A Transformation System for Developing Recursive Programs

R. M. BURSTALL AND JOHN DARLINGTON

University of Edinburgh, Edinburgh, Scotland

ABSTRACT A system of rules for transforming programs is described, with the programs in the form of recursion equations An initially very simple, lucid, and hopefully correct program is transformed into a more efficient one by altering the recursion structure Illustrative examples of program transformations are given, and a tentative implementation is described Alternative structures for programs are shown, and a possible initial phase for an automatic or semiautomatic program manipulation system is indicated

KEY WORDS AND PHRASES program transformation, program manipulation, optimization, recursion

CR CATEGORIES· 3 69, 4 12, 4 22, 5 24, 5 25

1. *Introduction*

We present here a system for transforming programs, where the programs are expressed as first order recursion equations. This recursive form seems well adapted to manipulation, much more so than the usual Algol-style form of program, and our transformation system consists of just a few simple rules together with a strategy for applying them. Despite their simplicity, these rules produce some interesting changes in the programs.

The overall aim of our investigation has been to help people to write correct programs which are easy to alter. To produce such programs it seems advisable to adopt a lucid, mathematical, and abstract programming style. If one takes this really seriously, attempting to free one's mind from considerations of computational efficiency, there may be a heavy penalty in program running time; in practice it is often necessary to adopt a more intricate version of the program, sacrificing comprehensibility for speed. The question then arises as to how a lucid program can be transformed into a more intricate but efficient one in a systematic way, or indeed in a way which could be mechanized.

It is perhaps surprising to notice that even in the rarefied language of purely recursive programs there is a sharp contrast between programs written for maximal clarity and those written for tolerable efficiency. As Knuth [11] points out, one does not have to consider translation from an Algol-style language to a machine code language as performed by optimizing compilers to get to grips with the issue; the contrast is in the program structure, particularly in the recursion (or loop) structure. We start with programs having extremely simple structures and only later introduce the complications which we usually take for granted even in high level language programs. These complications arise by introducing useful interactions between what were originally separate parts of the program, benefiting by what might be called "economies of interaction."

We proceed in a quite empirical manner, showing examples of various kinds of

These ideas were first presented in our paper at the Conference on Reliable Software, Los Angeles, 1975, of which this is a much revised and extended version

This work was supported by The Science Research Council

Authors' address Department of Artificial Intelligence, Edinburgh University, Hope Park Square—Meadow Lane, Edinburgh EH8 9NW, Scotland

program transformation and how they can be achieved with our system. We make no claim for any sort of completeness of the system; it embodies only one family of program transformations, and we have no formal delineation of this family. Nor do we have a general method of showing that the transformations improve efficiency. However, we hope that the examples will give the reader pleasure and convince him that the system has some power. He will see that the example programs become more complicated or "intertwined" as we transform them, less like mathematical definitions and more like "sensible" programs. We would be grateful for any suggestions for capturing this notion of intertwining more precisely.

The transformation rules can also be viewed as a possible initial phase of a mechanized program transformation system In fact they arose from our efforts to understand and systematize parts of an earlier system (Darlington [5], Darlington and Burstall [6]). That system started by removing recursions in favor of iterations where possible; only then did it make transformations from abstract to concrete data and eliminate some redundant computation; finally it arranged for overwriting of data structures. We now feel that as much manipulation as possible should be performed before removing recursion. In this we were largely influenced by Boyer and Moore's elegant and successful program for proving facts about LISP programs [2]

We have implemented our new rules as a semiautomatic program transformation system which relies on guidance from the user for key steps.

In Section 2 we introduce the transformation method informally In Section 3 we present it as a formal inference system whose sentences are sets of recursion equations. In Section 4 we give examples of its application. In Section 5 we outline a strategy for applying these transformation rules and describe the program improvement system which we have implemented. In Section 6 we discuss conversion to iterative form. In Section 7 we discuss a further rule that can be added to our system In Section 8 we discuss translation of programs on abstract data to ones on concrete data In Section 9 we list some open problems and related work. Finally, in Appendix 1 we apply our method to a more substantial example, and in Appendix 2 we show how it is possible to prove that the transformations do effect an improvement on a particular program.

2. *An Example*

Consider the following simple example. Given a function scalar product, written "$\cdot$", on vectors, defined by

$$x \cdot y = \sum_{i=1}^{n} x_i y_i,$$

we might wish to compute $a \cdot b + c \cdot d$. Rewriting this in recursive function form we have

$$dot(x, y, n) \Leftarrow \textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } dot(x, y, n-1) + x[n]y[n] \textbf{ fi}$$

and we want

$$f(a, b, c, d, n) \Leftarrow dot(a, b, n) + dot(c, d, n)$$

This is a clear definition of f, but we do not really need two separate recursive calculations (i.e. two independent loops) Let us try symbolically evaluating f using its definition

$$\begin{aligned} f(a,b,c,d,n) &\Leftarrow \textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } dot(a, b, n-1) + a[n]b[n] \textbf{ fi} \\ &\qquad + \textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } dot(c, d, n-1) + c[n]d[n] \textbf{ fi} \\ &\Leftarrow \textbf{if } n = 0 \textbf{ then } 0 + 0 \textbf{ else } (dot(a, b, n-1) + a[n]b[n]) \\ &\qquad\qquad + (dot(c, d, n-1) + c[n]d[n]) \textbf{ fi} \\ &\qquad\qquad\qquad \text{by a simple property of } \textbf{if} \cdots \textbf{then} \\ &\Leftarrow \textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } (dot(a, b, n-1) + dot(c, d, n-1)) \\ &\qquad\qquad + a[n]b[n] + c[n]d[n] \textbf{ fi} \\ &\qquad\qquad \text{by simple properties of } + \text{ and a sly rearrangement.} \end{aligned}$$

But $dot(a, b, n - 1) + dot(c, d, n - 1)$ is $f(a, b, c, d, n - 1)$, so we write

$f(a, b, c, d, n) \Leftarrow$ **if** $n = 0$ **then** 0 **else** $f(a, b, c, d, n - 1) + a[n]b[n] + c[n]d[n]$ **fi.**

This gives us a recursive definition of f, without using *dot*. It does not save any multiplications or additions (except the final one) but it combines the loop overheads and tests. Notice how the two scalar product calculations have become intertwined. Our (slight) economy comes from this interaction; we have lost lucidity by it

Thus we have first symbolically evaluated the program (we call this "unfolding"), then rearranged it, and then introduced a recursion (we call this "folding").[1] Let us now be more precise about these transformation rules

3. *Transformation Rules*

First let us polish our notation a little. A definition like

$dot(x, y, n) \Leftarrow$ **if** $n = 0$ **then** 0 **else** $dot(x, y, n - 1) + x[n]y[n]$ **fi**

is convenient for program execution, but for transformation purposes it seems rather easier to rewrite it as

$dot(x, y, 0) \Leftarrow 0$
$dot(x, y, n + 1) \Leftarrow dot(x, y, n) + x[n + 1]y[n + 1]$

This is easily translatable back into the conditional form, given that 0 and $x + 1$ are mutually exclusive and exhaustive forms for nonnegative numbers and that $x - 1$ is the inverse of $x + 1$.

As another example, the Fibonacci function

$f(x) \Leftarrow$ **if** $x = 0$ **or** $x = 1$ **then** 1 **else** $f(x - 1) + f(x - 2)$ **fi**

becomes

$f(0) \Leftarrow 1, \quad f(1) \Leftarrow 1, \quad f(x + 2) \Leftarrow f(x + 1) + f(x)$

The concatenation function on lists

$concat(x, y) \Leftarrow$ **if** $x = nil$ **then** y **else** $cons(car(x), concat(cdr(x), y))$ **fi**

becomes

$concat(nil, z) \Leftarrow z, \quad concat(cons(x, y), z) \Leftarrow cons(x, concat(y, z))$

Reworking the scalar product example, we get

$$\begin{array}{ll}
f(a, b, c, d, n) & \Leftarrow dot(a, b, n) + dot(c, d, n) \\
f(a, b, c, d, 0) & \Leftarrow dot(a, b, 0) + dot(c, d, 0) \\
 & \Leftarrow 0 \\
f(a, b, c, d, n + 1) & \Leftarrow dot(a, b, n + 1) + dot(c, d, n + 1) \\
 & \Leftarrow dot(a, b, n) + a[n + 1]b[n + 1] + dot(c, d, n) + c[n + 1]d[n + 1] \\
 & \Leftarrow dot(a, b, n) + dot(c, d, n) + a[n + 1]b[n + 1] + c[n + 1]d[n + 1] \\
 & \Leftarrow f(a, b, c, d, n) + a[n + 1]b[n + 1] + c[n + 1]d[n + 1]
\end{array}$$

We can now develop the method as a formal inference system whose sentences are recursion equations We omit definitions of well-known notions like instance and assume the usual call-by-name semantics of recursion equations.

Preliminaries. We need the following:

Primitive function – a set of primitive function symbols $k, l, \ldots$ and $c, d, \ldots$ with zero or more arguments; the subset $c, d, \ldots$ of primitive symbols are the *constructor* function symbols. (One of the primitive functions can be the conditional.) Examples of constructor functions would be *cons* and *successor*, which is written $\ldots + 1$ above

[1] The folding idea is also used by Manna and Waldinger [12] in their program synthesis work They developed it independently at about the same time as we did

Parameter — a set $x, y, \ldots$ of parameter variables.

Recursive function — a set $f, g, \ldots$ of recursive function symbols.

Expression — an expression built in the usual way out of primitive function symbols, parameter variables, and recursive function symbols. We allow the **where** construction E **where** $\langle u, \cdots, w\rangle = F$ or E **where** $u = F$, E and F being expressions and $u, \cdots, w$ being taken from a set of local variables (for example $u + u^2$ **where** $u = a + b$).

Left-hand expression — a left-hand expression is of the form $f(e_1, \cdots, e_n)$, $n \geq 0$, where $e_1, \cdots, e_n$ are expressions involving only parameter variables and constructor function symbols (disallowing **where**).

Right-hand expression — a right-hand expression is an expression.

We use E, F, G, possibly with primes or subscripts, as metasymbols to denote expressions.

Recursion equation — a recursion equation consists of a left-hand expression and a right-hand expression, written $E \Leftarrow F$.

Examples. $f(c) \Leftarrow k$. $f(d(x, y)) \Leftarrow l(u, u)$ **where** $u = m(x, y)$.

We have the usual notion of substitution and of one expression being an instance of another.

Inference rules for transforming recursion equations. Given a set of recursion equations, we may add to the set using the following inference rules, all except folding being rather obvious. We illustrate the rules by reference to the above example.

(i) *Definition.* Introduce a new recursion equation whose left-hand expression is not an instance of the left-hand expression of any previous equation. For example,

$$f(a, b, c, d, n) \Leftarrow dot(a, b, n) + dot(c, d, n).$$

(ii) *Instantiation.* Introduce a substitution instance of an existing equation. For example, instantiate

$$f(a, b, c, d, n) \Leftarrow dot(a, b, n) + dot(c, d, n)$$

to

$$f(a, b, c, d, 0) \Leftarrow dot(a, b, 0) + dot(c, d, 0).$$

(iii) *Unfolding* If $E \Leftarrow E'$ and $F \Leftarrow F'$ are equations and there is some occurrence in F' of an instance of E, replace it by the corresponding instance of E', obtaining F''; then add the equation $F \Leftarrow F''$. For example, unfolding with

$$dot(x, y, n + 1) \Leftarrow dot(x, y, n) + x[n + 1]y[n + 1] \quad (E \Leftarrow E')$$

takes

$$f(a, b, c, d, n + 1) \Leftarrow dot(a, b, n + 1) + dot(c, d, n + 1) \quad (F \Leftarrow F')$$

to

$$f(a, b, c, d, n + 1) \Leftarrow dot(a, b, n) + a[n + 1]b[n + 1] + dot(c, d, n) + c[n + 1]d[n + 1] \quad (F \Leftarrow F'').$$

(iv) *Folding.* If $E \Leftarrow E'$ and $F \Leftarrow F'$ are equations and there is some occurrence in F' of an instance of E', replace it by the corresponding instance of E, obtaining F''; then add the equation $F \Leftarrow F''$. For example, folding with

$$f(a, b, c, d, n) \Leftarrow dot(a, b, n) + dot(c, d, n) \quad (E \Leftarrow E')$$

takes

$$f(a, b, c, d, n + 1) \Leftarrow dot(a, b, n) + dot(c, d, n) + a[n + 1]b[n + 1] + c[n + 1]d[n + 1] \quad (F \Leftarrow F')$$

to

$$f(a, b, c, d, n + 1) \Leftarrow f(a, b, c, d, n) + a[n + 1]b[n + 1] + c[n + 1]d[n + 1] \quad (F \Leftarrow F'').$$

(v) *Abstraction.* We may introduce a **where** clause by deriving from a previous equation $E \Leftarrow E'$ a new equation,

$E \Leftarrow E'[u_1/F_1, \cdots, u_n/F_n]$ **where** $\langle u_1, \cdots, u_n \rangle = \langle F_1, \cdots, F_n \rangle$.

Abstraction is not used in the *dot* example. We can see an example of its use in the Fibonacci example in Section 4.

(vi) *Laws.* We may transform an equation by using on its right-hand expression any laws we have about the primitives $k, l, \ldots$ (associativity, commutativity, etc.), obtaining a new equation For example, the commutativity of + enables us to rewrite

$$f(a,b,c,d,n+1) \Leftarrow dot(a,b,n) + a[n+1]b[n+1] + dot(c,d,n) + c[n+1]d[n+1]$$

as

$$f(a,b,c,d,n+1) \Leftarrow dot(a,b,n) + dot(c,d,n) + a[n+1]b[n+1] + c[n+1]d[n+1].$$

Each new equation obtained by these rules may be taken as a definition of the function appearing on the left provided we take a disjoint and exhaustive subset of them. (The notions of disjointness and exhaustiveness depend on the data domain; we do not attempt an explicit definition but they should be clear enough for integers, lists, etc.)

We believe that these inference rules preserve correctness, although we do not have a formal proof of this. An informal argument, for which to thank G. Plotkin, is that the effect of using our rules could equally well be obtained as follows: First rewrite the definitions, say $E \Leftarrow E'$, $F \Leftarrow F'$, ... (i), as the corresponding equations $E = E'$, $F = F'$, ... (ii); now each of our transformation rules can be seen to correspond to a sound rule for deducing a new equation, so use these rules to get new equations, say $E_1 = E_1'$, $F_1 = F_1'$, .. (iii); then choose a subset of these (exhaustive and disjoint), say $E_2 = E_2'$, $F_2 = F_2'$, ... (iv), then rewrite these as definitions $E_2 \Leftarrow E_2'$, $F_2 \Leftarrow F_2'$, ... (v). Now the functions defined by (v) satisfy eqs. (iv) and are the least such functions. But the functions defined by (i) satisfy eqs. (ii) and hence eqs. (iii) and hence eqs. (iv) So the functions defined by (v) are less than or equal to the ones defined by (i). (f is less than or equal to g if $f(x) = g(x)$ whenever $f(x)$ is defined) That is, we retain correctness, but we might lose termination unless we impose some extra restriction

As to whether our transformations improve the efficiency of programs, we do not know sufficient conditions for this in general. However, in Appendix 2 we show that they do improve the Fibonacci program (and so a fortiori preserve its termination). The reasoning employed suggests a general argument that

(i) improvements can be introduced by rewriting lemmas and by abstraction;

(ii) instantiation and unfolding leave efficiency unchanged;

(iii) folding at least preserves efficiency provided that the argument of the equation used in the substitution is lower in some well-founded ordering than that of the equation being transformed.

Strategy. A simple strategy for applying these rules turns out to be quite powerful, and it will be used in the examples which follow. Thus.

(a) Make any necessary *definitions.*
(b) *Instantiate.*
(c) For each instantiation *unfold* repeatedly. At each stage of unfolding:
    (d) Try to apply *laws and where-abstraction.*
    (e) *Fold* repeatedly.

Stages (a) and (b) require some invention from the user, (d) requires his discretion, but (c) and (e), unfolding and folding, are routine symbol manipulation.

We discuss strategy and a semiautomatic implementation in Section 5.

4. *Examples of Use of the Transformation System*

Example 1. Fibonacci. Let us look at a case where we can make a substantial gain in efficiency by introducing a new recursive definition which intertwines what were originally separate computations. In this case we avoid computing certain values twice.

We use "eureka" to draw attention to certain unobvious steps in the transformations. The reader may feel that certain other steps deserve "eureka." In Section 5, however, we describe a program improving system based on the transformation rules and show how it can automatically achieve these steps. The steps marked here indicate the help the user has to give the system at present.

We take the definition of Fibonacci made in Section 3 as our starting point. (We regard $x + 1$ as an abbreviation for *successor*(x) and pairing as a primitive function written $\langle \ldots, \ldots \rangle$.)

1.	$f(0)$	$\Leftarrow 1$	given
2.	$f(1)$	$\Leftarrow 1$	given
3.	$f(x+2)$	$\Leftarrow f(x+1) + f(x)$	given
4.	$g(x)$	$\Leftarrow \langle f(x+1), f(x)\rangle$	definition (eureka)
5.	$g(0)$	$\Leftarrow \langle f(1), f(0)\rangle$	instantiation
		$\Leftarrow \langle 1, 1\rangle$	unfolding with 1 and 2
6.	$g(x+1)$	$\Leftarrow \langle f(x+2), f(x+1)\rangle$	instantiate 4
		$\Leftarrow \langle f(x+1) + f(x), f(x+1)\rangle$	unfold with 3
		$\Leftarrow \langle u+v, u\rangle$ **where** $\langle u, v\rangle = \langle f(x+1), f(x)\rangle$	abstract
		$\Leftarrow \langle u+v, u\rangle$ **where** $\langle u, v\rangle = g(x)$	fold with 4
7.	$f(x+2)$	$\Leftarrow u + v$ **where** $\langle u, v\rangle = \langle f(x+1), f(x)\rangle$	abstract 3
		$\Leftarrow u + v$ **where** $\langle u, v\rangle = g(x)$	fold with 4

Now notice the pattern of applying the inference rules. 4 comes by inspiration, although motivated somewhat by 3. 5 and 6 are the obvious instantiations of 4; for each of them we first unfold as far as possible, then for 6, abstract in order to fold. 7 is an unobvious abstraction of 3 made in order to fold.

Thus the new definition of Fibonacci is:

$$\begin{array}{ll}
f(0) & \Leftarrow 1 \\
f(1) & \Leftarrow 1 \\
f(x+2) & \Leftarrow u + v \ \textbf{where}\ \langle u, v\rangle = g(x) \\
g(0) & \Leftarrow \langle 1, 1\rangle \\
g(x+1) & \Leftarrow \langle u+v, u\rangle \ \textbf{where}\ \langle u, v\rangle = g(x)
\end{array}$$

This computes the result in linear time in x instead of exponential.

Example 2. The scalar product example combined two independent loops into one; the Fibonacci example transformed a binary recursion into a loop. Our next example combines two binary recursions into one. We assume a tree is either *tip* of an atom or *tree* of two trees, where *tip* and *tree* are constructor functions. We are given f, which computes the sum of the tips, and g, which computes their product, and we wish to compute both of them at once.

1.	$f(tip(x))$	$\Leftarrow x$	given
2.	$f(tree(x, y))$	$\Leftarrow f(x) + f(y)$	given
3.	$g(tip(x))$	$\Leftarrow x$	given
4.	$g(tree(x, y))$	$\Leftarrow g(x) * g(y)$	given
5.	$h(x)$	$\Leftarrow \langle f(x), g(x)\rangle$	given
6.	$h(tip(x))$	$\Leftarrow \langle f(tip(x)), g(tip(x))\rangle$	instantiation
		$\Leftarrow \langle x, x\rangle$	unfolding 1, 3
7.	$h(tree(x, y))$	$\Leftarrow \langle f(tree(x, y)), g(tree(x, y))\rangle$.	instantiation
		$\Leftarrow \langle f(x) + f(y), g(x) * g(y)\rangle$	unfolding 2, 4
		$\Leftarrow \langle u+v, w*t\rangle$ **where** $\langle u, w, v, t\rangle = \langle f(x), g(x), f(y), g(y)\rangle$.	abstraction
		$\Leftarrow \langle u+v, w*t\rangle$ **where** $\langle\langle u, w\rangle, \langle v, t\rangle\rangle = \langle h(x), h(y)\rangle$.	folding with 5

Thus h computes both functions at once.

Example 3. Table of factorials. Suppose we want to make a table of factorials. We may define naively:

1. $\mathit{fact}(0)$	$\Leftarrow 1$
2. $\mathit{fact}(n + 1)$	$\Leftarrow (n + 1)*\mathit{fact}(n)$
3. $\mathit{factlist}(0)$	$\Leftarrow \mathit{nil}$
4. $\mathit{factlist}(n + 1)$	$\Leftarrow \mathit{cons}(\mathit{fact}(n + 1), \mathit{factlist}(n))$

Thus *factlist*(4) is (24, 6, 2, 1), but each of these is computed afresh. Let us improve the definition of *factlist*:

5. $g(n)$	$\Leftarrow \langle \mathit{fact}(n + 1), \mathit{factlist}(n)\rangle$	definition (eureka)
6 $g(0)$	$\Leftarrow \langle \mathit{fact}(1), \mathit{factlist}(0)\rangle$	instantiate 5
	$\Leftarrow \langle 1, \mathit{nil}\rangle$	unfold 2, 4, 1 and use law about *
7. $g(n + 1)$	$\Leftarrow \langle \mathit{fact}(n + 2), \mathit{factlist}(n + 1)\rangle$	instantiate 5
	$\Leftarrow \langle (n + 2)*\mathit{fact}(n + 1), \mathit{cons}(\mathit{fact}(n + 1), \mathit{factlist}(n))\rangle$	unfold 2, 4
	$\Leftarrow \langle (n + 2)*u, \mathit{cons}(u, v)\rangle$ **where** $\langle u, v\rangle = \langle \mathit{fact}(n + 1), \mathit{factlist}(n)\rangle$	abstract
	$\Leftarrow \langle (n + 2)*u, \mathit{cons}(u, v)\rangle$ **where** $\langle u, v\rangle = g(n)$	fold with 5
8. $\mathit{factlist}(n + 1)$	$\Leftarrow \mathit{cons}(u, v)$ **where** $\langle u, v\rangle = \langle \mathit{fact}(n + 1), \mathit{factlist}(n)\rangle$	abstract 4
	$\Leftarrow \mathit{cons}(u, v)$ **where** $\langle u, v\rangle = g(n)$	fold with 5

This new definition of *factlist* computes *fact*$(n + 1)$ from *fact*(n), cutting down the computation from time n^2 to time n. Notice, however, that to do *factlist* in ascending order is not easy with our technique and seems to require an extension of the rules. We are currently investigating this problem.

Example 4. Testing trees for equality of frontiers. Another example of a more substantial nature is a program to test whether two binary trees have the same frontier, that is, the same sequence of atoms at their tips. An obvious definition involves first computing the frontier list for each and then comparing these two lists element by element. The comparison can stop as soon as two differing elements in these lists are detected, but by that time we would have already computed the whole frontier lists, quite unnecessarily. Because the two trees may differ in shape, it is not easy to compare the two frontiers element by element as they are generated. Indeed this was proposed as a problem to illustrate the virtues of coroutines. However, our transformation system can produce a satisfactory recursive program provided the user defines a generalization of the problem, namely comparing the frontiers of two lists of trees instead of two single trees The formal definition of the problem and the details of the transformations required are quite long, so we have relegated them to Appendix 1.

5. *Strategies for Applying the Transformation Rules, and Implementation of a Program Improving System*

Instead of just having a set of transformation rules which can be freely applied in all possible ways, we would like a more algorithmic system, avoiding search as far as possible. We are experimenting with strategies for applying the rules such as the strategy described briefly above Some observations seem helpful.

(i) Almost all the optimizing transformations consist of a sequence of unfoldings, rewriting by lemmas, and then foldings.

(ii) Use of associativity, commutativity, and **where**-abstraction can usually be delayed until just before folding.

We use (ii) to cut down fruitless use of associativity, commutativity, and **where**-abstraction by combining them with the folding process, using them only when they make a fold possible. We call this combined step "forced folding" and discuss it in more detail later.

The following heuristic algorithm is based on these assumptions. This algorithm is applied to each instantiation of the equation to be improved.

Algorithm 1

1. Arbitrarily do an unfold or rewriting by a lemma. Arbitrarily either repeat step 1 or go to step 2
2. Do an arbitrary forced fold. Repeat step 2 until no more folding is possible.

The arbitrary choices are made in an exhaustive manner using backtracking. Algorithm 1 is quite laborious, a further observation gives us a faster but less general algorithm

(iii) In the cases where our equational method of writing programs using constructor operations on the left-hand side ensures that unfolding cannot go on indefinitely, folding can usually be delayed until all possible unfolding has been done, provided that all the equations in the system are kept in fully unfolded form.

Algorithm 2 is based on this assumption.

Algorithm 2

0. Unfold each equation until no further unfolding is possible.

For each instantiation of the equation to be improved:

1. Unfold until no further unfolding is possible.
2. Arbitrarily either do rewriting by a lemma and goto step 1 or goto step 3.
3. Do an arbitrary forced fold Repeat step 3 until no more folding is possible.

Both Algorithms 1 and 2 succeed for all the examples given here with the exception of part of *Treesort* (Section 8).

A PROGRAM IMPROVING SYSTEM. We have implemented an experimental heuristic program improving system based on the transformation rules and Algorithms 1 and 2. As we have mentioned, an earlier program improving system has been described (Darlington and Burstall [6]) which enabled the user to write his program in a high level abstract language using recursion equations and have them translated into more efficient but less transparent versions. This earlier system used several separate transformation processes; the transformation rules described here unify and extend these processes except for storage overwriting, which we have not yet considered.

At present in the new system the work is shared, though not interactively, between the user and the system. As the system is developed we hope to shift more work from the user. At present (January 1976) the user is required to give:

(i) The list of equations augmented by any necessary definitions (i.e. the ones marked with "eureka" in the examples).

(ii) A list of useful lemmas in equation form (for use as rewrite rules) and statements of which functions are associative or commutative or both.

(iii) A list of all the properly instantiated left-hand sides of the equations on which the user wants the system to work.

The system then searches through the space of all possible transformations of (iii) looking for folds with (i) using either Algorithm 1 or Algorithm 2 as desired. The resulting new equations are printed out for examination by the user. At present no effort is made by the system to assess the efficiency of these new definitions.

A sample of a dialogue with the system for the Fibonacci improvement is shown in Table I.

To see whether a fold can be achieved we use a matching routine. Given two expressions this seeks a substitution which transforms the first into the second; for example, given $n + (m + k)$ and $(n + 1) + (m + k)$ it finds that the substitution n goes to $n + 1$. Our matching routine has commutativity and associativity built into it where this is specified; for example, given $n + (m + k)$ and $m + (n + 1 + k)$ it can still find the same substitution. It is also capable of matching to within an abstraction; an example of this is

TABLE I. Sample Dialogue for Fibonacci

: *START*;	(user starts dialogue)
INPUT EQUATIONS, END WITH Z	(system responds)
· $f(0) \Leftarrow 1$	(user inputs equations)
. $f(1) \Leftarrow 1$	
. $f(x+2) \Leftarrow f(x+1)+f(x)$	
. $g(x) \Leftarrow \langle f(x+1), f(x)\rangle$	
Z	
INPUT REWRITING LEMMAS, END WITH Z	
. *Z*	(no lemmas needed; associativity and commutativity are indicated when the functions are declared)
INPUT INSTANCES OF FUNCTIONS YOU ARE INTERESTED IN, END WITH Z	(system responds)
. $g(0)$	
$g(x+1)$	
. $f(x+2)$	
: *Z*	(system starts work outputting results as it gets them)
$g(0) \Leftarrow \langle 1, 1\rangle$	(system outputs any ground term it achieves)
$g(x+1) \Leftarrow \langle u+v, u\rangle$ **where** $\langle u, v\rangle = g(x)$	
.	(system outputs any fold it achieves)
several other folds	
$f(x+2) \Leftarrow u+v$ **where** $\langle u, v\rangle = g(x)$	
.	
several other folds	

given below. Thus these laws are never applied unless they immediately result in a fold. Plotkin [14] gives a general theory of building in laws to unification. We are grateful to Rodney Topor for the associative and commutative parts of the matcher [15].

For an example of inbuilt where-abstraction, consider the Fibonacci example. Simple unfolding gives the system $g(x+1) \Leftarrow \langle f(x+1)+f(x), f(x+1)\rangle$, which it is trying to fold with $g(x) \Leftarrow \langle f(x+1), f(x)\rangle$.

The matching routine spots that all the necessary components for a match with $\langle f(x+1), f(x)\rangle$ are present within $\langle f(x+1)+f(x), f(x+1)\rangle$, and it forces the rearrangement of the latter into $\langle u+v, u\rangle$ **where** $\langle u, v\rangle = \langle f(x+1), f(x)\rangle$ by applying abstraction; this folds immediately

Future Developments. A desirable next stage in the development of our system is to get it to produce automatically the definitions that the user currently has to supply. This is where a lot of the cleverness of the optimization resides. Though in a number of cases it is clear how to do this theoretically, it is not yet clear whether it can be done efficiently without excessive search. The idea is to expand out to some extent the computation tree generated by the equations and then to look for a match between the higher nodes in this tree and the lower ones. We need a substitution which when applied to the lower nodes gives the higher ones.

Consider, for example, the list of factorials problem above where we were given the equations

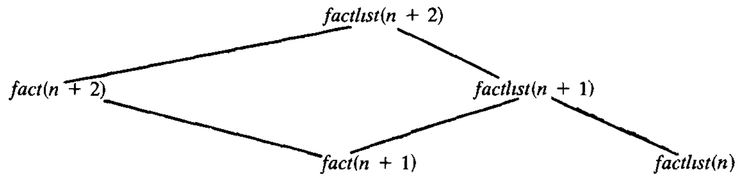
$$\begin{aligned} &\mathit{fact}(n+1) && \Leftarrow (n+1)*\mathit{fact}(n) \\ &\mathit{factlist}(n+1) && \Leftarrow \mathit{cons}(\mathit{fact}(n+1), \mathit{factlist}(n)) \end{aligned}$$

We would like to express *factlist* in terms of some new function, say g, which would itself have a recursive definition of the form $g(n+1) \Leftarrow \cdots g(n) \cdots$ or, more generally, $g(\sigma(n)) \Leftarrow \cdots g(n) \cdots$ for some arbitrary substitution σ.

Since we cannot expand $\mathit{factlist}(n+1)$ further, we try $\mathit{factlist}(n+2)$; thus

$$\begin{aligned} \mathit{factlist}(n+2) &\Leftarrow \mathit{cons}(\mathit{fact}(n+2), \mathit{factlist}(n+1)) \\ &\Leftarrow \mathit{cons}((n+2)*\mathit{fact}(n+1), \mathit{cons}(\mathit{fact}(n+1), \mathit{factlist}(n))) \end{aligned}$$

Pictorially the execution tree is of the form

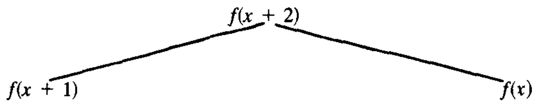


We notice that the substitution $\sigma(n) = n + 1$ takes the pair of terms on the bottom line to those in the previous line. Thus if we put

$g(n) \Leftarrow \langle fact(n + 1), factlist(n)\rangle$

we can express $factlist(n + 2)$ in terms of $g(n + 1)$ and express $g(n + 1)$ in terms of $g(n)$.

Similarly in the Fibonacci example we have the definition $f(x + 2) \Leftarrow f(x + 1) + f(x)$. Even without further expansion we have the computation tree



and notice that we can find a substitution $\sigma(x) = x + 1$ which takes the lower pair of nodes $f(x + 1)$ and $f(x)$ to the higher (overlapping) pair $f(x + 2)$ and $f(x + 1)$. Thus putting $g(x) \Leftarrow \langle f(x + 1), f(x)\rangle$ we can express $f(x + 2)$ in terms of $g(x + 1)$, and express $g(x + 1)$ in terms of $g(x)$.

Thus we see that the general approach is to expand out the computation tree and seek a substitution, taking some lower 'slice' across it into a higher slice across it. It is appropriate for examples where the auxiliary definition is a tuple of terms occurring in the computation, but we will see other examples where it does not work. In Section 6, recursion to iteration, we will need to introduce an extra variable, and in Appendix 1 we generalize from an element to a list of elements. Still it does show some rationale for auxiliary definitions.

One further development we wish to incorporate in the near future is to give the matcher the ability to synthesize subsidiary functions. Further details of this technique can be found in Darlington [7].

6. *Conversion to Iterative Form*

The same transformation system can be used to convert from recursive to iterative form. We say that a set of definitions of functions $\{f_1, \cdots, f_m\}$ are in iterative form if for each equation $f_i(x_1, \cdots, x_n) \Leftarrow E$, either E does not contain any of the f_i, or it is of the form $f_k(E_1, \cdots, E_n)$ and $E_1, \cdots, E_n$ do not contain any of the f_i, or it is a conditional expression whose alternatives are of one of these forms Such recursive definitions can be trivially rewritten as loops with the f_i as labels. The transformation is not automatic, as we have to introduce a new definition each time; however, these definitions are all of a similar pattern and are "generalizations" of the original function definition, replacing subexpressions on the right by variables and including the variables as extra parameters; in each case the major operator on the right is associative. Such generalizations were central to the success of the Boyer-Moore program prover [2], and we have profited from the studies of Aubin [1] and Moore [13] who extended that prover to deal with programs in the above iterative forms; they use generalization to translate from iterative to truly recursive form (the opposite approach to ours).

Example 1. Factorial.

1. $factorial(0)$	$\Leftarrow 1$	given
2. $factorial(n + 1)$	$\Leftarrow (n + 1) * factorial(n)$	given

Introduce a new function f by generalizing $n + 1$ to u.

3. $f(n, u)$	$\Leftarrow u * factorial(n)$	definition (eureka)
4. $f(0, u)$	$\Leftarrow u$	instantiate, unfold
5. $f(n + 1, u)$	$\Leftarrow u*((n + 1)*factorial(n))$	instantiate, unfold
	$\Leftarrow f(n, u*(n + 1))$	associativity of $*$, fold with 3
6. $factorial(n + 1)$	$\Leftarrow f(n, n + 1)$	fold 2 using 3

This definition (1, 6, 4, 5) is in iterative form.

A more succinct definition would be obtained by replacing 1 and 6 by $factorial(n) \Leftarrow f(n, 1)$. Our rules, as they stand, do not allow us to derive this, but in Section 7 we discuss an additional rule which would yield it.

Example 2. List reverse.

1. $reverse(nil)$	$\Leftarrow nil$	given
2. $reverse(a :: x)$	$\Leftarrow reverse(x) \langle\rangle (a :: nil)$	given

(:: *and* $\langle\rangle$ are infixes for *cons* and *concat*; see Section 3 for definition.)

Introduce a new function f by generalizing $a :: nil$ to u.

3. $f(x, u)$	$\Leftarrow reverse(x) \langle\rangle u$	definition (eureka)
4. $f(nil, u)$	$\Leftarrow u$	instantiate and unfold
5. $f(a :: x, u)$	$\Leftarrow (reverse(x) \langle\rangle (a :: nil)) \langle\rangle u$	instantiate and unfold
	$\Leftarrow f(x, (a :: nil) \langle\rangle u)$	associativity and fold

($\Leftarrow f(x, a :: u)$ if we allow further unfolds, which is however contrary to our mechanized strategies.)

6. $reverse(a :: x)$	$\Leftarrow f(x, a :: nil)$	fold 2 with 3

Again this is in iterative form As before $reverse(x) \Leftarrow f(x, nil)$ instead of 1 and 6 would be more succinct but requires the extra rule described in the Section 7.

Example 3. Frontier of a tree. This example uses the same generalization but does not produce an iterative function. It produces an equation of the form

$$f(x_1, \cdots, x_n) \Leftarrow f(E_1, \cdots, E_n)$$

but the E_i do contain f The new definition, however, is faster.

As in a previous example, by the frontier of a tree we mean the list of its tip elements We need two constructor functions: *tip* (to indicate a tip element) and *tree* (to form a binary branch).

1. $frontier(tip(a))$	$\Leftarrow a :: nil$	given
2. $frontier(tree(t1, t2))$	$\Leftarrow frontier(t1) \langle\rangle frontier(t2)$	given

Introduce f by generalizing

3. $f(t, u)$	$\Leftarrow frontier(t) \langle\rangle u$	definition (eureka)
4. $f(tip(a), u)$	$\Leftarrow a :: u$	instantiate, unfold
5. $f(tree(t1, t2), u)$	$\Leftarrow (frontier(t1) \langle\rangle frontier(t2)) \langle\rangle u$	instantiate, unfold
	$\Leftarrow f(t1, f(t2, u))$	associativity, fold, fold
6. $frontier(tree(t1, t2))$	$\Leftarrow f(t1, frontier(t2))$	unfold 2, fold with 3

This definition (1, 6, 4, 5) is faster since it only uses :: and not $\langle\rangle$.

$frontier(t) \Leftarrow f(t, nil)$ is more succinct, but as before it needs an extra rule.

7. *An Extra Transformation Rule: Redefinition*

The transformation rules described so far have allowed us to start with a definition of a function, instantiate it, unfold, and fold to get a new recursive definition of it. But sometimes for the sake of efficiency we may wish to move in the opposite direction. Here is an example of an improvement which cannot be made by our system so far (this is due to Michael Paterson). We define f by

1. $f(0)$	$\Leftarrow 0$	definition
2. $f(n + 1)$	$\Leftarrow f(n)$	definition

Now a better definition of f would be

$f(n)$	$\Leftarrow 0$	(?)

but this is clearly not obtainable by instantiation, unfolding, and folding, having $f(n)$ on the left.

But the reverse direction can be done by our rules:

3. $f'(n)$	$\Leftarrow 0$	definition
4. $f'(0)$	$\Leftarrow 0$	instantiate
5. $f'(n + 1)$	$\Leftarrow 0$	instantiate
	$\Leftarrow f'(n)$	fold with 3

Now Dana Scott pointed out to us recently that we could introduce an extra rule into our system, making use of the fact that if we can transform a function definition (say 3) into a set of equations (4, 5) identical to those defining some previous function (1, 2), we know that the newly defined function is equal to the previous one wherever the latter terminates. (We should check the totality of the previous definition to ensure that the new one does not introduce spurious values where the previous one failed to terminate.)

Since we have just shown that f' satisfies 4 and 5 and these are identical to 1 and 2 which define f, we may use this to redefine f to be

3. $f(n) \Leftarrow 0$

We call this new rule "redefinition." In general we are given a function (totally) defined by some equations and proceed as follows:

(a) Make a new definition for the given function (eureka)

(b) Transform this new definition by our previous rules to get equations identical to the original equations for the given function.

(c) Replace the original equations by the new definition (redefinition rule).

We have not had time to explore the utility of this new rule, which essentially allows us to reverse our previous transformations. However it does clear up a difficulty mentioned in our recursion to iteration examples above Recall that our final definition of factorial was

1.	*factorial*(0)	$\Leftarrow 1$
6.	*factorial*$(n + 1)$	$\Leftarrow f(n, n + 1)$
4.	$f(0, u)$	$\Leftarrow u$
5.	$f(n + 1, u)$	$\Leftarrow f(n, u*(n + 1))$

Now to obtain a better (smaller although no faster) version

7.	*factorial'*(n)	$\Leftarrow f(n, 1)$	definition (eureka)
8.	*factorial'*(0)	$\Leftarrow f(0, 1)$	instantiate 7
		$\Leftarrow 1$	unfold with 4
9.	*factorial'*$(n + 1)$	$\Leftarrow f(n + 1, 1)$	instantiate 7
		$\Leftarrow f(n, n + 1)$	unfold with 5, use $1*x = x$

Now we use our new rule, noting the identity of 8 and 9 with 1 and 6 and replacing the latter with a copy of 7.

10.	*factorial*(n)	$\Leftarrow f(n, 1)$	redefinition

Similarly we can obtain succinct definitions for *reverse* and *frontier*.

We have not implemented this new rule in our mechanized system, this could be put in as an extra option with the user asserting, say

$$\mathit{factorial}(n) \Leftarrow f(n, 1) \qquad \text{by redefinition}$$

where the system could look up 1 and 6, instantiate the new definition similarly, unfold, check the identity of the equations so obtained with the previous ones, and then remove 1 and 6 in favor of this new definition.

8. *Abstract Programming and Data Type Change*

In Darlington [5] and Darlington and Burstall [6] a method was presented where hierarchically structured functional programs were flattened into programs expressed entirely in terms of the lowest level primitives, with consequent gain in efficiency but loss of understandability. This was achieved (for straight line programs only) by a technique of combined optimization with replacement of procedure calls by their bodies. The implemented system had extra techniques built into it which took advantage of known relationships between the abstract objects and their representations (in this case sets and lists or bit strings) to perform extra optimizations. We now propose a new technique for structuring such programs and show how the new method can flatten such programs (which need not now be only straight line ones), doing away with the need to build in representation dependent optimizations.

The usual method of structuring data is to write primitive functions for the higher, more abstract data types in terms of the lower data types (see for example Hoare [10]). We propose to remove the need to provide these and just ask for a single representation function mapping the lower data type onto the higher. We are grateful to Hoare for suggesting this simplification to us. The advantages of this method are

(i) Less work is involved for the programmer.

(ii) The division between abstract object and representation is much cleaner and more natural. All abstract programs are written entirely in terms of abstract primitives. The representation relationship was implicit in the earlier method but was never made clear even to the programmer himself.

(iii) Resulting programs are much more modular and easier to modify If a user wants to add a new representation all he has to do is to add one new representation function, not rewrite a number of functions.

We still have the problem of rewriting the abstract programs in terms of the lower primitives. We show how our method accomplishes this by means of another simple example, which we hope also clarifies this method of structuring programs.

Example 1. Twisting a tree. Suppose someone wishes to write programs to manipulate trees labeled with atoms at their nodes. He can define labeled trees inductively, using constructors *niltree* and *ltree*.

$\mathit{niltree} \in \mathit{labeled\text{-}trees}$
$\mathit{ltree}\colon \ \mathit{atoms} \times \mathit{labeled\text{-}trees} \times \mathit{labeled\text{-}trees} \rightarrow \mathit{labeled\text{-}trees}$

(That is, *ltree* is a three-argument function taking an atom and two labeled trees and producing a labeled tree)

Assuming a LISP-like machine on which binary trees are available as a basic data structure with constructors *nil* and *pair*,

$\mathit{nil} \in \mathit{binary\text{-}trees}$
$\mathit{atoms} \in \mathit{binary\text{-}trees}$
$\mathit{pair}\colon \ \mathit{binary\text{-}trees} \times \mathit{binary\text{-}trees} \rightarrow \mathit{binary\text{-}trees}$

The programmer could choose to represent the labeled trees using for each node a binary tree consisting of first the atom and second another binary tree consisting of the left and right subtrees. For example, *pair*(*A*, *pair*(*nil*, *pair*(*B*, *pair*(*nil*, *nil*)))) represents *ltree*(*A*, *niltree*, *ltree*(*B*, *niltree*, *niltree*)). To do this he would simply define the *representation function*

R: *binary-trees* → *labeled-trees*
R(*nil*) ⇐ *niltree*
R(*pair*(*a*, *pair*(*p*1, *p*2))) ⇐ *ltree*(*a*, *R*(*p*1), *R*(*p*2))

The user can now write labeled tree manipulating functions entirely in terms of the labeled tree primitives. A very simple one is

twist: *labeled-trees* → *labeled-trees*
twist(*niltree*) ⇐ *niltree*
twist(*ltree*(*a*, *t*1, *t*2)) ⇐ *ltree*(*a*, *twist*(*t*2), *twist*(*t*1))

We now want to produce *TWIST*. *binary-trees* → *binary-trees* which simulates this on concrete data. Our method requires availability of a *coding function* *C*, inverse to the representation function *R*, such that *R*(*C*(*t*)) = *t* We have some ideas on how to produce such inverses automatically, but they are tentative and we omit them here. In this case *C* is

C: *labeled-trees* → *binary-trees*
C(*niltree*) ⇐ *nil*
C(*ltree*(*a*, *t*1, *t*2)) ⇐ *pair*(*a*, *pair*(*C*(*t*1), *C*(*t*2)))

We want *TWIST*(*p*) = *C*(*twist*(*R*(*p*))). Thus

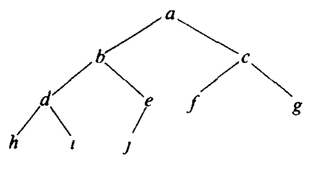
$$\begin{tikzcd}
\text{ABSTRACT} & \text{labeled-trees} \arrow[r, "twist"] \arrow[d, "R\ (\text{representation})"'] & \text{labeled-trees} \arrow[d, "C\ (\text{coding})"] & \text{ABSTRACT} \\
\text{CONCRETE} & \text{binary-trees} \arrow[r, "TWIST"] & \text{binary-trees} & \text{CONCRETE}
\end{tikzcd}$$

But this is not at all a usable definition since it uses *twist*, *R*, and *C*, which are not implemented. Let us massage it a little.

$$\begin{aligned}
&TWIST(nil) \Leftarrow C(twist(R(nil))) && \text{instantiate}\\
&\qquad \Leftarrow nil && \text{unfold}\\
&TWIST(pair(a, pair(p1, p2)))\\
&\qquad \Leftarrow C(twist(R(pair(a, pair(p1, p2))))) && \text{instantiate}\\
&\qquad \Leftarrow pair(a, pair(C(twist(R(p2))), C(twist(R(p1))))) && \text{unfold}\\
&\qquad \Leftarrow pair(a, pair(TWIST(p2), TWIST(p1))) && \text{fold}
\end{aligned}$$

This gives a recursive definition of *TWIST* in terms of the available concrete primitives *nil* and *pair*.

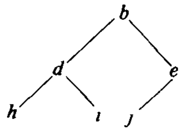
Example 2. Treesort. Now consider the Treesort algorithm of Floyd [8]. This is a sorting algorithm using arrays to represent trees. The algorithm makes repeated calls to a procedure *siftup*, which takes an arbitrary tree and moves its root element along some branch as long as it is smaller than one of its successor elements. We show here how a version of this algorithm acting on concrete data (arrays) can be obtained systematically from one acting on abstract data (labeled trees). The abstract labeled trees can be represented concretely by an array *A* of atoms, where the successor nodes of *A*(*n*) are *A*(2*n*) and *A*(2*n* + 1). For example, the tree



is represented by the array A:

i	1	2	3	4	5	6	7	8	9	10
$A(i)$	a	b	c	d	e	f	g	h	i	j

Now we have to deal with subtrees such as



and these will be represented by *partial* arrays such as A':

i	2	4	5	8	9	10
$A'(i)$	b	d	e	h	i	j

We need to form a notation for such partial arrays by selecting out certain indices from some other array. Let k be the size of the original tree; for simplicity we keep it fixed throughout. Now we define $n\uparrow$, for any $n \geq 1$, to be the set of indices corresponding to the subtree rooted at n; thus

$$\begin{aligned} n\uparrow \;&\Leftarrow \varnothing && \text{if } n > k \\ &\Leftarrow \{n\} \cup (2n)\uparrow \cup (2n+1)\uparrow && \text{otherwise.} \end{aligned}$$

In the example, $2\uparrow = \{2,4,5,8,9,10\}$, the indices of the left-hand subtree.

We call the set of partial arrays with subscripts in $n\uparrow$ *arrays*$_n$; for example A' above is in *arrays*$_2$. We call the set of trees which they represent *trees*$_n$. To be precise, *arrays*$_n$ is the set of functions ($n\uparrow \rightarrow$ *atoms*), and *trees*$_n$ is defined inductively by *trees*$_n$ is {*niltree*} if $n > k$, and otherwise by *trees*$_n$ is the set of all trees of the form *ltree*(a, $t1$, $t2$) where a is an atom, $t1$ is in *trees*$_{2n}$ and $t2$ in *trees*$_{2n+1}$.

An important operation corresponding to taking a subtree of a tree will be taking a subarray of an array. If A is in *arrays*$_n$ and m is in $n\uparrow$, we write A_m for the restriction of A to indices in $m\uparrow$. So, for example, A' in the above example could be written A^2, being the partial subarray rooted at 2. In general, if A is in *arrays*$_n$, then A_{2n} in *arrays*$_{2n}$ and A_{2n+1} in *arrays*$_{2n+1}$ represent the left and right subtrees of the tree represented by A. Clearly if $m \in n\uparrow$, $A_n(m) = A(m)$ and $(A_n)_m = A_m$.

Now for each pair of domains (*arrays*$_n$, *trees*$_n$) we must define a representation function r_n and a coding function c_n inverse to it Thus

r_n: *arrays*$_n \rightarrow$ *trees*$_n$ (representation, concrete to abstract)
c_n: *trees*$_n \rightarrow$ *arrays*$_n$ (coding, abstract to concrete)

They are defined recursively by

$r_n(A) \Leftarrow \textit{niltree}$ if $n > k$ (if $n > k$ then $n\uparrow = \varnothing$ and *arrays*$_n$ contains only the empty array $\varnothing$)
$r_n(A) \Leftarrow \textit{ltree}(A(n), r_{2n}(A_{2n}), r_{2n+1}(A_{2n+1}))$ otherwise
$c_n(\textit{niltree}) \Leftarrow \varnothing$
$c_n(\textit{ltree}(a, t1, t2)) \Leftarrow \{\langle n, a\rangle\} \cup c_{2n}(t1) \cup c_{2n+1}(t2)$

These have the desired property that if $A \in$ *arrays*$_n$, then $c_n(r_n(A)) = A$.

We will not do the whole of the treesort algorithm, but concentrate on the main procedure, which is called "*siftup*," by Floyd. (Since we write our trees with their roots up in the air, we should call it "*siftdown*", we just use "*sift*.") It produces a rearranged tree with the top element moved down a branch so far as possible over elements which are larger than it. (The idea of the algorithm is to get a tree with each branch sorted in order and to maintain this state of affairs when new elements are added, all of this using *sift*.)

There are a number of cases in the definition of *sift*, depending on the relative sizes of the top three elements of the tree. Since they are all similar, we discuss just one case.

The abstract function is

sift: *labeled-trees* $\rightarrow$ *labeled-trees*

In the case $a1 < a2$ and $a2 \geq a3$ it is defined by

$$\begin{aligned}&sift(ltree(a1, ltree(a2, t11, t12),\\ &\qquad ltree(a3, t21, t22)))\\ &\Leftarrow ltree(a2, sift(ltree(a1, t11, t12)),\\ &\qquad\qquad ltree(a3, t21, t22))\end{aligned}$$

We now define a function $SIFT_n$ on concrete data: for each $n \geq 1$,

$SIFT_n$: $arrays_n \rightarrow arrays_n$
$SIFT_n(A) \Leftarrow c_n(sift(r_n(A)))$

This cannot be run as it stands since it uses the abstract *sift*, we wish to transform it into one which can be run.

We consider the typical case $A(n) < A(2n)$ and $A(2n) \geq A(2n + 1)$ (assuming that $2n + 1 \leq k$ so that these elements exist).

$$\begin{aligned}SIFT_n(A) &\Leftarrow \{\langle n, A(2n)\rangle\}\\ &\quad \cup\ c_{2n}(sift(ltree(A(n), r_{4n}(A_{4n}), r_{4n+1}(A_{4n+1}))))\\ &\quad \cup\ c_{2n+1}(r_{2n+1}(A_{2n+1}))\end{aligned}$$

by unfolding with the definitions of r_n, *sift*, and c_n.

(Note that we view partial arrays as functions, i.e. sets of index value pairs, and use union to combine them, $\{\langle n, A(2n)\rangle\}$ is the partial array with just one index n.)

First $c_m(r_m(B)) = B$, for any m and $B \in arrays_m$, so the last term is just A_{2n+1}.

Now to do a fold on *SIFT* we would like the second term to be of the form $SIFT_m(A')$ for some m and some A'. Since it is in $arrays_{2n}$, $m = 2n$; now $SIFT_{2n}(A')$ is

$$c_{2n}(sift(r_{2n}(A'))), \quad \text{i.e. } c_{2n}(sift(ltree(A'(2n), r_{4n}(A'_{4n}), r_{4n+1}(A'_{4n+1})))),$$

so comparing this with the second term,

$$A'(2n) = A(n), \quad A'_{4n} = A_{4n}, \quad A'_{4n+1} = A_{4n+1}.$$

That is, A' is like A_{2n} except that its value for index $2n$ is $A(n)$ instead of $A(2n)$. This suggests that we introduce a substitution operation on arrays such that $B[i/a]$ is an array like B but with value a for its ith element. Formally, $B[i_1/a_1, \cdots, i_n/a_n]$ is an array B' such that $B'(i_1) = a_1, \cdots, B'(i_n) = a_n$ and otherwise $B'(j) = B(j)$. Now we can put $A' = A_{2n}[2n/A(n)]$.

This digression motivates us to rewrite $SIFT_n(A)$ as

$$\begin{aligned}SIFT_n(A) &\Leftarrow \{\langle n, A(2n)\rangle\}\\ &\quad \cup\ c_{2n}(sift(ltree(A'(2n), r_{4n}(A'_{4n}), r_{4n+1}(A'_{4n+1}))))\\ &\quad \cup\ A_{2n+1}\\ &\quad\quad \textbf{where } A' = A_{2n}[2n/A(n)] \quad \text{(eureka)}\\ &\Leftarrow \{\langle n, A(2n)\rangle\}\\ &\quad \cup\ c_{2n}(sift(r_{2n}(A')))\\ &\quad \cup\ A_{2n+1}\\ &\quad\quad \textbf{where } A' = A_{2n}[2n/A(n)]\end{aligned}$$

by folding with the definition of r

$$\begin{aligned}&\Leftarrow \{\langle n, A(2n)\rangle\}\\ &\quad \cup\ SIFT_{2n}(A')\\ &\quad \cup\ A_{2n+1}\\ &\quad\quad \textbf{where } A' = A_{2n}[2n/A(n)]\end{aligned}$$

by folding with the definition of *SIFT*.

This is the required recursive definition for *SIFT*. It operates on partial arrays. The substitution operation corresponds to an assignment to one element of the array.

The key step above, marked with "eureka," involves some tricky forethought and looks hard to mechanize. However it is a preparation for folding just as is the use of associativity in other examples; one might imagine a matching algorithm which has built into it various properties of substitution and uses them to force a fold. We have run the above transformations on our system, but only by supplying the key substitutions as rewrite lemmas. The intuitions behind these manipulations are less complex than our rather barbarous notation would suggest, and this is an open area for research.

But we are still not finished because Floyd's *sift* procedure is iterative, and it is important that one can work iteratively on a single array without copying. The stout-hearted reader may follow the further transformations required; others may skip to Section 9.

First we notice that we can use the properties of substitution to express the three terms in the above definition for $SIFT_n$ using just one array A'':

$$SIFT_n(A) \Leftarrow \{(n, A''(n))\} \cup SIFT_{2n}(A''_{2n}) \cup A''_{2n+1}$$
$$\textbf{where } A'' = A[2n/A(n), n/A(2n)] \qquad \text{(eureka)}$$

(This makes sense because it means "Exchange the nth and $(2n)$-th elements of A and *SIFT* the subarray rooted at $2n$.")

Notice that part of A'' is being replaced by a *SIFT*ed version. This suggests a general operation of replacing the subarray of A rooted at m by B, and we define, for A in $arrays_n$ and B in $arrays_m$ where $m \in n\uparrow$,

$$A +_m B = (A - A_m) \cup B \qquad \text{(eureka).}$$

This enjoys "associativity," which we know to be helpful in getting iterative programs.

$$A +_l (B +_m C) = (A +_l B) +_m C \qquad \text{(lemma).}$$

This + operation enables us to rewrite $SIFT_n$ simply as

$$SIFT_n(A) \Leftarrow A'' +_{2n} SIFT_{2n}(A''_{2n})$$
$$\textbf{where } A'' = A[2n/A(n), n/A(2n)]$$

Now we can analyze the computation of $SIFT_n$ by defining a subsidiary function to describe how it depends on A'',

$$I_n(m, B) \Leftarrow B +_m SIFT_m(B_m) \quad \text{definition} \qquad \text{(eureka)}$$

This produces an array like B, but with the subarray starting at m sifted B must be in $arrays_n$. (This enables us to write

$$SIFT_n(A) \Leftarrow I_n(2n, A'')$$
$$\textbf{where } A'' = \text{as above} \qquad \text{fold.)}$$

Can we transform this definition of I to make it iterative? Yes, if we use associativity of + in the familiar way and do some rather ticklish rewriting of expressions involving + and [ ].

$$I_n(m, B) \Leftarrow B +_m (B'' +_{2m} SIFT_{2m}(B''_{2m}))$$
$$\textbf{where } B'' = B_m[2m/B_m(m), m/B_m(2m)]$$

by unfolding with our last recursive definition of *SIFT*

$$\Leftarrow (B +_m B'') +_{2m} SIFT_{2m}(B''_{2m})$$
$$\textbf{where } B'' = B_m[2m/B_m(m), m/B_m(2m)]$$

by associativity of +

$$\Leftarrow B''' +_{2m} SIFT_{2m}(B'''_{2m})$$
$$\textbf{where } B''' = B[2m/B(m), m/B(2m)]$$

by various niggling properties of +, [ ], and subarray formation (eureka)

$$\Leftarrow I_n(2m, B''')$$

where $B''' = B[2m/B(m), m/B(2m)]$

by folding with definition of I_n.

This definition is iterative and quite simple. The road to it was hard and littered with *eureka*'s. We conclude that the basic transformation method works but runs up against the obscurity which usually bedevils reasoning about data-structure overwriting. We hope to stimulate further research on such reasoning.

9. *Conclusions and Future Work*

We have tried to abstract some general method from the particular tactics incorporated in our previous improvement system (Darlington [5], Darlington and Burstall [6]). Work is continuing, and on the theoretical side the following problems are open at the moment (January 1976).

(i) How wide a class of program improvements falls within the scope of our transformations? Can one obtain any formal characterization of this class?

(ii) What are necessary and sufficient conditions that guarantee that our transformations produce an improvement? Can we indeed make a general argument on the basis of the one for Fibonacci in Appendix 2?

On the practical side, work is continuing with the implemented system to investigate the behavior of different strategies. We would like to mechanize the generalization of old definitions to new ones, not just as outlined in Section 5 but also where the new definition needs an extra parameter, as in Section 6, or where totally new definitions are needed, as in Appendix 1. We would also like to look at the problem involved in structuring the optimization of large programs.

The system can be simply extended to achieve the synthesis of algorithms from their implicit (nonexecutable) definitions. Darlington [7] gives simple examples of this. Recently the same author has used this technique to investigate the structure of classes of algorithms by attempting to synthesize all algorithms in a class from a common high level definition. The first class to be investigated was the sorting algorithms, and so far six well-known sorting algorithms have been synthesized (manually) from one high level definition.

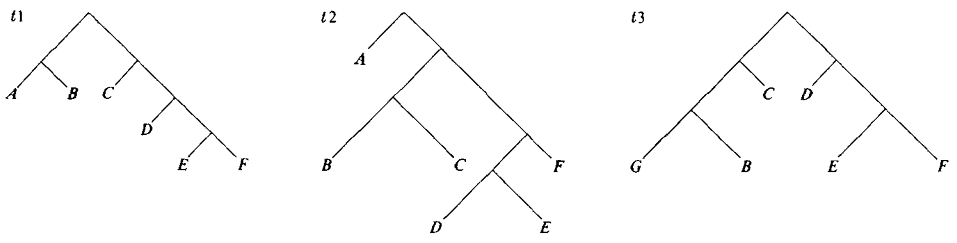
We should mention relevant work by other people. Courcelle and Vuillemin [4] provide a mathematically rigorous treatment of an inference system for a simple recursive language. Manna and Waldinger [12], in their work on program synthesis, independently develop a rule similar to our folding rule, although their presentation of the underlying ideas is rather different. In a more general way our work is akin to work by Gerhart [9] on transformations, to the Harvard work on program manipulation (Cheatham and Wegbreit [3]), and to the large literature on optimization techniques in compilers.

Appendix 1. *Testing Trees for Equality of Frontiers*

This is an example where the obvious definition may compute values which are never needed, a problem proposed originally to illustrate the usefulness of coroutines. We have no coroutine facility in our recursion equation language, but we can achieve a similar economy in computation, although in a rather less general way.

The problem is to test whether two given binary trees have the same frontier, where the frontier of a tree is the list of its tips. Thus in Figure 1 the trees $t1$ and $t2$ are equal in this sense, but $t1$ and $t3$ are not. A natural approach is to define the desired testing function *eqtree* in terms of a function *frontier* which produces a list from a tree, getting $(A\ B\ C\ D\ E\ F)$ for $t1$ and $(A\ B\ C\ D\ E\ F)$ for $t2$, and also a function *eqlist* to test whether these two lists are equal. But then for $t1$ and $t3$ we foolishly compute the whole of $(A\ B\ C\ D\ E\ F)$ and $(G\ B\ C\ D\ E\ F)$ before noticing that they disagree in the very first element. We will try to obtain an improvement which avoids this.

We need a data type atom, from which we derive a data type tree, using constructor functions *tip* to indicate a tip and *tree* to combine two subtrees



frontier(*t*1)	= (*A B C D E F*)
frontier(*t*2)	= (*A B C D E F*)
frontier(*t*3)	= (*G B C D E F*)
eqtree(*t*1, *t*2)	= *eqlist*(*frontier*(*t*1), *frontier*(*t*2)) = *true*
eqtree(*t*1, *t*3)	= *eqlist*(*frontier*(*t*1), *frontier*(*t*3)) = *false*

FIG 1 Trees

tip: *atoms* → *trees*
tree: *trees* × *trees* → *trees*

We also need lists of atoms and of trees, so for any type *alpha* let

nil ∈ *alpha-lists*
cons· *alphas* × *alpha-lists* → *alpha-lists*

We again write $x :: X$ for *cons*(x, X).

We make a habit of specifying the type of each new function, using the usual notation f: $S \rightarrow T$, although this is outside our formalism.

We first define some auxiliary functions, then the main function *eqtree* (Figure 1) which tests trees for equality of their frontiers.

concat: *alpha-lists* × *alpha-lists* → *alpha-lists* (concatenation, *alpha* is any type)

We again write $X \langle\rangle Y$ for *concat*(X, Y)

1. *nil* ⟨⟩ Y ⇐ Y
2. $(x :: X) \langle\rangle Y$ ⇐ $x :: (X \langle\rangle Y)$

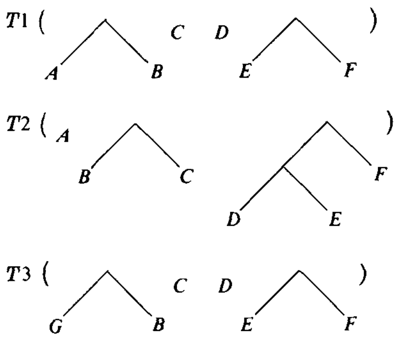
   eqlist: *atom-lists* × *atom-lists* → *truth values* (list equality)
3. *eqlist*(*nil*, *nil*) ⇐ *true*
4. *eqlist*(*nil*, $y :: Y$) ⇐ *false*
5. *eqlist*($x :: X$, *nil*) ⇐ *false*
6. *eqlist*($x :: X$, $y :: Y$) ⇐ *eq*(x, y) and *eqlist*(X, Y)

   where *eq* tests equality of atoms

   frontier: *trees* → *atom-lists* (list of atoms at tips of tree)
7. *frontier*(*tip*(a)) ⇐ a :: *nil*
8. *frontier* (*tree* (t1, t2)) ⇐ *frontier*(t1) ⟨⟩ *frontier*(t2)

   eqtree: *trees* × *trees* → *truth values* (tree equality, same tip sequence)
9. *eqtree*(s, t) ⇐ *eqlist* (*frontier* (s), *frontier* (t))

If we now try to improve *eqtree* by the methods used above, we have no success. To overcome this we introduce a more general function, *EQTREELIST* (see Figure 2) which tests whether two *lists of trees* have the same tip sequence. (We use upper case for variables taking lists of trees as values and for functions taking lists of trees as arguments, analogous to but distinct from the variables and functions in lower case.) The motive here is that as we decompose a tree the current state is some cross section across the tree, but this is just a list of subtrees In the coroutine method these subtrees would be there behind the scenes, associated with coroutine activations. We have to make them vulgarly



FRONTIERLIST(*T*1) = ((*A B*) (*C*) (*D*) (*E F*))
FRONTIER(*T*1) = *flatten*(*FRONTIERLIST*(*T*1))
= (*A B C D E F*)
FRONTIER(*T*2) = (*A B C D E F*)
EQTREELIST(*T*1, *T*2) = *eqlist*(*FRONTIER*(*T*1), *FRONTIER*(*T*2))
= *true*

FIG 2 Tree lists

explicit. Again we need an auxiliary function *FRONTIER* to give the list of atoms at the tips of the whole list of trees; thus *FRONTIER*: *tree-lists* → *atom-lists* (see Figure 2).

10. *FRONTIER*(*T*) ⇐ *flatten*(*FRONTIERLIST*(*T*)) (eureka)

where *FRONTIERLIST* takes a list of trees to the list of their individual frontiers *FRONTIERLIST*: *tree-lists* → *atom-list-lists*

11. *FRONTIERLIST*(*nil*) ⇐ *nil*
12. *FRONTIERLIST*(*t* :: *T*) ⇐ *frontier*(*t*) :: *FRONTIERLIST*(*T*)

and *flatten* takes this list of lists to a list of atoms, by concatenating its elements *flatten*: *atom-list-lists* → *atom-lists*

13. *flatten*(*nil*) ⇐ *nil*
14. *flatten*(*l* :: *L*) ⇐ *l* ⟨⟩ *flatten*(*L*)
EQTREELIST: *tree-lists* × *tree-lists* → *truth values* (equality for tree lists)
15. *EQTREELIST*(*S*, *T*) ⇐ *eqlist*(*FRONTIER*(*S*), *FRONTIER*(*T*))

Now we can use transformations to redefine *eqtree* in terms of *EQTREELIST* since a tree is a singleton list of trees. We need a lemma, *l* ⟨⟩ *nil* = *l*.

16. *eqtree*(*s*, *t*) ⇐ *eqlist*(*frontier*(*s*), *frontier*(*t*)) repeat of 9
⇐ *eqlist*(*frontier*(*s*) ⟨⟩ *nil*, *frontier*(*t*) ⟨⟩ *nil*) lemma about ⟨⟩ (eureka)
⇐ *EQTREELIST*(*s* :: *nil*, *t* :: *nil*) fold 13, 14, 11, 12, 10, 15

Now let us improve *EQTREELIST*. It is most clear if we start on *FRONTIER*, transforming each equation.

17. *FRONTIER*(*nil*) ⇐ *flatten*(*FRONTIERLIST*(*nil*)) instantiate 10
⇐ *nil* unfold 11 and 13
18. *FRONTIER*(*tip*(*a*) :: *T*) ⇐ *flatten*(*FRONTIERLIST*(*tip*(*a*) :: *T*))
instantiate 10
⇐ (*a* :: *nil*) ⟨⟩ *flatten*(*FRONTIERLIST*(*T*))
unfold 12, 7, 14
⇐ (*a* :: *nil*) ⟨⟩ *FRONTIER*(*T*) fold 10

19. $FRONTIER(tree(t1, t2) :: T) \Leftarrow flatten(FRONTIERLIST(tree(t1, t2) :: T))$
instantiate 10
$\Leftarrow (frontier(t1) \langle\rangle frontier(t2)) \langle\rangle flatten(FRONTIERLIST(T))$
unfold 12, 8, 14
$\Leftarrow frontier(t1) \langle\rangle (frontier(t2) \langle\rangle flatten(FRONTIERLIST(T)))$
associativity of $\langle\rangle$
$\Leftarrow FRONTIER(t1 :: (t2 :: T))$
fold 14, 12, 14, 12, 10

(Notice that the use of associativity here requires some insight since an alternative step is to fold with 10 immediately, which does not give the result we want)

Finally we use this new definition of *FRONTIER* to improve *EQTREELIST* itself, and thus improve *eqtree* which uses it

20. $EQTREELIST(nil, nil) \Leftarrow true$ instantiate 15, unfold 17, 3
21. $EQTREELIST(tip(a) :: S, nil) \Leftarrow false$ instantiate 15, unfold 18, 17, 2, 5
22. $EQTREELIST(nil, tip(b) :: T) \Leftarrow false$ similarly
23. $EQTREELIST(tip(a) :: S, tip(b) :: T) \Leftarrow eq(a, b)$ and $eqlist(FRONTIER(S), FRONTIER(T))$
instantiate 15, unfold 18, 2, 1, 6
$\Leftarrow eq(a, b)$ and $EQTREELIST(S, T)$
fold 15
24. $EQTREELIST(tree(s1, s2) :: S, T) \Leftarrow eqlist(FRONTIER(s1 :: (s2 :: S)), FRONTIER(T))$
instantiate 15, unfold 19
$\Leftarrow EQTREELIST(s1 :: (s2 :: S), T)$
fold 15
25. $EQTREELIST(S, tree(t1, t2) :: T) \Leftarrow EQTREELIST(S, t1 :: (t2 :: T))$
similarly

20–25 give a direct recursive definition of *EQTREELIST* with no auxiliary functions. This reduces each tree from the left-hand end as far as necessary, as shown in the example in Figure 3 24 and 25 are used in any order (nondeterministically) until the first tip in each treelist is reached; then the tips are compared using 23, which stops the whole process immediately if they are not equal. 20–22 cope with the *nil* cases. Execution of *EQTREELIST* is radically different from that using the original definition 9, which built up the concept in a well-structured but computationally inefficient way involving unnecessary computation of large intermediate lists.

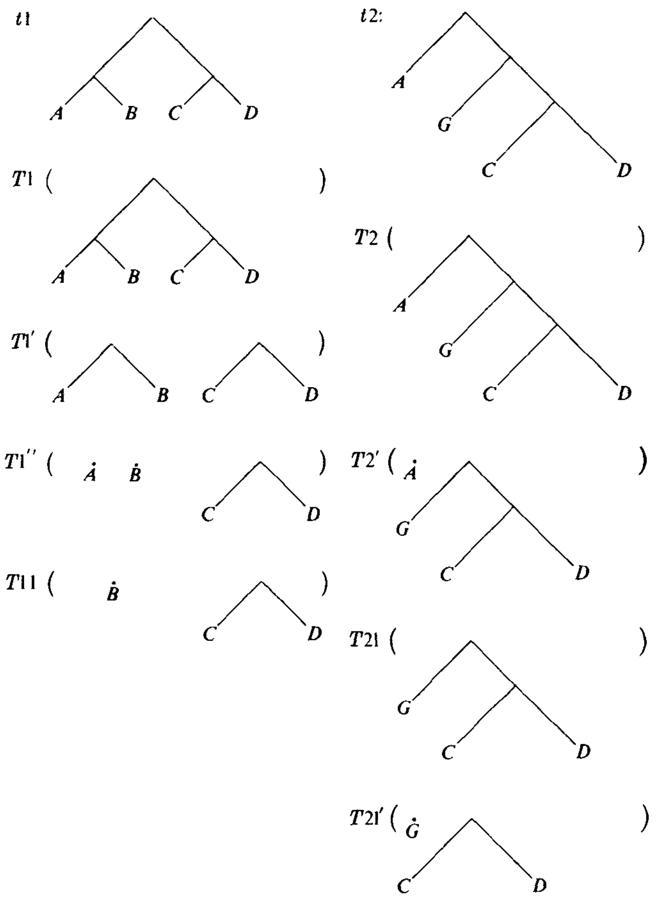
Appendix 2. Improving the Fibonacci Function

To examine whether a sequence of transformations by our rules improves the efficiency of the program, let us try to prove that this is the case for the Fibonacci example. This should throw light on the principles involved without requiring an elaborate and imperspicuous formal apparatus

First we rewrite the transformations, giving a subscript to distinguish each new function symbol as we define it, since these variants, although they may not differ in meaning, certainly differ in efficiency.

We concentrate on the auxiliary function g, which is defined in terms of the original Fibonacci function f.

$g(x)$	$\Leftarrow \langle f(x + 1), f(x)\rangle$	definition
$g_1(0)$	$\Leftarrow \langle f(1), f(0)\rangle$	instantiate
$g_2(0)$	$\Leftarrow \langle 1, 1\rangle$	unfold
$g_3(x + 1)$	$\Leftarrow \langle f(x + 1 + 1), f(x + 1)\rangle$	instantiate



eqtree($t1$, $t2$) = *EQTREELIST*(*T*1, *T*2) = *EQTREELIST*(*T*1′, *T*2)
= *EQTREELIST*(*T*1″, *T*2) = *EQTREELIST*(*T*1″, *T*2′)
= *eq*(*A*, *A*) *and EQTREELIST*(*T*11, *T*21)
= *EQTREELIST*(*T*11, *T*21) = *EQTREELIST*(*T*11, *T*21′)
= *eq*(*B*, *G*) *and* . = *false*

FIG 3 Using improved definition of *eqtree*

$g_4(x+1)$	$\Leftarrow \langle f(x+1)+f(x), f(x+1)\rangle$	unfold
$g_5(x+1)$	$\Leftarrow \langle u+v, u\rangle$ **where** $\langle u, v\rangle = \langle f(x+1), f(x)\rangle$	abstraction
$g_6(x+1)$	$\Leftarrow \langle u+v, u\rangle$ **where** $\langle u, v\rangle = g(x)$	folding
$g_{7,2}(x+1)$	$\Leftarrow \langle u+v, u\rangle$ **where** $\langle u, v\rangle = g_{7,2}(x)$	folding

Notice that folding is done in two steps, first replacing an instance of the right-hand side of the original g equation by the left-hand side, which still leaves g_6 ultimately defined in terms of f, then replacing this by a recursion. We call the new function $g_{7,2}$ because we wish to imply that the equation for g_2 is to be used when $x = 0$.

Let us now write $\phi[n]$ to denote the number of arithmetic operations needed to compute the value of the function symbol ϕ, using its equations, for the number n as argument. For Fibonacci this is the number of additions (ignoring +1, successor).

Now instantiation and unfolding do not affect the number of operations, so

$$g_2[0] = g_1[0] = g[0] \quad \text{and} \quad g_4[x+1] = g_3[x+1] = g[x+1].$$

By a trivial induction $f[x+1] \geq 1$ if $x \geq 1$, so $g_5[x+1] < g_4[x+1]$ if $x \geq 1$ (that is, **where**-abstraction makes an improvement).

Clearly the first stage of folding does not affect the number of operations, so $g_6[x+1]$

$= g_5[x + 1] < g[x + 1]$ if $x \geq 1$. We wish to show from this that $g_{7,2}[x + 1] < g[x + 1]$ if $x \geq 1$. But for this it is easy to show by induction that for all $x \geq 0$,

$$\begin{aligned} g_{7,2}[x] &< g[x] \quad \text{if } x \geq 2, \\ &\leq g[x] \quad \text{if } x = 0 \text{ or } x = 1. \end{aligned}$$

Base. Immediate if $x = 0$ or $x = 1$.

Step. Suppose $x \geq 1$ and $g_{7,2}[x] \leq g[x]$; we need to show that $g_{7,2}[x + 1] < g[x + 1]$. But the equation for $g_{7,2}$ is just like that for g_6 with $g_{7,2}(x)$ for $g(x)$. By our hypothesis that $g_{7,2}[x] \leq g[x]$ we have $g_{7,2}[x + 1] \leq g_6[x + 1]$. But we already have $g_6[x + 1] < g[x + 1]$, so $g_{7,2}[x + 1] < g[x + 1]$.

To summarize, we have proved directly that **where**-abstraction makes an improvement and that folding preserves it (in fact it amplifies it by doing it at each level of the recursion)

In general one can see that the improvements are introduced by **where**-abstraction or rewriting lemmas, and also that folding will preserve any such improvements provided that the base case is no worse and that the argument of the equation used in the substitution is lower in some well-founded ordering than that of the equation undergoing the fold.

ACKNOWLEDGMENTS Independent work by J Moore and Raymond Aubin extending the Boyer-Moore theorem prover to handle essentially iterative programs that accumulate a result helped us develop the technique of translating recursion outlined in Section 6. We would like to thank M. Paterson and D Scott for their contributions described in Section 7 and C.A.R. Hoare for suggesting that we look at the representation function problem which we tackled in Section 8. We have had valuable conversations with our colleagues R. Aubin, G. Plotkin, and R. Topor J. Schwarz helped us with the Treesort example R Boyer and J Moore influenced us a great deal, as did J. Vuillemin. Our recent work on synthesis was stimulated by Z. Manna and R. Waldinger. Members of IFIP Working Group 2.3 provided a valuable stimulus. Clearly we owe much to E. Dijkstra for the influence of his structured programming ideas. Finally we would like to thank the referees (J. Morris and A.N. Other) for their very helpful and perceptive comments and E. Kerse for typing.

REFERENCES

1 AUBIN, R Some generalisation heuristics in proofs by induction Proc IRIA Symp Proving and Improving Programs, Arc-et-Senans, France, 1975, pp 197–208
2 BOYER, R S , AND MOORE, JS Proving theorems about LISP functions *J ACM 22*, 1 (Jan 1975), 129–144
3 CHEATHAM, T E JR , AND WEGBREIT, B A laboratory for the study of automating programming Proc AFIPS 1972 SJCC, Vol 40, AFIPS Press, Montvale, N.J., pp 11–21
4 COURCELLE, B , AND VUILLEMIN, J Semantics and axiomatics of a simple recursive language Proc Sixth Annual ACM Symp Theory of Comptg , 1974, pp 13–26
5 DARLINGTON, J A semantic approach to automatic program improvement Ph D Th , Dep Artif Intel , U of Edinburgh, Edinburgh, 1972
6 DARLINGTON, J , AND BURSTALL, R M A system which automatically improves programs Proc Third Int Joint Conf on Artif Intel , Stanford, Calif , 1973, pp. 479–485. (To appear in *Acta Informatica*, 1976.)
7 DARLINGTON, J Application of program transformation to program synthesis Proc IRIA Symp Proving and Improving Programs, Arc-et-Senans, France, 1975, pp 133–144
8 FLOYD, R.W Algorithm 245, TREESORT 3 *Comm ACM 7*, 12 (Dec 1964), 701–702
9 GERHART, S L Correctness-preserving program transformations Conf Rec Second Symp Principles of Programming Languages, Palo Alto, Calif , 1975, pp 54–66
10 HOARE, C A R Proof of correctness of data representations *Acta Informatica 1* (1972), 271–278
11 KNUTH, D E Structured programming with go to statements *ACM Computing Surveys 6*, 4 (1974), 261–301
12. MANNA, Z., AND WALDINGER, R Knowledge and reasoning in program synthesis *Artif Intel J 6*, 2 (1975), 175–208

13 MOORE, JS Introducıng ıteratıon ınto the pure LISP theorem prover CSL-74-3, Xerox Palo Alto Res Ctr , Palo Alto, Calıf , 1975
14 PLOTKIN, G Buıldıng ın equatıonal theorıes *Machıne Intellıgence 7,* B Meltzer and D. Mıchıe, Eds , Edınburgh U Press, Edınburgh, 1972, pp 73-90
15 TOPOR, R W Interactıve program verıficatıon usıng vırtual programs Ph D Th , Dep Artıf Intel , U of Edınburgh, Edınburgh, 1975

RECEIVED JULY 1975; REVISED JANUARY 1976