text{iter} \circ [2\text{max},\ s \circ 2]$,

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

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

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 $\bar{\perp}$-preserving, so according to the linear expansion theorem, f can be expressed as the infinite conditional

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

or

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

Therefore, our program (1) can be rewritten as

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

Now using the abbreviation

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

we can write the general term of (6) as

(8) $eq0 \circ 2 \circ G \to 1 \circ G$.

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

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

Therefore,

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

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

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

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

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

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) $/_L 2max \circ reviota$

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 just the language of functional programs, therefore the programmer who wants to reason about (*i.e.* to understand 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.

Section 4. The Formal Functional Programming Language, FFP

In this section we describe the FFP language defined in [Ba 78], and we show how it can be used to realize all of the notions in the informal FP language we have been studying. In particular we define a mapping from objects and functions in FP into formal expressions in FFP, and we extend the work of [Ba 78] by giving an algorithm for producing formal representations for arbitrary recursively defined FP functions. First we review the FFP language, giving its complete syntax and semantics, and then we show how it can be used to realize all of the notions of FFP.

The syntax of FFP is specified as follows:

1) Given a set A of atoms, the set E of expressions is recursively defined by:

a) $\perp$ is in E ("bottom" denotes the undefined object as in FP),

b) A is contained in E (all atoms are expressions),

c) if $e_1, \ldots, e_n$ are in E and $e_i \neq \perp$, then $\langle e_1, \ldots, e_n \rangle$ is in E, (all sequences of non-$\perp$ expressions are themselves expressions, and, as in FP, the sequence constructor is $\perp$-preserving).

d) if e_1 and e_2 are in E, then $(e_1{:}e_2)$ is in E, (this is a new constructor for denoting the application of one expression to another).

2) An expression that does not contain any applications, *i.e.* an expression built without using clause d) above, is called a *constant*, and the set of constant expressions is denoted C.

The semantics of FFP is given by a meaning function μ that maps every expression into a constant, *i.e.*

$$\mu{:}E \rightarrow C \ .$$

In presenting the definition of μ we use an auxiliary function $\rho(x)$ which can be read, "the function represented by the atom x." The definition of ρ will follow that of μ. We give the definition of $\mu(x)$ by cases on the form of the expression x as follows (the parenthesized line numbers are simply for later reference):

$\mu(x) =$

(μ1) $x = \perp \rightarrow \perp$

(μ2) $x \in A \rightarrow x$

(μ3) $x = \langle x_1, \ldots, x_n \rangle \rightarrow \langle \mu(x_1), \ldots, \mu(x_n) \rangle$

$x = (y{:}z) \rightarrow$

$y = \perp \rightarrow \perp$

(μ4) $y \in A \rightarrow \mu(\rho(y){:}\mu(z))$

(μ5) $y = \langle y_1, \ldots, y_n \rangle \rightarrow \mu(y_1{:}\langle y, z \rangle)$

(μ6) $\mu(\mu(y){:}z)$ otherwise .

Given this definition of μ we make the following observations:

a) The meaning of an atom is just itself (line μ2). Note that in most languages the meaning of an atom depends on the contents of some environment.

b) The meaning of a sequence is the sequence of the meanings of its constituent expressions (line μ3).

c) Note that a) and b) imply that μ maps all elements of **C** into themselves -- thus the name "constants."

d) The meaning of an atom applied to an expression $(a{:}e)$ is gotten by first applying the function represented by a to the meaning of e, and then taking the meaning of the result (μ4). Notice that this makes it possible to include an "apply" function,

$$\text{apply}{:}\langle e_1, e_2 \rangle = (e_1{:}e_2) ,$$

something that couldn't exist in FP, since there functions map objects into objects, and $(e_1{:}e_2)$ is not an object. In the definition of ρ we let $\rho(\text{APPLY}) = \text{apply}$.

e) The meaning of a sequence $\langle y_1, \ldots, y_n \rangle$ applied to an expression e is the meaning of the first element of that sequence applied to the pair -- sequence and expression ($\mu 5$). Backus calls this the rule of *metacomposition*, and it is this clause in the definition of μ that allows us to represent combining forms and recursive functions without the need for any environment maintaining mechanism.

f) As with Lisp [Mc 60], if in $(y{:}z)$ y is not the representation of some known function, then it is evaluated, and that value is applied to z ($\mu 6$).

The auxiliary function ρ has the functionality

$$\rho{:}A \to (C \to E) \ .$$

That is, ρ maps atoms into functions from (constant) expressions to expressions. In specifying ρ, we will use an FP-like notation for describing functions from $C \to E$. For example,

$$\rho(\text{NULL}) = \text{null} \ ,$$
$$\rho(\text{TAIL}) = \text{tl} \ ,$$

and so forth for each of the primitive functions of FP.

The combining forms can also be represented by a (finite) collection of representing atoms, due to the nature of the metacomposition rule. For example, we can take <CONST, X> to represent the constant combining form of FP $\bar{x}$, by defining

$$\rho(\text{CONST}) = 2 \circ 1 \ ,$$

since if y is some constant non-$\perp$ expression,

$$\mu(\langle \text{CONST, X} \rangle{:}y)$$

$$= \mu(\text{CONST}{:}\langle\langle \text{CONST, X} \rangle, y \rangle)$$

$$= \mu(\rho(\text{CONST}){:}\mu(\langle\langle \text{CONST, X} \rangle, y \rangle))$$

$$= \mu(2 \circ 1{:}\langle\langle \text{CONST, X} \rangle, y \rangle)$$

$$= \mu(2{:}\langle \text{CONST, X} \rangle)$$

$= \mu(X)$

$= X$.

Similarly, we can represent the combining form construction by defining

$$\rho(\text{CONS}) = \alpha\text{apply} \circ \text{tl} \circ \text{distr} \quad .$$

Exercise 5. Show that with this definition for ρ(CONS), $\mu(\langle \text{CONS, TAIL, ID} \rangle : \langle \text{A, B} \rangle) = \langle\langle \text{B} \rangle, \langle \text{A, B} \rangle\rangle$.

Exercise 6. Define ρ(COMP) so that <COMP, G, H> will correctly represent $g \circ h$ if G and H represent g and h respectively.

Exercise 7. Define ρ(ALPHA) so that <ALPHA, G> will correctly represent αg if G represents g.

At this point we have reviewed Backus' definition of the formal language FFP and we have seen how he used it to represent the objects, primitive functions, and combining forms of FP. In particular, we have seen that the rule of metacomposition enabled him to represent the combining forms of FP as FFP sequences. In [Ba 78] he also states, "in addition to its use in defining functional forms, metacomposition can be used to create recursive functions directly without the use of recursive definitions of the form"

def $f = Ef$,

and he shows by way of example that the object *<LAST>* will correctly represent the FP function defined by:

def last = null∘tl → 1; last∘tl

if *LAST* is taken to be the FFP object that represents the (nonrecursive) function

null∘tl∘2 → 1∘2; APPLY∘[1, tl∘2] .

One can verify that the meaning of $\langle LAST \rangle$ applied to the sequence $\langle A, B \rangle$ is indeed B, *i.e.* that

$$\mu(\langle LAST \rangle : \langle A, B \rangle) = B$$

and in general that

$$\mu(\langle LAST \rangle : \langle x_1, \ldots, x_n \rangle) = x_n$$

when $\langle x_1, \ldots, x_n \rangle$ is a constant expression of FFP.

For example,

$$\mu(\langle LAST \rangle : \langle A, B \rangle)$$

$$= \mu(LAST : \langle\langle LAST \rangle, \langle A, B \rangle\rangle)$$

$$= \mu(\rho(LAST) : \mu(\langle\langle LAST \rangle, \langle A, B \rangle\rangle))$$

$$= \mu(\rho(LAST) : \langle\langle LAST \rangle, \langle A, B \rangle\rangle)$$

$$= \mu((\text{null} \circ \text{tl} \circ 2 \rightarrow 1 \circ 2;\ \text{APPLY} \circ [1, \text{tl} \circ 2]) : \langle\langle LAST \rangle, \langle A, B \rangle\rangle)$$

$$= \mu(\text{APPLY} \circ [1, \text{tl} \circ 2] : \langle\langle LAST \rangle, \langle A, B \rangle\rangle)\ ,$$
$$\text{since}\quad \text{null} \circ \text{tl} \circ 2 : \langle\langle LAST \rangle, \langle A, B \rangle\rangle = F$$

$$= \mu(\text{APPLY} : \langle\langle LAST \rangle, \langle B \rangle\rangle)$$

$$= \mu(\langle LAST \rangle : \langle B \rangle)$$

and one step of the recursion is completed.

It is remarkable that by using such a simple device as metacomposition Backus was able to represent such recursively defined functions without relying on the existence of the (rather complicated) Y combinator of [Ch 41] and [Cu 58] or introducing into his system an additional mechanism to maintain an environment of name/value bindings and a special "labeling" form as was done in Lisp [Mc 60]. In fact, so powerful is the added environment of Lisp that the introduction of the special form LABEL is actually unnecessary. To represent the function defined by

DEFINE (LAST (λ (X) E_{LAST}))

we can use the pure Lisp expression

(λ (X) ((λ (LAST) (LAST X)) (QUOTE (λ (X) E_{LAST}))))

instead of the special form

(LABEL LAST (λ (X) E_{LAST})) .

So it is noteworthy that metacomposition enables us to represent the function "last" without the use of any added environment-maintaining mechanism.

However, the defining equation for "last" is an example of a particularly simple class of recursive forms -- the so called "tail recursions", and it remains to be shown that the existence of the FFP object *<LAST>* is not just due to some special properties of tail-recursive equations, but rather that all FP functions can be so represented in FFP.

We extend the treatment of [Ba 78] by giving a general algorithm for producing FFP representatives for arbitrary, recursively defined FP functions. In doing so, we will have demonstrated the universality of FFP, since all partial recursive functions can be defined in FP. (That all primitive recursive functions can be defined is clear. The *min* operator $\mu x{:}R(x)$ of [KL 52] can be realized by the defined function

Def least = R $\rightarrow$ id; least$\circ$+$\circ$[id, $\overline{1}$]

since then $\mu x{:}R(x) = \text{least}\circ\overline{0}$.)

We will first give the algorithm for functions defined by a single equation and then generalize it to include functions defined by sets of mutually recursive equations. In order to show that every function definable in the informal FP system can be represented in FFP, we must give some representation-producing mechanism π that maps the domain **F** of expressions denoting functions in FP into the domain of FFP expressions such that

$\forall f$ in **F**, $\rho(\pi(f)) = f$.

Note that π will be a purely syntactic mapping from expressions in FP to expressions in FFP.

Recall that there are three types of functions in the FP system:

1) primitives **P**,
2) the closure of **P** under the combining forms, and
3) functions defined by (recursive) functional equations.

For functions of type 1) and 2) π will produce the representations chosen by Backus, which we have described above. For example,

$$\begin{aligned}
\pi(\text{tl}) &= \text{TAIL} \\
\pi(1) &= 1 \\
\pi(\text{null} \circ \text{tl}) &= \langle \text{COMP, NULL, TAIL} \rangle \\
\pi([\text{id}, \alpha\text{tl}]) &= \langle \text{CONS, ID, } \langle \text{ALPHA, TAIL} \rangle \rangle \\
\pi(\overline{3}) &= \langle \text{CONST, 3} \rangle \\
&\text{etc.}
\end{aligned}$$

We will make one modification to the representation of [Ba 78]. Since the sequence constructor of FFP is $\perp$-preserving, we cannot use $\langle \text{CONST}, x \rangle$ to represent $\overline{x}$ for all objects x. Notice that while the object $\langle \text{CONST}, \perp \rangle$ *will* correctly represent the function $\overline{\perp}$, a function such as

$$\text{zerofilter} = \text{eq0} \rightarrow \text{id}; \overline{\perp}$$

is not correctly represented by the FFP object

$$\langle \text{COND, EQ0, ID, } \langle \text{CONST, } \perp \rangle \rangle$$

since the presence of $\perp$ reduces that object to $\perp$. We will use the object UNDEF to represent the FP function $\overline{\perp}$.

For functions defined by a recursive equation we define π as follows:

Definition If $f = Ef$ is a recursively defined FP function, where E is built up by applying combining forms to primitive functions and the function letter f, then $\pi(f)$ is defined to be the FFP object $\langle \pi_f(Ef) \rangle$ where $\pi_f(Ef)$ is the FFP object that results from

applying the following representation-producing algorithm π_f to the form Ef.

In describing this algorithm we will use a shorthand notation to clarify the exposition. Since we know that all primitives and their closures under the combining forms have representing expressions in FFP, we will frequently use a more readable FP-like notation to denote those FFP representations. For example, instead of <COMP, TAIL, 2> we will write tl∘2, and instead of <COMP, APPLY, <CONS, 1, 2>> we will write APPLY∘[1, 2].

Algorithm (Representation-producing) The FFP representation of a form Ef with respect to the function letter f, π_f(Ef), is constructed recursively by cases on the structure of Ef:

Form of Ef	**FFP representation of Ef**
p (p in **P**)	p∘2
f	APPLY
$\bar{x}$	$\bar{x}$ if $x \neq \perp$, else UNDEF
Pf → Qf; Rf	π_f(Pf) → π_f(Qf); π_f(Rf)
[Pf, Qf]	[π_f(Pf), π_f(Qf)]
Pf∘Qf	π_f(Pf)∘[1, π_f(Qf)]
α(Pf)	???

Exercise 8. Define the FFP representation of the form α(Pf) .

To illustrate the use of this algorithm, recall the definition

def last = null∘tl → 1; last∘tl .

Then the FFP object representing last is <*LAST*> where

LAST

$= \pi_{last}(E(last))$

$= \pi_{last}(null \circ tl \rightarrow 1; last \circ tl)$

$= \pi_{last}(null \circ tl) \rightarrow \pi_{last}(1); \pi_{last}(last \circ tl)$

$= \pi_{last}(null) \circ [1, \pi_{last}(tl)] \rightarrow \pi_{last}(1); \pi_{last}(last \circ tl)$

$= (null \circ 2) \circ [1, tl \circ 2] \rightarrow 1 \circ 2; \pi_{last}(last) \circ [1, \pi_{last}(tl)]$

$= null \circ 2 \circ [1, tl \circ 2] \rightarrow 1 \circ 2; APPLY \circ [1, tl \circ 2]$

$= null \circ tl \circ 2 \rightarrow 1 \circ 2; APPLY \circ [1, tl \circ 2]$,
since defined $\circ$ 2 implies defined $\circ$ 1,

which, as we have seen, is exactly the expression chosen by Backus in [Ba 78].

To prove that $<\pi_f(Ef)>$ always correctly represents the function f defined by

def f = Ef

we must show that $\rho(<\pi_f(Ef)>) = f$,
i.e. that for all objects x, $\mu(<\pi_f(Ef)>:x) = f:x$
or equivalently (according to the definition of metacomposition) that

$\mu(\pi_f(Ef):<<\pi_f(Ef)>, x>) = f:x$.

We will not give the formal proof in these notes. The interested reader can find the details in [Wi 81].

Thus we know that even very complex functions can be represented in the FFP notation. For example, the non-primitive-recursive "Ackermann function" defined by

def ack = eq0 $\circ$ 1 $\rightarrow$ a $\circ$ 2; eq0 $\circ$ 2 $\rightarrow$ ack $\circ$ [s $\circ$ 1, $\overline{1}$];
ack $\circ$ [s $\circ$ 1, ack $\circ$ [1, s $\circ$ 2]]

(recall that a and s are primitives that, respectively, add and subtract 1) is represented by $<ACK>$ where, according to the algorithm,

ACK

$= \pi_{ack}(E(ack))$

$= \pi_{ack}(eq0 \circ 1 \rightarrow a \circ 2;\ eq0 \circ 2 \rightarrow ack \circ [s \circ 1, \bar{1}];\ ack \circ [s \circ 1, ack \circ [1, s \circ 2]])$

$= \pi_{ack}(eq0 \circ 1) \rightarrow \pi_{ack}(a \circ 2);\ \pi_{ack}(eq0 \circ 2) \rightarrow \pi_{ack}(ack \circ [s \circ 1, \bar{1}]);$
$\pi_{ack}(ack \circ [s \circ 1, ack \circ [1, s \circ 2]]))$

$= eq0 \circ 1 \circ 2 \rightarrow a \circ 2 \circ 2;\ eq0 \circ 2 \circ 2 \rightarrow \pi_{ack}(ack \circ [s \circ 1, \bar{1}]);$
$\pi_{ack}(ack \circ [s \circ 1, ack \circ [1, s \circ 2]])$

$= eq0 \circ 1 \circ 2 \rightarrow a \circ 2 \circ 2;\ eq0 \circ 2 \circ 2 \rightarrow APPLY \circ [1, [s \circ 1 \circ 2, \bar{1}]];$
$APPLY \circ [1, [s \circ 1 \circ 2, APPLY \circ [1, [1 \circ 2, s \circ 2 \circ 2]]]]$

To construct representations for functions defined by mutually recursive equations, we revise the representation producing algorithm as follows.

Algorithm (Extended representation-producing) If D is an ordered set of n atoms denoting defined function names, then the FFP representation of a form E with respect to the set D, $\pi_D(E)$, is constructed recursively by cases on the structure of E:

Form of E	**FFP representation of E**
p (an atom not in D)	$p \circ 2$
f_i (the i^{th} element of D)	$APPLY \circ [[i, 1, 2, \ldots, n] \circ TAIL \circ 1, 2]$
$\bar{x}$	$\bar{x}$ if $x \neq \perp$, else UNDEF
$P \rightarrow Q; R$	$\pi_D(P) \rightarrow \pi_D(Q); \pi_D(R)$
$[P, Q]$	$[\pi_D(P), \pi_D(Q)]$
$P \circ Q$	$\pi_D(P) \circ [1, \pi_D(Q)]$

(Note that π_D is just the same as π_f except for the case $E = f_i$.)

Then given a set of definitions for the n functions $\mathbf{f} = \{f_1, \ldots, f_n\}$:

$$\begin{aligned} &\textbf{def } f_1 = E_1\mathbf{f} \\ &\textbf{def } f_2 = E_2\mathbf{f} \\ &\qquad \cdot \\ &\qquad \cdot \\ &\qquad \cdot \\ &\textbf{def } f_n = E_n\mathbf{f} \end{aligned}$$

we will choose for $\pi(f_i)$ (the FFP representation of the function f_i) the $(n+1)$-element sequence

$$\langle F_i, F_1, F_2, \ldots, F_n \rangle$$

where $F_i = \pi_D(E_i\mathbf{f})$ for $1 \leq i \leq n$.

To show that these objects correctly represent the defined functions, we need to show for all objects x and FP expressions Hf (*i.e.* functions built up from the primitives and the functions f_i by the application of combining forms) that

$$\mu(\pi_D(\mathrm{Hf}):\langle ENV, x\rangle) = \mathrm{Hf}:x\ ,$$

where ENV is taken to be any sequence of $n+1$ elements whose $(i+1)^{st}$ element is the representation of the i^{th} defined function and whose first element is any arbitrary FFP object. Again, the interested reader can find the detailed proof in [Wi 81].

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." Draft paper distributed at Summer Workshop on Programming Methodology, Univ. of Calif. at Santa Cruz, August 1979.

[Ch 41] Church, A. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, N.J., 1941.

[Cu 58] Curry, H.B. and Feys, R. *Combinatory Logic, Vol. 1.* North-Holland Pub. Co., Amsterdam, 1958.

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

[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 60] McCarthy, J. "Recursive functions of symbolic expressions and their computation by machine, Part 1." *CACM*, April 1960.

[Wi 80] Williams, J.H. "On the development of the algebra of functional programs." Report No. RJ2983, IBM Research Laboratory, San Jose, California, October 1980.

[Wi 81] Williams, J.H. "Formal representations for recursively defined functional programs." in *Formalization of Programming Concepts*, Lecture Notes in Computer Science No. 107, Springer-Verlag, April 1981.