From Function Level Semantics to Program Transformation and Optimization

John Backus
IBM Research Laboratory
San Jose, California 95193

ABSTRACT The software crisis results from our disorderly concepts of "program". These make programming an art, rather than an engineering discipline. Such a discipline would at least require that we have stocks of useful off-the-shelf programs and collections of standard theorems that can be applied repeatedly. We have neither.

Mathematical systems are often distinguished by a set of operations that (a) map a set of entities into itself, (b) have simply understood results, and (c) obey a set of strong algebraic laws. Neither conventional programs nor "object level" functional programs are entities belonging to such a system. The standard operations on conventional programs violate (b) and (c); object level functional programs normally employ lambda abstraction as their program building operation and it violates (a) and (c). Other problems of these program concepts are reviewed.

Function level programs are the entities of just such a mathematical system: programs are built by program-forming operations having good algebraic properties. Hence they are the subject of a large number of general theorems, theorems that are applicable in practice. We give examples. Function level programs also have the possibility of providing solutions to many of the other problems reviewed.

The paper reviews the function level FP system of programs, sketches a function level semantics for it, and from the equations of that semantics develops some moderately general results concerning linear, recursively defined functions, one concerning recursion removal. It then discusses other general, directly applicable results in the literature and shows that they are essentially function level results and are best presented and recognized in that form.

The final section is about optimization; it shows how some FP programs can be transformed into others that run as fast as Fortran programs. It introduces "Fortran constructs" into FP, pure functions that have an obvious corresponding Fortran-like program. It exhibits a number of function level identities for these constructs and shows how these can be used to convert inefficient FP programs into efficient Fortran-like ones.

1. Introduction

The primary reason for the software crisis is that programming has failed to become an engineering discipline. One mark of such a discipline is a collection of standard constructs and theorems. Thus electronics has a set of standard circuits and standard theorems that are used over and over by practicing engineers to design a new device and verify that it will work correctly.

In contrast, programs are not constructed from a collection of off-the-shelf programs; each program is basically written anew from the beginning. There are few general theorems that can applied directly to a given program to help prove it correct. Typically each program is proved correct, if at all, also from the beginning.

Thus programming is a low level business, one that is guided by many helpful principles, but which has not reached a level of elegance and power in which we can accumulate reusable programs and generally applicable theorems.

This software crisis is basically caused by the fact that we have chosen the wrong concepts of "program" to work with. Let us make a brief survey of the difficulties that attach to two basic kinds of program concepts.

1.1 Problems with the von Neumann concept of "program"

The von Neumann concept of a program as a mapping of one store into another has the following basic problems:

- Programs depend on storage plans; programs to be combined must have a common storage plan, hence programs cannot be built from existing, independently written, off-the-shelf programs. Instead they are typically built from a set of subprograms that have all been planned and written together, de novo.

- The combining forms used to put existing program fragments together to form larger ones do not have good mathematical properties [they do not obey a strong set of algebraic laws]. Further, the most powerful of these, **while-do** and its analogs, lack the most important property of good combining forms: if you know what programs p and q do, then you should know in the most simple and direct way what **combine**(p,q) does.

- Programs are concerned with a low level of detail and with their "environment". Programming is still primarily the art of combining words to form a single new word and then arranging to repeat this process [a really "loopy" way to work!]. This low level style encourages a concern with computer efficiency that is humanly very inefficient. [Instead of relying on general theorems to transform inefficient high level programs into more complex efficient ones.]

As a result of these defects programming has made little progress toward becoming an efficient engineering discipline over the last quarter century. Their combined effect is that programming is a terribly repetitive business, a black art. Beyond principles and techniques we have very few things in our profession of lasting usefulness: few reusable programs, few general theorems [and the statement of these is usually so complex that it is difficult to recognize an instance of one]. For a further discussion of the problems flowing from the von Neumann concept of "program" see [Backus 81a].

One example of this lack of generality and reusability is our treatment of "housekeeping operations". These consist of "control structures" to repeat a set of statements, together with certain index variables scattered throughout those statements [thus a housekeeping operation is not just a "for statement" but includes all the occurrences of the index variable]. As a result, these ubiquitous operations in our work are not reusable and compact operations, but comprise bits of information scattered over a program text. Thus housekeeping operations, which lie at the center of our business, cannot be defined, cannot be reused, cannot be transferred from one program to another, but must always be drearily redeveloped from scratch in each new program.

The area of reasoning about and transforming programs represents another example of the lack of generality in conventional programming. There are a great many papers in our literature giving examples of transformations and correctness proofs. Nevertheless, we still seem to be a long way from having a powerful and generally accepted body of knowledge that really simplifies the task of reasoning about programs, even very ordinary kinds of programs. We have few general theorems that are immediately useful to a programmer for transforming his program or proving it correct. [Although this situation is beginning to improve, at least in the area of functional programming.] What we do have is a large collection of principles that one may struggle to apply to an individual program; thus, for example, [Manna 74] contains a number of important theorems that justify various kinds of inductive proofs.

In the absence of general theorems, some authors propose that programs be developed starting from a predicate that is to be true when the program ends. The development process is illustrated by many examples and principles that may help one to develop a desired program. This approach is described in David Gries' book on structured programming, The Science of Programming. See also [Dijkstra 76, Hoare 69, Mills 75].

Structured programming is quite successful [it gives many helpful techniques to try, and deals with many difficult and subtle problems], but, like all conventional programming, it is demanding and infuriatingly repetitive -- with practice, one acquires technique but no lasting general results -- each problem is a new challenge. It is as if we were to solve many quadratic equations, treating each as a new problem and cleverly using various factoring techniques -- instead of solving the general equation once and for all, and then applying that solution to each equation.

1.2 Problems with functional, "object level" concepts of "program"

There has been a movement in recent years to work with functional programs [that map objects into objects, rather than stores into stores]. This approach has many advan-

tages, although it has to overcome a number of problems before it becomes the obvious choice for everyday use. Even here, however, the most common notion of program is the "object level" one and this has problems. It is based on the use of variables that range over "objects" [the things programs transform]. In "object level" programming, building a new program from given ones involves applying the given programs to objects or object variables until the "result object" is built; the new program is then obtained by abstracting the object variables. Here are some of the problems:

- Unlike conventional programs, which use standard combining forms [e.g., composition, if-then-else, while-do], these programs tend to rely on lambda abstraction as the principal tool for building programs. Lambda abstraction is not a combining form. It combines a variable and an expression [with free variables] to form either a program [=function] or another expression. Its properties are syntactic rather than algebraic. Thus it combines two entities, neither of which is a program, to form a program; it fails to build programs from simpler ones and it fails to provide an algebra of programs on which to base general results about programs.

- Since these programs are not built by combining forms, reasoning about them concerns operations on objects and the mathematics of objects. This leads to theorems about objects but does not easily lead to general theorems about programs.

To have general theorems about programs one needs programs that are built by applying operations to existing programs, and these operations [combining forms, functionals, program-forming operations] must have attractive algebraic properties. This is the fundamental structure of many mathematical systems. Why are we in computer science so reluctant to build our basic entity, the program, in this way?

As in the case of conventional programs, work with object level programs has produced few general results, for the reasons sketched above. For example, one promising approach to transforming object level programs is that of [Burstall & Darlington 77]. Its authors give a set of transformation rules for transforming recursively defined functions at the object level. But again, as in the structured programming approach, there are few general results; instead there are lots of examples and strategies. These may be helpful when one is confronted with a new problem, but a lot of ingenuity is needed to define the right auxiliary function [the "eureka" step in the process]. Some powerful results are obtained by this approach. The problem is to try to make them more general and reusable.

1.3 Function level programs

The function level view of programs is one in which programs are constructed from existing ones by the use of program-forming operations (PFOs), operations that have two properties:

- If G(f,g) is a program built from programs f and g by the program-forming operation G, and if one knows how f and g behave, then one knows very simply and directly how the new program G(f,g) behaves.

- There is a powerful set of algebraic laws relating PFOs, particularly those that equate an expression in which one PFO is the main connective with another expression in which a different PFO is the main connective.

This way of building things -- by the application of operations that a) produce clearly understood results, and b) have strong algebraic properties -- distinguishes many of our most useful mathematical systems. The function level view asks that programs be built this way, that programming, by adopting this concept of program, become a mathematical system, the kind of system that history has proven to be extremely useful.

The function level approach seeks to bring this mathematical viewpoint to programming -- one that has heretofore been lacking -- just as the development of abstract data types has brought mathematics to bear on the subject of types. Originally we thought of data types as sets and we were concerned with the structure [or representation] of the objects belonging to a given type. Later we understood that we were really more interested in the abstract properties of the objects, not their structure or representation. And these properties are given by the algebraic structure of the operations on the objects, as determined by the laws they are required to satisfy. It is these algebraic laws that enable us to reason about data types, to prove general theorems about any type whose operations obey certain laws.

The function level view of programs seeks to do for programming what abstract data types have done for types, what algebra has done for arithmetic. It seeks to focus our attention on the operations on programs and on their algebraic structure, rather than on the structure and representation of the programs themselves. It seeks to have us view a program as the result of an operation, not as a syntactic entity whose structure is determined by punctuation -- semicolons and **begin - ends**, nor as a predicate transformer, nor as the syntactic result of substituting an argument for a variable in some expression.

In addition to their own algebraic structure, the program-forming operations [= combining forms = functionals] give a hierarchical structure to programs built with them. The operation that constructs a program is its main connective. Each of the programs put together by the main connective is itself built by various secondary connectives, and so on down, giving a tree structure to the entire program, with program-forming operations at the nodes and its constituent programs at the leaves. The importance of structure in conventional programs has long been recognized; we should now become aware of its importance in functional ones.

Just as we can now prove general theorems about objects by utilizing the algebraic properties of the operations on those objects [e.g., prove that all quadratic equations have a general solution], so we can prove general theorems about programs by the same means. But to do so, programs must be the result of operations, operations that have desirable algebraic properties. This is exactly what the function level view demands.

In contrast to this goal, the program-forming operations (PFOs) of von Neumann programs do not have desirable algebraic properties, and object level functional programs are not constructed by PFOs at all.

In summary, the fundamental goal of the function level view is to emphasize the mathematical structure of programs rather than the textual one. This view also offers solutions to the other difficulties discussed above, with the following possibilities:

1) The ability to construct a new program from off-the-shelf ones, since functional programs do not depend on storage plans.

2) Combining forms such that, if you understand the programs that are to be combined, then you very easily understand the resulting combined program.

3) Programs that are not concerned with low level details or with the names, size, or storage arrangement of their operands.

4) The ability to use powerful algebraic laws and theorems to convert simple but inefficient programs into complex, efficient ones. This can lessen the need to write efficient programs. [Our concern with efficiency is a major cause of inefficiency in programming.]

5) General "housekeeping" operations [data rearrangements] that can be defined and used over and over again.

6) A technology for reasoning about and transforming programs in which general theorems can be proved once and for all and easily applied in practice.

In this paper we hope to give indications for the validity of some of the above advantages of the function level view. We exhibit and use combining forms with the claimed properties; we use reusable, compact housekeeping operations. We exhibit compact programs that have a great disregard for efficiency and then turn them into very efficient ones. By developing a general result and applying it in examples, by examining the general results of others and restating them in function level form, and by using function level equations to optimize function level programs, we hope to show that this style offers a uniform, helpful technology for reasoning about, transforming, and optimizing programs.

1.4 Outline of the paper

Section 2 offers a brief review of the FP language [Backus 78, 81b] and some examples of function level equations for its primitives and combining forms that might make up a semantic description. Section 3 reviews some earlier general results, then develops some new ones leading to a moderately general recursion removal theorem. Section 4 presents a survey of examples of general results by some authors and their application, and exhibits them in function level form.

Section 5 presents a strategy for optimizing certain kinds of FP programs. It introduces some new "constructs" into FP -- pure functions with obvious counterparts as conventional programs -- and some equations involving them. It shows how these can be used to form identity functions for the arguments of functions to be optimized -- identity functions that describe the "shape" of those arguments. And it shows the use of these constructs and equations in optimizing some examples. These examples show how inefficient FP "housekeeping" operations, copying, and the computation of intermediate results can be eliminated by a relatively simple strategy to produce programs that run as fast as Fortran programs.

Section 6 presents some conclusions.

2. Review and semantics of FP

2.1 Review of FP

Readers familiar with FP may skip this section or use it as a reference for understanding the examples of later sections.

We adopt most of the notation and primitives of the FP system of [Backus 78] -- the few deviations are noted below with "(*)". Thus the set O of objects contains $\perp$

["bottom"], the set of atoms, the empty sequence <>, and the sequence $\langle x_1,\ldots,x_n\rangle$ of non-bottom objects x_i [where $\langle\ldots,\perp,\ldots\rangle = \perp$]. The set of atoms typically include numbers, symbols or identifiers, and T and F for "true" and "false".

An object expression is either an object or a sequence of object expressions or an application F:E, where F is a function expression and E is an object expression.

A function f or respectively, a [function expression] is either

a) a primitive function [primitive function symbol], or

b) $G(f_1,\ldots,f_n)$, where G is one of the combining forms {composition, condition, construction, left insert, right insert, apply-to-all}, the f_i are functions [function expressions], and n corresponds to the arity of G, or

c) $\bar{}x$, for some object x [object expression], where $\bar{}x$ is the function that is everywhere x except at $\perp$. Such a function is called a constant function. Or,

d) sel(n) or selr(n), for some integer [object expression with integer value] $n\geq 1$. Such functions are called selector functions and select the nth element of a sequence starting from the left end -- sel(n) -- or the right end -- selr(n)]. When it is clear that a selector function is intended, we write, e.g., "3" in place of "sel(3)". Or,

e) a defined function [defined function symbol], for which a unique definition exists [see definitions below].

A function definition has the form

def f = E

where f is an unused function symbol and E is a function expression that may involve f. [To see that the strict, bottom-up semantics of FP constitute a safe computation rule to compute the least fixed point of this recursive equation, refer to [Williams 82].] We may also give "extended" definitions [see 2.3 below for their description].

Each function is strict and maps a single object into a single object. We write f:x for the result of applying f to x. A function f is defined at x if $f{:}x \neq \perp$.

In the following examples of primitive FP functions and combining forms, we give object level descriptions of them so the reader may have an informal understanding of their behavior without first having to become familiar with the function level style. However, the function level equations of the next section indicate the way we propose to formally describe their semantics. [But we shall only indicate this in this paper.]

2.1.1 Examples of primitive FP functions. Here we give some FP primitive functions with brief explanations. These descriptions are merely to indicate the "object level" effect of these functions, their semantics are ultimately to be given by "function level" equations, some of which appear in 2.5 - 2.8 below.

Function	Description, examples
1,2.. 1r,2r..	Selector functions. 1:<A,B,C> = A, 2:<A,B,C> = B, 2:<A> = $\perp$, 1r:<A,B,C> = C, 2r:<A,B,C,D> = C [We use the integers to represent selector functions when there is no confusion about whether we mean a function or an integer; when there is, we use the PFO sel that converts an integer n into the corresponding selector function. Thus in the above we could write sel(n):x rather than n:x.]
tl, tlr	Tail and tail right. tl:<A,B,C> = <B,C>, tlr:<A,B,C> = <A,B>
al, ar	Append left and append right. al:<A,<B,C>> = <A,B,C>, ar:<<A,B>,C> = <A,B,C>
distl, distr	distribute left, distribute right. distl:<A,<B,C>>=<<A,B>,<A,C>>, distr:<<A,B>,C>=<<A,C>,<B,C>>
trans	Transpose. trans:<<a,b>,<c,d>,<e,f>> = <<a,c,e>,<b,d,f>>
id	Identity. id:x = x for all objects x.
+, -, *, /	The arithmetic functions. +:<2,3> = 5, *:<3,5> = 15, etc.
addl, subl	addl:3 = 4, subl:7 = 6
and, not, eq	and:<T,T> = T, and:<T,F> = F, not:F = T, eq:<A,<B>> = F
atom	atom:A = T, atom:F = T, atom:<A,B> = F, atom:3.14 = T (defined for every non-bottom object)
null	null:<> = T, null:<A,B> = F, null:A = $\perp$ (defined only on sequences)

2.1.2 Examples of function expressions using combining forms. Here are some function expressions involving functions p, f, g, $f_1,\ldots,f_n$ showing the notation for the combining forms and the functions they build. In these examples, if no value is given for a function f at some object x, then f:x = $\perp$.

Expression	Description, examples
f•g	**Composition** of f and g. f•g:x = f:(g:x)
p -> f; g	**Condition** of p [predicate], f and g. Read "if p then f else g". (p -> f; g):x = f:x if p:x = T = g:x if p:x = F We write p -> f; q -> g; h for p -> f; (q -> g; h) etc.
$[f_1,\ldots,f_n]$	**Construction** of $f_1,\ldots,f_n$.

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

$[]:x = \langle\rangle$

/f **Right insert** of f. (*)

$/f:\langle x,y\rangle = y$ if $x = \langle\rangle$ (*)

$= f:\langle 1:x,\ /f\cdot\langle tl:x,y\rangle\rangle$ if $x = \langle x_1,\ldots,x_n\rangle$ (*)

\f **Left insert** of f (*)

$\backslash f:\langle x,y\rangle = y$ if $x = \langle\rangle$ (*)

$= f:\langle\backslash f:\langle tlr:x,y\rangle,\ lr:x\rangle$ if $x = \langle x_1,\ldots,x_n\rangle$ (*)

@f **Apply-to-all** of f.

$@f:\langle\rangle = \langle\rangle$

$@f:\langle x_1,\ldots,x_n\rangle = \langle f:x_1,\ldots,f:x_n\rangle$

(a f b) Infix notation for $f\cdot[a,b]$. $f + g = +\cdot[f,g]$

2.2 Specification of semantics: object level vs. function level

The semantics of the original FP language [Backus 78] was specified at the object level. That is, for each function denoted by an expression E and each object x the result of applying E to x [the object E:x] was specified by a collection of rules. From these object level semantics a set of function level laws was derived that expressed various algebraic properties of the primitive functions and of the combining forms [= functionals = PFOs].

Instead, one could have specified the function level laws relating the primitive functions and PFOs. For example, to partly describe the semantics of append left [al], instead of giving the object level equation: for all objects x, $y_1,\ldots,y_n$,

$$al:\langle x,\langle y_1,\ldots,y_n\rangle\rangle = \langle x,y_1,\ldots,y_n\rangle ,$$

that expresses the equality of two objects, one could give the corresponding "lifted" function level equation: for all functions x, $y_1,\ldots,y_n$

$$al\cdot[x,[y_1,\ldots,_n]] = [x,y_1,\ldots,y_n] ,$$

that expresses the equality of two functions. Not only can one obtain from this the object level equation, but its function level form is very useful in reasoning about programs: one has a general law in which one can substitute any functions for the variables of the law and then replace the resulting function expression on the left by that on the right [or vice versa]. On the other hand, the object level version is useful only for reasoning about objects.

2.3 Extended definitions in FP

Many people find it hard to read "proper" FP definitions like the following one for an iterative factorial function f, where factorial = $f\cdot[id,\sim 1]$,

7) f = eq0•1 -> 2; f•[sub1•1, *•[1,2]]

whereas they have no difficulty with the object level equation

8) f:<x,y> = eq0:x => y; f:<sub1:x, *:<x,y>>

which is to hold for all objects x and y [here "=>" denotes the object level operation if-then-else]. To make function level equations as easy to read as (8), object level equations can always be "lifted" [Backus 81c] to corresponding function level ones by doing the following

a) Change all object variables to function variables, thus x will range over all functions instead of all objects.
b) Change all pointed brackets < >, to square ones [ ].
c) Change all applications to compositions, i.e. change ":" to "•".
d) Change => to ->.

If we do this to (8) we get the function level equation that holds for all functions x and y

9) f•[x,y] = eq0•x -> y; f•[sub1•x, *•[x,y]]

that serves as readably to define f as (8) and has the additional advantage that it is a general law about the defined function f. We will call such a definition lifted from the object level an extended definition [Backus 81b] and use them freely. If we note that [1,2] is the identity on pairs, then for pairs, by substitution for x and y we get

f = f•[1,2] = eq0•1 -> 2; f•[sub1•1, *•[1,2]]

which is the original proper definition (7). We have found that extended definitions are by far the best form to use in defining a function, for readability, whereas proper definitions are the best to use for reasoning about [transforming, calculating with] the defined function. We will make use of this fact later.

2.4 FP semantics: soundness and completeness

There is the question of the soundness and completeness of any semantic description of a language. This question has not been completely answered for any treatment of the semantics of the FP language of [Backus 78], however this has now been done [Halpern, Williams, Wimmers, Winkler 85] for an FP system [let me call it FP_{84}] much richer than the one we deal with here. The FP system we treat here is almost identical to that described in [Backus 78]: all functions are strict, the sequence constructor is strict, all sequences are of finite length, and there are no functions of higher type than functionals.

None of the above properties hold for FP_{84}. That is, FP_{84} is an untyped system

in which functions at all levels exist and are first class entities of the system: sequences of functions can be formed and other functions can be applied to them, thus "apply:<f,g>" is a meaningful expression in FP_{84}, whereas it is not in FP. Infinite sequences are admissible constructions.

Halpern, Williams, Wimmers, Winkler give the operational semantics of FP_{84} in terms of rewrite rules. They also define an independent denotational semantics. They prove that the rewrite rules preserve the denotational "meaning" of an expression [soundness]. They then prove that if the "meaning" of an expression is a finite object [the "meaning" of an expression may be a function or an infinite sequence], then the rewrite rules are sufficient to reduce the expression to that object [completeness].

2.5 The function level algebraic semantics of FP

Our object here is to give some examples of the kind of equations that might make up an algebraic description of the semantics of FP. The following equations do not constitute a complete description; the object of a complete description would be to give a practically useful set of equations, some of which might be redundant, as may be the case here. Some of the following equations will be used to derive propositions of the next section, others only illustrate the kind of equations that could be used.

2.6 Preliminaries

In what follows all variables will range over the space of FP functions [except in the form ~x, where x ranges over objects, or sel(n) or selr(n), where n ranges over integers, a subset of the objects] and are universally quantified. An equation may be qualified; thus we write

$$p \rightarrow\rightarrow f = g$$

to mean that, for all objects x, f:x is the same object as g:x provided that p:x = T. For example, if p:x = T and f:x = $\perp$, then the above asserts that g:x = $\perp$ ["=" denotes strong equality]. If p:x is anything but T, even undefined (=$\perp$), then the above makes no assertion.

We shall need the function "def" to qualify equations, where

$$def = \sim T.$$

Since every function is strict, def·f yields T in exactly the domain in which f is defined, and $\perp$ elsewhere.

2.7 Semantics of primitive functions, examples of function level equations

selectors, append left, and tail

1) $al \cdot [x,[]] = [x]$

2) $al \cdot [x,[y_1,\ldots,y_n]] = [x,y_1,\ldots,y_n]$

3) $def \cdot [x_2,\ldots,x_n] \to\!\!\to 1 \cdot [x_1,\ldots,x_n] = x_1$

4) $def \cdot x_1 \to\!\!\to tl \cdot [x_1,\ldots,x_n] = [x_2,\ldots,x_n]$ if $n \geq 2$
$= []$ if $n=1$

5) $ar \cdot [[],x] = [x]$

6) $ar \cdot [[x_1,\ldots,x_n],y] = [x_1,\ldots,x_n,y]$

7) $def \cdot [x_1,\ldots,x_{n-1}] \to\!\!\to lr \cdot [x_1,\ldots,x_n] = x_n$

8) $def \cdot x_n \to\!\!\to tlr \cdot [x_1,\ldots,x_n] = [x_1,\ldots,x^{n-1}]$ if $n \geq 2$
$= []$ if $n=1$

9) $not \cdot null \to\!\!\to al \cdot [1,tl] = id$

10) $not \cdot null \to\!\!\to ar \cdot [tlr,lr] = id$

11) $f^n = f \cdot f^{n-1}$ for $n \geq 1$ $f^0 = id$

12) $sel(n) = 1 \cdot tl^{n-1}$, $selr(n) = lr \cdot tlr^{n-1}$ for $n \geq 1$

Identity

13) $id \cdot x = x \cdot id = x$

Constant functions

14) $def \cdot y \to\!\!\to \bar{}x \cdot y = \bar{}x$

Atom

15) $atom \cdot \bar{}a = \bar{}T$ for each atom a (this represents an infinite set of equations, one for each atom)

16) $atom \cdot [] = \bar{}F$

17) $def \cdot al \cdot [x,y] \to\!\!\to atom \cdot al \cdot [x,y] = \bar{}F$

null

18) $null \cdot [] = \bar{}T$

19) $def \cdot al \cdot [x,y] \to\!\!\to null \cdot (al \cdot [x,y]) = \bar{}F$

length.

20) $len \cdot x = null \cdot x \to \bar{}0;\ (\bar{}1 + len \cdot tl \cdot x)$

2.8 Semantics of the combining forms, examples of function level equations

21) $f \cdot (g \cdot h) = (f \cdot g) \cdot h$

22) $f \cdot (p \to q;r) = p \to f \cdot q;\ f \cdot r$

23) $(p \to q;r) \cdot f = p \cdot f \to q \cdot f;\ r \cdot f$

24) $def \cdot g \to\!\!\to [] \cdot g = []$

25) $[f_1,\ldots,f_n] \cdot g = [f_1 \cdot g,\ldots,f_n \cdot g]$

26) $[\ldots(p \to f;g)\ldots] = p \to [\ldots f\ldots];[\ldots g\ldots]$

27) /f•[x,y] = null•x -> y; f•[1•x, /f•[tl•x,y]]

28) \f•[x,y] = null•y -> x; f•[\f•[x,tlr•y], lr•y]

3. Some general function level results about programs

We summarize some general results from earlier papers [Backus 78, 81b] and derive a few others that we shall use in the examples to follow. The machinery outlined below for transforming and reasoning about programs has a somewhat mechanical character, involving a good deal of routine calculations using function level identities. However, we believe these techniques provide a uniform, accurate technology for building up a set of general results in a form that makes it easy to identify and apply them, either mechanically or by people.

3.1 Some simple general results

Here we state without proof a number of simple equations and results that can be derived from the algebraic semantics.

29) /f•[[x_1,...,x_n], f•[y,z]] = /f[[x_1,...,x_n,y], z]

30) \f•[f•[y,z], [x_1,...,x_n]] = \f•[y, [z,x_1,...,x_n]]

31) if h is associative and has unit u, then /h•[id,~u] = \h•[~u,id]

32) p -> q; p -> r; s = p -> q; s

3.2 The general solution of linear functional equations

The following linear expansion theorem and related facts about linearity are often generally useful for reasoning about recursively defined functions, as we shall see in later examples.

3.2.1 Definition of linear forms and equations. An equation of the form

f = p -> q; Hf

is linear if the expression or form Hf in the variable f is linear. A form Hf is linear if there exists a form $H_t p$ such that the following two conditions hold:

L1) For all functions a, b, and c,

H(a -> b;c) = H_ta -> Hb; Hc

L2) For all objects x, if H~⊥:x ≠ ⊥, then, for all functions a,

H_ta:x = T .

H_t is the predicate transformer of H. Clearly Hf is linear with H_t if H and H_t satisfy (L1) and H~⊥ = ~⊥.

3.2.2 Linear expansion theorem. If Hf is a linear form, then the [least] solution of

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

is

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

where the infinite condition on the right denotes the function that is the limit of the following increasingly defined functions:

$$f_i = p \to q;\ldots;H_t^i p \to H^i q;\ \sim\bot$$

and where $H^{n+1}f$ denotes $H(H^n f)$. Note that the distribution rules for condition apply to infinite conditions, in particular:

33) $(p_1 \to q_1;\ldots;p_i \to q_i;\ldots)\cdot g = (p_1\cdot g \to q_1\cdot g;\ldots;p_i\cdot g \to q_i\cdot g;\ldots)$

This extension of the rule (23) for finite conditions depends on the continuity of $Hf = f\cdot g$, so that $H(\lim f_i) = \lim H(f_i)$; see [Williams 82].

3.2.3 Composition of forms. If H and G are two forms [function expressions each with one free variable], then by their composition HG we mean the form such that $HGf = H(Gf)$ = the form obtained by replacing the free variable of H by the form Gf [G with its free variable replaced by the function or variable f]. Take care not to confuse HGf [form composition] with $Hf\cdot Gf$ [function composition].

3.2.4 Theorem: composition of linear forms. If H and G are linear, then HG [where $HGf = H(Gf)$] is linear with predicate transformer $(HG)_t = H_t G_t$.

3.2.5 Theorem: basic linear forms. In the following function expressions we use Roman letters for the free variable of a form and bold-faced letters for arbitrary fixed functions. The following forms Hf are linear in f with the given predicate transformer $H_t p$:

$Hf = \mathbf{r}$	with $H_t p = \sim T$
$Hf = \mathbf{r}\cdot f$	with $H_t p = p = Ip$
$Hf = f\cdot\mathbf{r}$	with $H_t p = p\cdot\mathbf{r} = Hp$
$Hf = [\mathbf{g},f]$	with $H_t p = p = Ip$
$Hf = [\ldots,f,\ldots]$	with $H_t p = p = Ip$
$Hf = \mathbf{p} \to \mathbf{q};f$	with $H_t r = \mathbf{p} \to \sim T;r$
$Hf = \mathbf{p} \to f;\mathbf{q}$	with $H_t r = \mathbf{p} \to r;\sim T$
$Hf = f \to \mathbf{g};\mathbf{h}$	with $H_t p = p = Ip$

Applying the composition theorem to the above basic forms generates a great many linear forms and their predicate transformers. Note that all forms Hf with a single occurrence of f and built with composition, construction, and condition, are linear. Thus, for example, $Hf = \mathbf{h}\cdot[\mathbf{i},f\cdot\mathbf{j}]$ is linear with $H_t p = p\cdot\mathbf{j}$, since H = RST where

Rf = h•f, Sf = [i,f], Tf = f•j, and $H_t p = R_t S_t T_t p = II(p•j) = p•j$. Many forms Hf with multiple occurrences of f are linear. For example,

Hf = s -> f•k; h•[i,f•j]

is linear with $H_t r$ = s -> r•k; r•j. Of course the usefulness of the linear expansion theorem to solve f = p -> q;Hf depends on being able to deal with the expressions $H_t^n p$ and $H^n q$; in this case these expressions become unmanageable unless the values of p, k, h, i, j allow simplifications, as may happen.

For some further results on linear forms see [Backus 81b] and for results on nonlinear equations see [Williams 82].

3.3 Application of the linear expansion theorem: recursion removal

We use the linear expansion theorem to obtain a moderately general result that provides an iterative solution for a class of recursively defined functions. Similar techniques could be used to obtain similar results for other, more general recursive equations. Stronger results can be found in [Kieburtz and Shultis 81]; we present this one to illustrate a different method of proof with some useful intermediate facts. One of these is corollary 3.3.1, which provides useful data for reasoning about a larger class of functions.

3.3.1 Corollary of the linear expansion theorem. Let f satisfy the following

34) f = p -> q; h•[i,f•j]

Then

35) $p•j^n$ -> /h•[[i,i•j,...,$i•j^{n-1}$],$q•j^n$]

is the general term of the expansion for f.

3.3.2 Corollary of the linear expansion theorem. Let f′ satisfy the following for all functions x and y:

36) f′•[x,y] = p•x -> h•[y,q•x]; f′•[j•x, h•[y,i•x]]

Then the general term of the expansion for f′ that is valid for pairs and $n \geq 1$ is

37) $p•j^n$•1 -> \h•[2, [i,i•j,...,$i•j^{n-1}$,$q•j^n$]•1]

3.3.3 Recursion removal theorem. Given

38) f = p -> q; h•[i, f•j]
39) f′•[x,y] = p•x -> h•[y,q•x]; f′•[j•x, h•[y, i•x]]

where h is associative with unit u, then

40) $f = f' \cdot [id,\tilde{}u]$

Proof. From (37) we get the general term of the expansion for f' on pairs and, using the law (33) to distribute [id,~u] over the expansion for f' from the right, we get the general term of the expansion for $f' \cdot [id,\tilde{}u']$ and progressively transform it:

$p \cdot j^n \cdot 1 \cdot [id,\tilde{}u] \rightarrow \backslash h \cdot [2, [i, i \cdot j, \ldots, i \cdot j^{n-1}, q \cdot j^n] \cdot 1\ ] \cdot [id,\tilde{}u]$

41) $p \cdot j^n \rightarrow \backslash h \cdot [\tilde{}u, [i, i \cdot j, \ldots, i \cdot j^{n-1}, q \cdot j^n]]$ by (3) (12) (13) (25)

42) $p \cdot j^n \rightarrow /h \cdot [[i, i \cdot j, \ldots, i \cdot j^{n-1}, q \cdot j^n], \tilde{}u]$ by (31) since h associative

43) $p \cdot j^n \rightarrow /h \cdot [[i, i \cdot j, \ldots, i \cdot j^{n-1}], h \cdot [q \cdot j^n, \tilde{}u]]$ by (29)

44) $p \cdot j^n \rightarrow /h \cdot [[i, i \cdot j, \ldots, i \cdot j^{n-1}], q \cdot j^n]$ since u is a unit of h

This last is the general term of the expansion of f. The initial term of $f \cdot [id,\tilde{}u]$ is, by (39), $p \rightarrow h \cdot [\tilde{}u, q]$, which reduces to $p \rightarrow q$. Therefore the expansions of f and of $f' \cdot [id,\tilde{}u]$ are equal, therefore the functions are equal. Q.E.D.

4. The use of general theorems to obtain program transformations

Here we compare the use of general theorems in the approach of various authors. Let me first make a general comment about those that present their results in the form of object level equations. To take an example from [Wadler 81]: he gives the object level rule

```
(map f) (map g) xs
   => (map h) xs
        where   h x = f g x
```

This is exactly the FP law III.4 of [Backus 78]

$@f \cdot @g = @(f \cdot g)$

The point here is that the FP function level equation allows one to immediately substitute one side of the equation for the other in any function expression, whereas the object level statement of the principle is much less directly useful and requires the use of a lot of unnecessary indirection, as in the where clause that is required by failure to use the combining form composition. For this reason I submit that it is time we recognize that such principles are essentially function level ones and should uniformly be given as such.

4.1 The approach of [Kieburtz and Shultis 81] = K & S

This paper contains some good examples of the function level approach we are advocating and has a number of results that go beyond the simple recursion removal

theorem presented here. It uses the FP function level technology to derive a general theorem [Prop 2] of which our recursion removal theorem is a corollary. We present our proof only because it represents a quite different approach to obtaining such results.

Their proof starts from the most basic function level identities and involves two inductive arguments, whereas ours relies on the linear expansion theorem, which was helpful in obtaining the formulation of the theorem by examining the general terms of the expansion for the given function f. Their proof is shorter and cleaner, ours involves some mechanical calculation that is tedious but easy.

They also prove several other interesting and generally useful theorems concerning recursion removal for the scheme f = p -> q; h•[i,f•j] of our theorem. They go on to prove several general theorems [props 5 & 6] for removing recursion from non-linear functions of the forms

f = p -> q; h•[f•r, f•s] and

f = p -> q; h•[r,h•[f•s, f•t]]

It is interesting to note that the basic scheme for the solutions of these last two questions came, according to the authors, from [Cohen 79]. However the function level presentation here of the result seems to make it much shorter and more accessible than in its more general presentation in terms of conventional programs in [Cohen 79].

This paper provides some of the most directly useful theorems for recursion removal that I am aware of. And they are presented as function level results that can be straightforwardly applied, rather than as principles and methods for obtaining results, or as lengthy, difficult to recognize, object level schemes.

4.2 The approach of [Burstall and Darlington 77] = B & D

It is interesting to examine the differences between the algebraic approach to program transformation and that of B & D, who originated the basic ideas about transforming recursively defined functions. Their approach is to provide a set of transformation rules to be applied to the transformation of an individual function or set of functions. This involves defining an auxiliary function in terms of the given one [the "eureka" step], then instantiating object variables, unfolding, simplification and folding; the object being to get a recursive expression for the defined function that does not depend on the given function, and then redefine the given one in terms of the new one. The goal is to define the new function so that the resulting redefinition of the given one is more efficient.

The approach we advocate is to develop general function level identities and theorems. The goal is to provide a basic uniform technology for transforming programs in a mechanical way that captures, once and for all, a lot of difficult details and reasoning, and to develop general theorems that can be easily applied to transform many different programs. Our methodology is just that of substitution in equations and needs no special rules. The above theorems and those of K & S and others are easier to apply [when they apply] than B & D's methods, and they preserve termination properties, whereas B & D's do not.

The most difficult programs to reason about are recursively defined ones. For linear recursive programs and for certain non-linear ones [Williams 82] the FP technology provides non-recursive expansions that often help one to understand and reason about such programs. Many programs that are normally defined recursively are easier to define in FP in a simple closed form in which one can reason about them directly using the FP technology. We have given above an example of a general theorem for transforming a class of linear recursive functions into iterative ones, K & S give several more plus two for classes of non-linear equations.

B & D's methods apply to many programs to which our theorems above do not. However, many of those programs may be handled by the results of K & S discussed above.

4.2.1 Example 1. Factorial. Given the definition of factorial

```
fact = eq0 -> ~1; *•[id, fact•sub1]
```

we see that the recursion removal theorem applies since * is associative with unit 1. Thus in

```
f'•[x,y] = p•x -> h•[y,q•x]; f'•[j•x, h•[y, i•x]]
```

we need only replace p by eq0, h by *, q by ~1, i by id, and j by sub1 to give

```
f'•[x,y] = eq0•x -> *•[y,~1•x]; f'•[sub1•x, *•[y, id•x]]
f'•[x,y] = eq0•x -> y; f'•[sub1•x, *•[y,x]]
```

and fact = f'•[id,~1], which is essentially the same program derived by B & D. Their derivation of this example required some ingenuity and work, whereas our effort went into developing a general theorem which can be routinely applied to this example as well as many others.

In the FP programming style we would prefer to define

```
fact = /* • iota ,
```

a non-recursive definition using the iota function of APL, where

```
iota:n = <1,2,...,n> .
```

4.2.2 Example 2. Reverse. When reverse is defined using concatenate, as it is in B & D, it fits exactly the same scheme as factorial, since ++ is associative with unit []. Thus it can be treated exactly as above to obtain B & D's result. If we define reverse, using append left, as

rev = null -> []; al•[1r, rev•tlr] ,

then Proposition 2 of [Kieburtz and Shultis 81] applies directly to provide an iterative solution that uses append right.

4.3 The approach of [Bird 84]

This is another paper that presents some general function level results. However it uses a somewhat idiosyncratic mixture of function and object level equations, curried functions, infix functions, "sections" and other special notation that makes for terse expressions but very difficult reading. One annoying feature of the use of curried functions is that it is difficult to determine the arity of a function in an expression unless the expression is object level, and even then it requires some analysis. I believe the use of uncurried functions promotes readability and eliminates the need for object level equations.

4.3.1 Example. The "promotion" scheme. We give the example first in Bird's terms [beginning with his paragraph 3, p491], and then give his derivation in an uncurried, function level FP version.

(D1) spec x = f(Hx)

(D2) H[] = ...

H(a;x) = h a (Hx)

(C1) f•(ha) = (h'a)•f

The equations (D1) and (D2) represent a specification in which we shall think of H as building an object from a sequence [a;x denotes appending a to the sequence x] by combining [with h] the head and H of the tail of its argument. For efficiency it would be preferable to compute spec by using the existence of an h' satisfying (C1). Here is the FP derivation of Bird's result, but using an uncurried version, uh, for his curried two-argument function h, and uh' for his h':

(D1') spec = f•H

(D2') H = null -> r; uh•[1,H•tl] r is supplied here for the [] case.

(C1') f•uh•[x,y] = uh'•[x, f•y] for all functions x, and y.

Thus we have the following

spec = f•H

= f•(null -> r; uh•[1,H•tl]) def of H ["unfolding"]

= null -> f•r; f•uh•[1,H•tl) by (22)

= null -> f•r; uh'•[1, f•H•tl] by (C1')

= null -> f•r; uh'•[1, spec•tl] by (D1')

which is Bird's result in FP form. Here it is obtained at the function level. The pattern-matching style of SASL equations, in which conditions are broken up into multiple equations using mutually exclusive arguments, requires that the arguments always tag along in Bird's object level derivation.

If one applies the corollary 3.3.1 to this last equation for spec, and then simplifies the general term in its expansion,

$$\text{null}\bullet tl^{n} \rightarrow /uh'\bullet[[1, 1\bullet tl, \ldots, 1\bullet tl^{n-1}],\ f\bullet r\bullet tl^{n}]$$

$$= \quad (\text{len} = n) \rightarrow /uh'\bullet[id,\ f\bullet r\bullet[]\ ]$$

one then obtains the following closed form expression for spec

spec = /uh'•[id, f•r•[]]

which is easier to reason about and to use.

5. Optimizing FP programs

The function level style of FP replaces the scattered form of "housekeeping" operations found in conventional programs with operations that rearrange data structures. This means that complex housekeeping operations can be defined and reused in FP, whereas such operations in conventional programs must be redeveloped for each program from scratch. Thus we believe that the FP style of housekeeping operations is the right way to think about and write such operations. But the price one pays in efficiency is enormous if programs with these operations are executed literally.

In addition to those due to FP housekeeping operations, there are many other potential sources of inefficiencies in FP programs. One of these is the creation of large and unnecessary intermediate results. This general problem for functional programs is discussed in [Wadler 84] where the author deals with the problem not only at the object level but by means of a "listless machine". Here we propose to deal with this issue at the function level in source language. Eliminating making copies of large objects is an important subcase of the problem of eliminating the creation of intermediate results.

We propose to optimize FP programs having these problems into programs in an FP language extended with additional constructs [which we laughingly call "Fortran constructs"]. These constructs denote pure functions but have obvious realizations as efficient Fortran or other conventional programs. We hope to optimize most FP

programs that have potential efficient Fortran equivalents into those equivalents or something very close to them; that is, if an FP program is not essentially a "list processing" program, we hope to make it run as fast as a Fortran program. We hope to produce a Fortran-like program that addresses its data just as a Fortran program does, without using list structures or unnecessary levels of indirection in addressing.

The process of optimizing such FP programs into efficient extended FP programs will depend on the use of sophisticated strategies employing a collection of function level identities involving the Fortran constructs. The work on this approach is in an early stage, and it is not clear how far it can be carried nor how much of it can be automated. Much of its ultimate power and utility, if successful, will depend on:

a) Powerful strategies for employing functional identities or rewrite rules for transforming programs.

b) Systems for specifying programs and abstract data types within FP in terms of functional identities that are consistent with (a). These have yet to be completely developed, although an excellent basis already exists in [Guttag, Horning, Williams 81].

These goals are clearly difficult to achieve, but we believe they comprise an interesting effort to put together a practical, higher level functional style of programming. We have yet to study carefully the problems of optimizing programs that are recursively defined. We have some hope that some of the above machinery for linear programs will be helpful in this regard. For other programs some of the techniques of Burstall & Darlington should be helpful.

All we shall do here is to exhibit some examples and show how a given set of functional identities can be used to optimize them.

5.1 The basic optimizing strategy

Our basic strategy is the following. First, introduce some "Fortran constructs" into extended FP [which we will refer to as "constructs"]. These must have three properties:

a) any "Fortran-like" FP program can be represented as a construct.
b) any construct has an obvious Fortran-like program or similar counterpart that is directly efficiently realizable.
c) it is possible to write a construct that is the identity function for <u>all</u> the arguments of the function one wants to optimize.

Then, given such constructs as above, our strategy to optimize a program f is as

follows:

1) Find a construct, C, that is the identity for all the valid arguments for f. This construct we call a "structure operator" for f [because it essentially describes the structure of f's arguments]. Thus f•C = f. We will work with f•C.

2) If f = **G**(g,C') for some combining form **G**, then find an equation of the form **G**(g,C') = C''; if no such equation exists, that is, if g is too complex, try to find **H** and **K** so that f = **H**(r,**K**(s,C')) where s is simple enough that there is an equation **K**(s,C') = C'' and then continue in this way until you get f = C'''. In our examples so far, this strategy involves just **G** = **H** = **K** = composition. In this case the strategy is just, e.g.,

   If f = g•h and f = f•C, and if

   h•C = C' and g•C' = C'' are instantiations of established equations then

   f = g•h•C => f = g•C' => f = C''

   transforms f into the construct C''.

5.2 The **for** construct

Our most basic construct denotes a construction of a sequence of functions whose length may depend on the argument to which it is applied. The functions in the construction are generated by some expression E(i) that converts an integer i into a function E(i). We write the **for** construct using bold-faced square brackets as follows

45) [E(i) i=f,g]

where f and g are integer valued functions and i is a bound variable of the construct. The meaning of [E(i) i=f,g] is defined as follows for any object x: if $1 \leq f{:}x \leq g{:}x$, then

46) [E(i) i=f,g]:x = [E(f:x), E(f:x + 1),...,E(g:x)]:x

If f:x > g:x then

   [E(i) i=f,g]:x = <>

Thus, for example,

   [i i=~1,len]:<a,b,c> = [1,2,3]:<a,b,c> = <a,b,c>

Here E(i) = i (more properly we should write E(i) = sel(i), but here there is no doubt that the integer i is a selector function). So [i i=~1,len] is the identity function for every non-empty sequence.

Since a **for** construct denotes a construction, we will extend the definition of the **for** construct to include any construction whose elements are either functions or for constructs. Thus $[c_1,...,c_n]$ is a **for** construct whether or not some or all of

the c_i's are **for** constructs.

5.2.1 The Fortran-like program corresponding to a **for** construct. Every construction $[f_1,\ldots,f_n]$ has a corresponding Fortran-like program that will compute $[f_1,\ldots,f_n]:x$ -- given Fortran-like programs p(i,x) that will compute $f_i:x$ -- (in the case of Fortran we must assume that $f_i:x$ is a single word, in other languages we might not need this assumption). The program is as follows

```
for i = 1, n
  r[i] := p(i,x)
```

For a **for** construct [E(i) i=f,g], if p(E(i),x) is a program that will compute E(i):x for any integer i and any object x, then the program to compute [E(i) i=f,g]:x is

```
for i = f:x, g:x
  r[i] := p(E(i),x)
```

5.2.2 Nested for constructs. Consider the construct

```
[[E(i,j) i=f,g] j=r,s]
```

In the innermost construct the variable i is bound, so E'(j) = [E(i,j) i=f,g] is a legitimate expression in the free variable j that will produce a function for each integer j. For example

```
[[i•j i=~1,len•1] j=~1,len]
```

is the identity construct for every sequence of equal-length sequences [e.g., a rectangular matrix].

```
    [[i•j i=~1,len•1] j=~1,len]:<<a>,<b>>
=   [[i•1 i=~1,len•1], [i•2 i=~1,len•1]]:<<a>,<b>>
=   <[i•1 i=~1,len•1]:<<a>,<b>>, [i•2 i=~1,len•1]:<<a>,<b>>]
=   <[1•1]:<<a>,<b>>, [1•2]:<<a>,<b>> >
=   <<a>,<b>>
```

similarly

```
[[i•j i=~1,len•j] j=~1,len]
```

is the identity construct for every sequence of sequences.

5.3 The insertion construct

5.3.1 Simplified left and right insert for optimization. In the following and in the examples it is convenient to introduce a simplified notion for left and right insert, in which /f and \f are functions of one argument rather than two. We define these one argument versions in terms of the two argument ones used above as follows

[these one argument inserts are those used in the original FP]. We use bold-faced /f and \f for the two argument versions and /f and \f for the one argument one.

47) **def** /f = /f•[id,~u] if f has a right unit u
= /f•[tlr,lr] otherwise

48) **def** \f = \f•[~u,id] if f has a left unit u
= \f•[1,tl] otherwise

5.3.2 The insertion construct. Our second construct is the insertion construct. If C is a **for** construct, then /f•C and \f•C are insertion constructs.

5.3.3 Fortran-like programs corresponding to insertion constructs. The program to compute \f•[E(i) i=g,h]:x is

```
r := u_f                    where u_f is a left unit of f
for i = g:x, h:x
  r := f:<r, p(E(i),x)>
```

5.4 Algebraic laws for constructs

The following identities have the form f•C = C′ where C and C′ are **for** constructs and f is a primitive FP function.

49) trans•[[E(i,j) i=f,g] j=r,s] = [[E(i,j) j=r,s] i=f,g]

50) trans•[[$E_1(i_1)$ i_1=f,g],...,[$E_n(i_n)$ i_n=f,g]] = [[$E_1(i)$,...,$E_n(i)$] i=f,g]

51) distr•[[E(i) i=f,g], h] = [[E(i),h] i=f,g]

52) distl•[h, [E(i) i=f,g]] = [[h,E(i)] i=f,g]

53) @f•[E(i) i=g,h] = [f•E(i) i=g,h]

54) tl•[E(i) i=f,g] = [E(i) i=(f + ~1),g] |provided the left construct

55) tlr•[E(i) i=f,g] = [E(i) i=f,(g - ~1)] | does not produce <>.

These identities concern change of variables or bounds, distribution of a function from the right.

56) [E(i) i=f,g] = [E(j) j=f,g] [change bound variable]

57) [E(i) i=~n,f] = [E(i + ~(n-1)) i=~1, f - ~(n-1)]

58) [E(i) i=f,g] • h = [E(i)•h i=f•h,g•h]

There are many other possible identities, but these will suffice for our examples and indicate the possibilities. Note that they are easily derivable from the function level semantics of FP.

5.5 Examples of program optimization

5.5.1 Inner Product. The following FP program IP for inner product has the right form: it is concise and clear; it does not depend on the size of its arguments; it is non-recursive and therefore easy to reason about and transform. But its first operation is an inefficient housekeeping operation, transpose, which our optimization should eliminate.

def IP = \+ • @* • trans

Thus, for example

IP•[[a,b,c],[d,e,f]] = \+•@*•[[a,d],[b,e],[c,f]]
= (a*d) + (b*e) + (c*f) using infix function level notation

Now the structure operator for IP -- the identity construct for any proper argument of IP, a pair of equal-length sequences -- is the **for** construct

C = [[i•j i=~1,len•1] j=~1,~2]

We proceed to transform IP = IP•C by using the above rewrite rules

IP•C = \+•@*•trans•[[i•j i=~1,len•1] j=~1,~2]
= \+•@*•[[i•j j=~1,~2] i=~1,len•1] by (49)
= \+•[*•[i•j j=~1,~2] i=~1,len•1] by (53)
= \+•[*•[i•1,i•2] i=~1,len•1] by def of **for** construct

This last program corresponds directly to the following Fortran-like program for inner product for the argument <a,b>, where a and b are sequences of equal length.

```
r := 0                [the unit of +]
for i = 1, len:a
  r := r + (a[i] * b[i])
```

So our transformation has eliminated the need to perform transpose or to create the intermediate result @*•trans:<a,b> [provided we implement insertion constructs properly]. It has produced the standard conventional program. Using the same approach we could develop the general identity for IP•C:

59) IP•[[E(i) i=~1,f], [F(j) j=~1,f]] = \+•[*•[E(i),F(i)] i=~1,f]

5.5.2 Matrix multiplication. Here is the standard FP matrix multiplication program, MM, that will multiply any pair of conformable matrices, <A,B>, where each matrix is represented as the sequence of its rows.

def MM = @(@IP)•@distl•distr•[1,trans•2]

Again, this is a good program that concisely gives the essence of the operation, but it is very space and time inefficient. The first three operations of this non-recursive program are data-rearranging "housekeeping" operations; all of them should be eliminated by our transformation [and they are]. Our goal is to transform MM into

the following construct:

60) MM = [[\+•[*•[k•i•1, j•k•2] k=~1,len•2] j=~1,len•1•2]] i=~1,len•1]

The detailed transformation is given below. The construct (60) represents an efficient Fortran-like program. Let matrices A and B be given with

n_1 = len:A = no. of rows of A
n_2 = len:B = no. of rows of B = len•1:A = length of a row of A, and
n_3 = len•1:B = length of a row of B

Then the construct (60) corresponds exactly to the following Fortran-like program for the product of A and B, i.e., MM:<A,B>

```
for i = 1, n1
  for j = 1, n3
   r[i,j] := 0
   for  k = 1, n2
     r[i,j] := r[i,j] + (A[i,k] * B[k,j])
```

Thus our transformation has turned all the FP "housekeeping" operations [the first three operations of MM] into Fortran-like housekeeping operations. In a sense the transformation has turned the original program MM inside out; it has eliminated all the unnecessary intermediate results of the FP program and produced the standard matrix multiplication program.

Details of the transformation of MM. To use our strategy we wish to find a construct that is the identity for pairs of conformable matrices. Let

61) MI = [E(i) i=~1,len] where
62) E(i) = [k•i k=~1,len•1]

Then MI is the identity construct for any matrix, and

63) [MI•1, MI•2] is the identity for any pair of matrices
64) len•1•1 = len•2 is the conformability requirement that the length of a row of A (len•1•1) equals the length of a column [i.e., the no. of rows] of B (len•2)

So our strategy is to transform MM = MM•[MI•1,MI•2] by the above laws, where we can assume that (64) holds.

65) [1,trans•2]•[MI•1, MI•2] = [MI•1, trans•MI•2] by (25) (3) (12)
= [MI•1, MI'•2]

where

66) MI' = trans•MI = [E'(j) j=~1,len•1] by (49) (56)
where E'(j) = [j•k k=~1,len]

So we now have

```
67)  MM = @(@IP)•@distl•distr•[MI•1, MI'•2]
68)     = @(@IP)•@distl•distr•[[E(i)•1 i=~1,len•1], MI'•2]   by (61) (58) (14)
69)     = @(@IP)•@distl•[ [E(i)•1, MI'•2] i=~1,len•1]        by (51)
70)     = @(@IP)•[ distl•[E(i)•1, MI'•2] i=~1,len•1]         by (53)
71)     = @(@IP)•[ distl•[E(i)•1, [E'(j)•2 j=~1,len•1•2] ] i=~1,len•1]
                                                          by (66) (58) (14)
72)     = @(@IP)•[ [[E(i)•1, E'(j)•2] j=~1,len•1•2] ] i=~1,len•1]    by (52)
73)     = [ [IP•[E(i)•1, E'(j)•2] j=~1,len•1•2] i=~1,len•1]   by (53) twice
```

Now

```
IP•[E(i)•1, E'(j)•2] = IP•[[k•i•1 k=~1,len•1•1], [j•k•2 k=~1,len•2]]
                                                          by (62) (66)
```

Therefore we can use the identity for IP (59) to get

```
IP•[E(i)•1, E'(j)•2] = \+•[ *•[k•i•1, j•k•2] k=~1,len•2]
```

since by (64) len•1•1 = len•2. Substituting this last result in (73) gives (60), the desired construct.

5.5.3 Eliminating copying. Our final example demonstrates that our optimization technique can eliminate unnecessary copying of arguments. The preceding examples proceeded uniformly "downhill". That is, whenever a rule applies it eliminates a function from the composition that we are working on: g•C => C'. In this example we see a step g•C => g•C' that requires a "pattern matching" step that merely alters the construct so that a reducing rule g•C' => C'' will then apply.

Consider the following program

```
f = @+ • trans • [tlr,tl]
```

This program, given $\langle x_1,\ldots,x_n\rangle$, computes $\langle x_1+ x_2,\ldots,x_{n-1}+ x_n\rangle$. The first operation constructs two almost-complete copies of the argument, whereas it would be simpler to compute the result from the original argument. Beginning with C = [i i=~1,len], the identity construct for sequences, the proper arguments for f, we transform f = f•C as follows

```
f = @+•trans•[tlr,tl]•[i i=~1,len]
  = @+•trans•[tlr•[i i=~1,len], tl•[i i=~1,len]]     by (25)
  = @+•trans•[[i i=~1,(len - ~1)], [i i=~2,len]]     by (54) (55)
```

Now the rule (50) would apply if the bounds on i were the same in both constructs, but they are not. However (57) applied to the second construct makes the bounds equal. This is the pattern-matching step.

```
  = @+•trans•[[i i=~1,(len - ~1)], [(i+~1) i=~1,(len - ~1)]]  by (57)
  = @+•[[i,i+~1] i=~1,(len - ~1)]                             by (50)
```

= [+•[i,i+~1] i=~1,(len - ~1)] by (53)

Thus again we have reduced our program to a construct, one that represents the following Fortran-like program to compute f:x

```
for i = 1, (len:x - 1)
  r[i] := x[i] + x[i+1]
```

This has eliminated all intermediate results and copying.

6. Conclusions

We have argued that programs with good mathematical properties must be built at the function level. We have exhibited a system whose semantics are [or can be] given by function level equations, equations whose operations are program-forming operations and whose constants and variables range over programs. From these equations theorems are derived whose conclusions are again equations, for example, one that asserts the equality of a recursive and an iterative function.

These equations, together with many others, constitute a fairly large body of general knowledge, knowledge that can be easily and uniformly applied to a fragment [i.e., subtree] of any function level program. We hope the examples show that this is a clear and preferable way to present general results about programs, so that they can be easily recognized and applied by users.

Although some functional languages are stunningly elegant for expressing certain programs, notably SASL and KRC [Turner 81], they often tie themselves to the object level in one or more ways: by using lambda abstraction, by using curried functions, and by separating conditional expressions into separate equations: When lambda abstraction is used to express an object-to-object function it requires the use of object variables. If h is an arbitrary curried function, then one cannot perceive its arity in (h a) unless one knows that, e.g., (h a x y) is an object. Thus for general expressions to be readable they must be object level [even then, considerable analysis may be needed to resolve the arity of h]. Finally, SASL-style pattern matching equations without conditions depend on the presence of mutually exclusive object level expressions as arguments, hence they must be object level.

I believe languages like SASL are excellent for <u>writing</u> programs, but for the reasons outlined above, they tend to be object level ones and hence lack many of the advantages of the function level technology for <u>general</u> reasoning about and transforming programs [activities that are <u>essentially</u> function level]. With the addition of higher level functions and infinite sequences as in FP_{84} [Halpern, Williams,

Wimmers, Winkler 85], it seems clear that any object level program can be transliterated into a function level equivalent, thereby making it subject to all the laws and theorems of that technology. And if it is a close transliteration, as I believe it can be in most cases, then that means that function level programs should be as clear, and as easy to read and to write as SASL programs.

Uncurried function level expressions [using construction as a combining form] help to make clear the structure of a function's argument in a sharper way than in curried expressions; they also reduce the number of function levels needed, reserving higher levels for truly higher level matters. Uncurried function level definitions that use construction and function variables correspond exactly to object level ones; they also serve as useful function level laws about the defined function.

The examples of FP program optimization give some hope that FP data-rearranging housekeeping operations [when they can be transformed into conventional ones], are practically viable, even though they represent extreme inefficiencies when done literally. If that turns out to be true, this should be a considerable advance for programming since conventional housekeeping operations represent one of the most central but disorderly and non-reusable aspects of programming today. The FP housekeeping operations, on the other hand, are compact and reusable, and have useful mathematical properties.

The work on optimization reported here is, I believe, just the beginning of a line of interesting and useful research. It shows that some programs [that one might call "fixed-data processing" in contrast to "list-processing", and that are much clearer and more orderly than Fortran programs] can be transformed into Fortran programs that run as fast as if they had been written in Fortran in the first place. It shows that the strategy for doing this, in some cases, is just the use of mechanical "downhill" rules: find a general equation $f \cdot C = C'$, where f is the rightmost operation in the function to be optimized, substitute the proper constants, replace $f \cdot C$ by the construct C', and continue.

At least for certain kinds of programs, the optimization technology suggested here has a potential for producing higher speed results than various combinator technologies [Hughes 82, Stoye, Clarke & Norman 84, Turner 79]. Our approach is source level, leading to programs of a fairly standard kind with known, easy to estimate efficiencies. Although these other methods employ some interesting source level transformations, their target language is a list-processing one and has several complexities [e.g., pointer manipulation, garbage collection] behind the language "curtain" that make estimations of running times complex. This makes it hard to

evaluate alternative strategies for optimization, and in any case these target languages do not have the potential efficiency that Fortran-like languages do.

Many questions remain, of course. How to distinguish between the fixed-data and list-processing cases? Is it possible to develop mixed techniques for handling the two, in which some data will have relatively fixed allocations and be addressed by Fortran-like techniques, and some will be stored and treated by list-processing techniques? How can recursively-defined functions be optimized? What recursively defined functions can be transformed into closed form definitions that can then be optimized? Can other constructs be added that will enable us to use algebraic techniques in doing a lot of lower level optimizations also at the extended-source level? How complex will optimization strategies have to be to deal effectively with a large class of practical programs?

In spite of all the remaining questions, the work on optimization indicates that making full use of the mathematical properties of function level programs promises to have considerable payoffs. These will certainly be required to do the really extensive optimization that is needed to make easy-to-write function level programs run at the speed we have come to expect of Fortran programs.

Acknowledgments It is a pleasure to acknowledge the valuable help I received from John H. Williams on a number of the examples, and the many helpful discussions about various issues in the paper with him and with Edward L. Wimmers.

References

Backus, J. (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. CACM **21**:8 (August)

Backus, J. (1981a) Is computer science based on the wrong fundamental concept of "program"? An extended concept. in Algorithmic Languages, de Bakker & van Vliet, eds., North-Holland, Amsterdam (1981)

Backus, J. (1981b) The algebra of functional programs: function level reasoning, linear equations, and extended definitions. Lecture Notes in Computer Science #107, Springer-Verlag (April)

Backus, J. (1981c) Function level programs as mathematical objects. Proc. Conf. on Functional Programming Languages and Computer Architecture. ACM (October)

Bird, R. S. (1984) The promotion and accumulation strategies in transformational programming. Trans. Prog. Langs. & Systs. **6**,4 ACM (October)

Burstall, R. M., and Darlington, J. (1977) A transformation system for developing recursive programs. JACM **24**,1 (January)

Cohen, N. H. (1979) Characterization and elimination of redundancy in recursive programs. Proc. 6th Symp. on Principles of Prog. Langs. ACM

Dijkstra, E. W. (1976) A Discipline of Programming. Prentice-Hall

Guttag, J., Horning, J., and Williams, J. (1981) FP with data abstraction and strong typing. Proc. Conf. on Functional Programming Languages and Computer Architecture. ACM (October)

Halpern, J. Y., Williams, J. H., Wimmers, E. L., and Winkler, T. C. (1985) Denotational semantics and rewrite rules for FP. Proc. 12th Symp. on Princ. Prog. Langs., ACM (January)

Kieburtz, R. B., and Shultis, J. (1981) Transformations of FP program schemes. Proc. Conf. on Functional Programming Languages and Computer Architecture. ACM (October)

Hoare, C. A. R. (1969) An axiomatic basis for computer programming. CACM **12**,10 (October)

Hughes, R. J. M. (1982) Super combinators: a new implementation method for applicative languages. Proc. Symp. on LISP and Functional Programming ACM (August)

Manna, Z. (1974) Mathematical Theory of Computation. McGraw-Hill

Mills, H. D. (1975) The new math of computer programming. CACM **18**,1 (January)

Stoye, W. R., Clarke, T. J. W. and Norman, A. C. (1984) Some practical methods for rapid combinator reduction. Proc. Symp. on LISP and Functional Programming ACM (August)

Turner, D. A. (1979) A new implementation technique for applicative languages, Software Practice & Experience, vol 9, pp31-49

Turner, D. A. (1981) The semantic elegance of applicative languages. Proc. Conf. on Functional Programming Languages and Computer Architecture. ACM (October)

Wadler, P. (1981) Applicative style programming, program transformation, and list operators. Proc. Conf. on Functional Programming Languages and Computer Architecture. ACM (October)

Wadler, P. (1984) Listlessness is better than laziness: lazy evaluation and garbage collection at compile time. Proc. Symp. on LISP and Functional Programming ACM (August)

Williams, J. H. (1982) On the development of the algebra of functional programs. Trans. Prog. Langs. & Systs. **4**,4 ACM (October)
