FUNCTIONAL PROGRAMMING
USING FP

by Peter G. Harrison and Hessam Khoshnevisan

How to program without objects

In 1977, John Backus introduced a functional style of programming in which variable-free programs are built from a set of primitive programs by a small set of combining forms (functionals that are also often referred to as program-forming operations or PFOs) and by recursive definitions. This style is embodied in the language FP, which facilitates the manipulation of the functions themselves rather than repeatedly creating new objects from old ones in an auxiliary domain. FP thus relates to a higher level of analysis than do the more common, object-oriented functional languages. In fact, FP has its own functional algebra that prescribes rules for manipulation of functions and so simplifies reasoning about programs. FP systems have the following properties:

- Programs have extremely simple semantics.
- There often exist nonrecursive expressions for many functions that are normally recursively defined (similar to using a loop in Pascal).
- Programs exhibit a clear hierarchical structure.
- The principal combining forms are the operations of the powerful algebra of FP programs. This algebra can be used to solve equations for recursively defined programs and to transform programs into versions that run more efficiently or consume less space.
- Transformation of recursive to nonrecursive functions and to loops can often be achieved automatically in FP.

The main obstacle to the advancement of functional programming languages has been their poor run-time performance on conventional computers. This is primarily due to the large number of (mainly stack-based) manipulations required to preserve referential transparency in the languages. Von Neumann computers execute instructions sequentially and so are tailored toward supporting sequential languages with destructive assignment. One way to improve performance is to offer a radically different computer architecture, specifically tailored toward supporting functional languages (see references 1 and 2). But there is also a need to provide efficient implementations of these languages on conventional machines. These are likely to remain widespread for the foreseeable future, in particular in the personal computer market, whatever the impact of any new architectures. A route to improved performance is to transform recursively defined solutions into iterative ones. FP programs lend themselves to this type of transformation very well since they do not refer to the auxiliary domain of objects, which often obscures the process of program transformation. Transformations of this sort may also prove beneficial to parallel implementations by increasing the size of the basic unit of work performed by each processor. This naturally reduces communication overhead, which often limits the performance of parallel machines.

(continued)

Dr. Peter G. Harrison is a lecturer in the Department of Computing, Imperial College. Hessam Khoshnevisan received his bachelor's degree in computing science from Imperial College, where he now works as a research assistant on the ALICE project. They can be reached at Imperial College, 180 Queen's Gate, London SW7 2BZ, England.
The FP algebra provides a formal basis for program transformation and so facilitates its automation. [Editor's note: The idea of program transformation is discussed in "Program Transformation" by John Darlington on page 201.] Conventional transformation techniques prescribe algorithms for transforming functions when applied to certain arguments. The transformation strategy in FP is based upon theories that state identities between functional expressions. A transformation is then simply an instance of the application of a theorem.

**FP Systems**

An FP system consists of the following:

- A set of primitive functions (for example, the arithmetic operators +, -, *, etc.).
- A set of PFOs. These are programming constructs analogous to while loops, conditionals, and so on. found in conventional languages like Pascal, and they may be used to create more complex functions from simpler ones.
- A domain of objects that might be, for example, integers, characters, sequences, etc.

User-defined functions are defined in terms of these FP system components.

PFOs are the programming constructs of FP (like while, if...then...else, etc. of Pascal) and differ from the programming constructs of other languages (including other functional languages) in that they are predefined operations on functions as opposed to objects. We are all familiar with the conditional statements, such as if P then Q else R in Pascal. The conditional operation of FP (f → g; h) is similar except that the predicate and the true and false branches of the conditional are expressions involving only functions, namely f, g, and h. These expressions can be primitive functions, user-defined functions, or expressions built using the PFOs. In short, all PFOs take a number of functions as arguments and return a single FP function. All FP functions take a single object as input and produce a single object as their result: i.e., they are of type Object → Object. All FP systems are equipped with an operation called application, which, given a function and an object, produces the result of applying the function to the object. The notation f:x is used to represent the application of function f to the object x.

To define an FP system, we must specify the set of primitive functions, the set of PFOs, and the set of objects. Listed in the text box "FP Syntax" on page 228 are some examples of primitives and PFOs that might be present in an FP system. The meaning of each is given by specifying the result of its application to various kinds of objects. If PFOs are applied to any other kind of object, not mentioned in their meaning, the result of the application will be ⊥. This is read as "bottom" and denotes the undefined object or error. So, for example, an attempt to add the integer 1 to the character a will yield ⊥ since a is not the correct type for the + primitive.

The set of objects is determined by the set of atoms chosen. For example, if we choose to consider the set of integers, the set of nonempty strings, and the atoms T and F denoting true and false, then T, 66, -88, 0, c, Hello, F, and GH are all valid atoms.

Considering now the set of objects that we can write down, we know that
- Every atom is an object.
- All sequences of objects, denoted by <x1, x2, ..., xn> where each xi is an object other than ⊥ (1 ≤ i ≤ m), are objects.
- ⊥ is an object.
- Thus, <(66, ⊥), ALA > GAB FO W is an object.

**USER-DEFINED FUNCTIONS**

The set of primitives and PFOs determines the set of functions that can be defined. The user can define functions using the def statement in FP. For example,

```
def MyFun = AFun → BFun; CFun
```

defines a function in terms of an FP PFO—in this case, the conditional. Note that AFun, BFun, and CFun must all be functions (which can themselves be built by the use of PFOs).

Each PFO takes a number of functions as parameters.

The number of parameters is determined by the particular PFO. The “conditional” PFO takes three parameters, whereas the “construction” PFO can accept any number of function parameters. Recursive function definitions are allowed, so in the above example any of the expressions AFun, BFun, or CFun can refer to MyFun.

Note that the syntax for function application and composition in basic FP can be rather clumsy; for example, add: <2, 3> as opposed to 1 + 3 or a: <2, <3, 4, 5> > as opposed to 2: <3, 4, 5>.

However, it is easy to incorporate infix notation into many variable-free functional expressions. For example (using infixes ·, +), f:g represents a·f[i, g] and f + g represents +·f[i, g]. But note that if f = [h, g], ·+f is clearer than (1·f) + (2·f) or (1 + 2)·f.

**PROGRAMMING IN FP**

A user-defined function is defined by one and only one definition. The name given to the function must be unique and must not coincide with the names associated with the primitive functions and PFOs. Remember that no part of a definition is a result itself; instead, each part is a function that must be applied to an argument to obtain a result.

Here are four simple examples of recursive user-defined functions:

```
def last = null? {1; last\_}
def len = null → {0; +·[1; len\_]}    
def cat = null → {1 + 2·; a·[1+1, cat\_[1+1, 2]}    
def fact = eq0 → {1; ·+·[id, fact\_sub]}    
```

(continued)
The first returns the last element of a sequence, the second returns the length of a sequence, the third concatenates two sequences, and the last defines the factorial function.

Observe that the examples of function definitions given above, although written in FP, are written in a recursive style that is familiar from conventional functional languages such as Hope, LISP, and KRC. For instance, the definition of last says that if the tail of the sequence is empty then select the first element of it. Otherwise look for the last element of the tail of the object using a recursive call to last. In the FP style, it is often possible to replace an explicitly recursive definition by an equivalent nonrecursive functional expression.

The function last could, in most systems, be expressed purely in terms of primitives; for example, def last = 1r. Similarly a more natural FP definition for factorial is def f = (1 / *)\text{ iota} or, alternatively, def f = (\times)\text{ iota}. This "inserts" a * between each element of the sequence <\, 1, 2, \ldots, n >.

A function to concatenate (append in Hope) two sequences is def cat = (\, 1\, a)\text{ op}ar. This nonrecursive definition successively appends an item from the end of the first sequence onto the beginning of the second. Note that / cat will then be a function that concatenates any number of sequences.

The function len, for length, would be a primitive in most systems, but if not, the more natural FP definition could be def len = null -> 0 ; + (c \downarrow)

This definition says that len x is 0 if x is empty; otherwise, change each element of x into 1, and then add up the 1s. The items are literally counted.

Observe that the more natural FP solutions are nonrecursive (effectively the recursion has been pushed into the PFOs used). Although these nonrecursive definitions may look strange initially, they express an equally obvious solution. By becoming familiar with the high-level PFOs and their concise syntax, you can give concise, expressive, and flexible definitions that are often nonrecursive, using just a few symbols. (It has been suggested that programmer productivity is inversely proportional to the number of characters required in a program.)

Here, then, are some more complex examples that make use of the PFOs:

**VECTOR PRODUCT:** We can define the function VectorProduct to be def VP = (/ +)\text{ op}(\times)\text{ op trans}. Application of VP to a pair of equal-length vectors first creates the sequence of matched pairs of components (result of trans), then multiplies each pair, and finally sums these results. For nonequal-length vectors, the result of trans is \bot and so therefore is the result of VP. We can explain each step in evaluating VectorProduct applied to the pair of vectors <\, <1,2,3>, <5,6,4> > as shown in Table 1.

**MATRIX MULTIPLY:** We can define the function MatrixMultiply to yield the product of any pair <\, y, z > of conformable matrices, where each matrix is represented as the sequence of its rows:

\[ y = <\, y_1, \ldots, y_m >\]
\[ where \quad y_i = <\, y_{i1}, \ldots, y_{im} > \quad for \quad i = 1, \ldots,m \]
\[ z = <\, z_1, \ldots, z_n >\]
\[ where \quad z_i = <\, z_{i1}, \ldots, z_{in} > \quad for \quad i = 1, \ldots,n \]

The function is then defined as

\[ def \quad MM = (a_\alpha \text{ V P}) \circ (a_\alpha \text{ dist}) \circ \text{ dist} \circ [1, \text{ trans} \circ 2] \]

**BINARY TREE INSERT:** Suppose that a binary tree is represented by a sequence of three elements, where the first element is the left binary tree, the second is the data at the node (which we will assume is a number), and the third is the right binary tree. A function InsertInSortedBinaryTree (IISTB), which inserts a number into a tree in such a way that all elements in the left subtree are less than the smallest number in the whole of the right subtree, might look like this:

\[ def \quad \text{ II STB} = \text{ null} \rightarrow 1 \rightarrow [\, 1, \, 2, \, 3 \, ]; \]
\[ \text{ le} \rightarrow [\, 1, \, 2 \, ] \rightarrow [\, 1, \, 2, \, 3 \, ]; \]
\[ [\, 1, \, 2, \, 3 \, ]; \quad \text{ II STB} \rightarrow [\, 1, \, 2, \, 3 \, ]; \]
\[ \text{ II STB} \rightarrow [\, 1, \, 2, \, 3 \, ]; \quad \text{ II STB} \rightarrow [\, 1, \, 2, \, 3 \, ]; \]

**PART PRODUCT:** A function ParProds, when given a sequence of integers <\, x_1, \ldots, x_n >, produces a sequence of integers <\, y_1, \ldots, y_m > such that, for 1 \leq i \leq m,

\[ y_i = y_i \times \ldots \times y_i \]

Thus ParProds\text{ iota} = 5 = <\, 1, 2, 6, 24, 120 >

i.e., the sequence of factorials of the numbers 1 through 5. We may start from the observation that

(continued)
From a set of axioms that are not self-contradictory, an algebra may be defined.

ParProds : \langle x_1, \ldots, x_n \rangle
= PARar : < ParProds : \langle x_1, \ldots, x_n \rangle, y >
where PARar = ar \circ [1, \ast \circ [1 \ast 1, 2]]

Unfortunately, we cannot simply right-insert the PARar function because it requires that the sequence to which it is applied has a sequence for its first element. We first have to make this element into a sequence. The function ParProds therefore becomes:

\text{def } \text{ParProds} = \text{null } \to \emptyset;
\text{def } (\text{\textbackslash PARar}) \text{\textbackslash eal} \circ [1], [1];
\text{def } \text{PARar} = \text{ar} \circ [1, \ast \circ [1 \ast 1, 2]]

We may now use this nonrecursive function in the definition of other useful functions. For example, it is a relatively simple matter to extend these concepts for the evaluation of polynomials.

THE FP ALGEBRA OF PROGRAMS
Just as a set of functions mapping a domain of objects into itself may define an algebra on that domain, so too may a set of functionals define an algebra on a domain of functions. The reader will probably be quite familiar with the concept of the field of real numbers (objects) under the composition rules of addition and multiplication (functions) that possess the necessary properties such as associativity; he or she is surely familiar with the well-known algebra that follows. In the same way, a set of axioms that are not self-contradictory may be defined on a set of functions under composition rules given by a set of functionals. From these axioms, it may be possible to establish, as theorems, further properties about the sets of functions and functionals; i.e., an algebra may be defined. Through the algebra, relationships between functions may be established as identities, independent of the domain of objects to which they are applied. The two sides of such an identity yield an equation at the object level for every argument to which they are each applied. Thus the functional algebra provides a more general, higher level of reasoning in which quite powerful arguments can be expressed and results deduced. Note that any set of axioms could be chosen provided they are consistent, but in order that the resulting algebra be useful, the axioms should not contradict known properties of conventional functional languages when the functions of each side of an identity are applied to objects. Thus, for example, we would not choose for an axiom the statement

\text{f} _{o} [g, h] = \text{[f} _{o} g, \text{f} _{o} h]
PROGRAMMING IN FP

Backus's set of axioms

for FP is consistent but has

not been shown to be complete.

for all functions \( f, g, h \), since for object \( x \) it is not true in general that

\[
f(g(x), h(x)) = f((g(x), h(x))
\]

However, we can and do choose

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

Backus has presented consistent axioms of the FP functional algebra (reference 3), although the set has not been shown to be exhaustive. When applied to an arbitrary object, each is duly seen to yield an equality that is known to hold. As a simple example of the use of the FP algebra in formal reasoning, we first prove the equivalence between the recursive and nonrecursive definitions of factorial considered above.

We know that, by definition,

\[
\text{ iota } = \text{ eq0 } \rightarrow \text{ null}; \text{ are}[\text{ iota } \circ \text{ sub1 } \circ \text{ id }]
\]

and FP laws state that for all functions \( f, g, h \),

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

(this is easily checked from the definition of \( \circ \)), and for Boolean-valued function \( p \),

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

(easily checked from the definition of \( \rightarrow \)). Thus,

\[
(\circ \circ ) \circ \text{ iota } = \text{ eq0 } \rightarrow (\circ \circ ) \rightarrow \text{ null}; \quad (\circ \circ ) \circ \text{ are}[\text{ iota } \circ \text{ sub1 } \circ \text{ id }]
\]

\[
= \text{ eq0 } \rightarrow 1; \quad \circ [\circ ] \circ \text{ iota } \circ \text{ sub1 } \circ \text{ id }
\]

and writing \( 1 \) for \( (\circ \circ ) \circ \text{ iota } \) gives

\[
1 = \text{ eq0 } \rightarrow 1; \quad \circ [\circ ] \circ \text{ sub1 } \circ \text{ id }
\]

The power of the algebra has been further developed and exploited by Williams (reference 4) and Backus (reference 5), who introduced the linear class of functional forms. A functional form is a functional expression that contains function variables, and a linear form possesses certain properties relating to function expansion, discussed below. A function \( f \) is defined to be (functionally) linear if it has the "else part" given by a linear form in \( f, i.e., f = p \rightarrow q; \text{ Hf } \) for some fixed functions \( p \) and \( q \) and linear form \( H \). Linear functions such as \( f \) satisfy the Linear Expansion Theorem (LET), which states, loosely, that for object \( x \), if \( f(x) \) is defined, \( f(x) = (H \circ q)(x) \) for some integer \( i \), given by another form \( H_1 \) (the "predicate transformer") known in terms of \( H \). Specifically, \( i \) is the least integer such that \( (H \circ q)(x) \rightarrow 1. \) This solution is clearly iterative at the function level of description—if a function

(continued)
FP SYNTAX

Examples of primitives that might be present in an FP system include the following:

Add, Subtract, Multiply, Equals, etc. (+, -, *, eq, etc.)
When \( x \) is of form \(<yz>\) and \(yz\) are numbers, then \( +x \) yields the sum of \(y\) and \(z\)
otherwise \( +x = \perp \)
(The others are defined similarly.)

Greater Than, Less Than, Less Than or Equal, etc. (gt, lt, le, etc.)
When \( x = <yz> \) and \(yz\) are numbers then
\( \text{if } y > z \text{ then } gt(x) = T \text{ else } gt(x) = F \)
(The others are defined similarly.)

And, Or, Not (and, or, not)
When \( x = <T, T> \) then and\(x = T\)
When \( x = <F, T> \) or \(x = <T, F>\) or \(x = <F, F>\) then and\(x = F\)
(The others are defined similarly.)

Null (null)
When \( x = <> \) then null\(x = T\)
When \( x = \perp \) then null\(x = \perp\)
otherwise null\(x = F\)

Append Left (al)
When \( x = <y, <> > \) then al\(x = <y>\)
When \( x = <y, z_1, \ldots, z_m> > \) then al\(x = <y, z_1, \ldots, z_m> >\)

Append Right (ar)
When \( x = <>, y > \) then al\(x = <y>\)
When \( x = <z_1, \ldots, z_m, y> > \) then ar\(x = <z_1, \ldots, z_m, y> >\)

Selectors (1, 2, 3, \ldots)
When \( x = <z_1, \ldots, z_m> > \) then if \(m \geq 1\) then \(i:x = z_i\) else \(\perp\)

Right Selectors (1, 2, 3, \ldots)
As above, selects first, second, etc., from the right of the sequence.

Identity (id:x)
\(x\)

Transpose (trans)
When \( x = <>, \ldots, <> > \) then trans\(x = <>\)
When \( x = <x_1, \ldots, x_n> > \) where \(x_i = <x_{1i}, \ldots, x_{ni}> \)
for \(1 \leq i \leq n\) then trans\(x = <z_1, \ldots, z_n> >\)
where \(z_j = <x_{1j}, \ldots, x_{nj}> \) for \(1 \leq j \leq k\)
otherwise trans\(x = \perp\)

Distribute Left (distl)
When \( x = <y, <> > \) then distl\(x = <> \) When \( x = <y, <z_1, \ldots, z_n> >\)
then distl\(x = <y, z_1, \ldots, y, z_n> >\)

Distribute Right (distr)
When \( x = <>, y > \) then distr\(x = <>\)
When \( x = <z_1, \ldots, z_n, y> > \)
then \( <z_1, y, \ldots, z_n, y> >\)

Iota (iota)
If \( x = 0 \) iota\(x = <> \) if \( x \) is a positive integer iota\(x = <1,2,\ldots, x>\)
otherwise iota\(x = \perp\)

Others
Some other possible primitives are head (hd), tail (tl), right tail (rtl), rotate left (roll), rotate right (rotr), subtract one (sub1), is equal to zero (eq0), and so on. Note that all these are just as easily written in FP using top primitives and the PFOs below. For example.

cdf eq0 = eq>[id, 0]
cdf sub1 = -eq[id, 1]

Listed below are some examples of the type of PFOs that could be chosen for an FP system.

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

Construction
\([f_1, f_2, \ldots, f_n]\) such that \(x = <f_1x, f_2x, \ldots, fnx>\)

Condition
\((p \rightarrow f, g)\):
\(x = \begin{cases} f & \text{if } (p(x)) = T \\ g & \text{if } (p(x)) = F \end{cases}\)

Insert Left
\([f \ldots x = \begin{cases} g & \text{if } x = <y> \text{ then } y \\ \perp & \text{if } x = <y_1, \ldots, y_n> \text{ and } m > 2 \\ f(y_1, \ldots, y_n) > & \text{otherwise } \perp \end{cases}\]

Insert Right
\([f \ldots x = \begin{cases} g & \text{if } x = <y> \text{ then } y \\ \perp & \text{if } x = <y_1, \ldots, y_n> \text{ and } m > 2 \\ f(y_1, \ldots, y_n) > & \text{otherwise } \perp \end{cases}\]

Apply To All
\(af: x = \begin{cases} f & \text{if } x = <> \text{ then } <> \\ f(y_1, \ldots, y_n) > & \text{otherwise } f \end{cases}\)

Constant
\(f: x = \begin{cases} f & \text{if } x = \perp \text{ then } \perp \\ \text{otherwise } f \end{cases}\)
(Here \(f\) is an object parameter.)

To date, FP has been available only to researchers. Interpretive FP systems for relatively large computers have been implemented at such institutions as INRIA; the University of Paris; the University of California at Berkeley; and Westfield College and Imperial College, London. However, we at Imperial College hope to make an FP compiler for VAX and for conventional microcomputers such as the IBM PC available to the general public by the end of this year.
Linear functions are typically translatable into iterative form.

could be "accumulated" by successively passing round a loop, its result after i cycles could simply be applied to the object x. Translation into a loop at the object level appears possible. The importance of expansion theorems in general is that they give nonrecursive solutions to the recursion equations defining certain functions. Further use of the FP algebra may also derive nonrecursive solutions as pure FP expressions from linear expansions (reference 4).

We have demonstrated that the linear functions constitute a well-behaved class. We will conclude the discussion of linearity by identifying some linear forms and indicating how they may be detected automatically. It can be shown that the primitive forms of composition, condition, and construction are linear and that the linear class is closed under functional composition. The closed property means that if a linear form is applied to a function argument, which is itself the result of applying another linear form to a function variable, the resulting composite form is linear in the function variable. Thus the compiler can detect in many cases whether a defined function is linear and, if so, determine its predicate transformer, referred to above. For example, any form that is built up from the PFOts composition, construction, and condition and has only one occurrence of its function variable argument must be linear. More important, linear functions are typically translatable into iterative form, and the subject of current research at Imperial College is to automatically generate an iterative implementation (a loop) for a linear function, in particular any defined by a multiple composition of primitive linear forms.

Clearly then, the class of linear forms is an important one, and recent results (reference 6), again relying on the functional algebra, facilitate automatic transformation of a significant class of nonlinear functions into linear form, from which an iterative implementation follows (it is hoped). Mutually recursive definitions also may be similarly transformed under appropriate conditions, further extending the class of optimizable functions. Probably the prime example of these results is the transformation of the Fibonacci function into \( f = \text{log} \) where

\[
g = \text{let} x := \text{[T, T];} (+,1)\circ g \circ \text{sub1}
\]

By application of the linear expansion theorem for the linear form \( H \) given by \( Hg = [+1] \circ g \circ \text{sub1} \), with \( H, a = a \circ \text{sub1} \) for function \( a \).

\[
g \circ x = [+1]^{-1}; <1, 1> = [+1]^{-1} : <2, 1>
\]

\[
= [+1]^{-1} : <3, 2> = \ldots
\]

(continued)
PROGRAMMING IN FP

This reflects the usual way of implementing the Fibonacci iteration using two accumulators. [Editor's note: Compare this with the Hope transformation of the same function in John Darlington's article on page 201.] Further optimization is often possible for a set of mutually recursively defined functions (see reference 6). When such functions are combined with the linearity techniques used in the previous example, some powerful optimization becomes possible. Dijkstra's FUSC function satisfies the appropriate conditions and can be converted into iterative form. Denoting "divide by two" by $d$, $s = s_{\text{bl}}$, and $p = p_{\text{sl}}$, FUSC is defined by:

$$\text{fusc} = \text{id} \rightarrow \text{id}; \text{even} \rightarrow \text{fusc} \circ d; + [\text{fusc} \circ d; \text{fusc} \circ d;]$$

The theorem gives $\text{fusc} = 1_{\text{eg}}$ where

$$g = \text{id} \rightarrow \text{id}; \text{le} \rightarrow (\text{even} \rightarrow [L, g, s]; [M, g, s]); \text{even} \rightarrow [L, g, M, g]; [M, g, L, g]$$

where $L, g = 1_{\text{eg}}; L, g = 2_{\text{eg}}$;

$$M, g = +_{\text{eg}}; M, g = +_{\text{eg}}$$

Thus the last branch of the definition for $g (\geq 2)$ becomes

$$\text{even} \rightarrow [1, +_{\text{eg}}]; (+, 2)_{\text{eg}}$$

This reflects precisely the iteration of Dijkstra, and since the function is readily recognizable as linear in this form, the corresponding loop in an imperative programming language could be generated by the compiler.

REFERENCES


BIBLIOGRAPHY
