TRACKING THE ELUSIVE EUREKA

by

Eric Norman

Computer Sciences Technical Report #636

March 1986

Tracking the Elusive Eureka

Eric Norman

University of Wisconsin

ABSTRACT

The algebraic approach to computation of FP is used to transform recursive definitions of the Fibonacci and factorial functions into iterative code. The approach is similar to the one used to solve many differential equations and also similar to what many call "higher order" or "semantic" unification. An algorithmic procedure is not obtained; the intent is to demonstrate the usefulness of the algebraic approach.

Key words and phrases: algebraic programming, functional programming, recursion elimination, semantic unification, program transformation, FP.

1. Introduction.

For many years, researchers in computer science have been interested in techniques for optimization of code. One area of particular interest is elimination of recursion; the idea is to generate iterative code (code that does not need arbitrary stack space) from recursive definitions. Particularly noteworthy is the approach described by Burstall and Darlington in [B&D77]; however, they rely on the user of the recursion removal system to supply certain key steps, which they call "eurekas".

In this paper, I present an alternative approach, which breaks a eureka down into smaller eurekas and consequently makes them easier to find. The approach is similar to what one does to solve difference equations and differential equations; one writes down a general solution, substitutes it in the equation, does some algebra to discover some parameters of the general solution, and then checks it against the boundary conditions to discover the remaining parameters. With this approach, what is ordinarily a eureka can fall out naturally in the algebraic manipulations required to determine the general solution's parameters.

I will use removal of recursion to demonstrate the potential of an algebraic approach to computation. We will not end up with an algorithmic method, although we will see enough of a pattern to indicate the general strategy of an algorithm. What we're about to do is akin to high school algebra. The only difference is that we're working with what we interpret as functions and operations that combine those functions instead of numbers and their operations. The idea is the same, though; we have some symbols and some rules (algebraic laws like those below) that we can use to shuffle symbols around. There seems to be a similarity between what we're doing here and what we do when solving differential equations. Differential equations is one of the branches of mathematics where we operate on functions instead of numbers; i.e., we have functional operators such as D_x which we apply to other functions and that yield functions as results. We will find that the analogy doesn't stop there. Specifically, one of the first tricks we learn in differential equations is to write down a general solution, substitute that in the differential equation, do some algebra to discover some particulars

Author's address: Eric Norman, Computer Sciences Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 53706.

about the general solution, and check that against the boundary conditions to discover the remaining particulars.

One of the common approaches to recursion elimination is through the use of program transformations [D&B73, H&L78]. The compiler has available a library of schemata for recursive definitions and their corresponding iterative schemata. If the recursive schema can be matched with a portion of the program, then that portion can be replaced with the corresponding iterative schema *mutatis mutandis*.

There are some disadvantages of this approach. First, where does the knowledge that an iterative schema is computationally equivalent to a recursive schema come from? We can look at them in many cases and say "it's obvious"; we can always prove that they're equivalent, but what is it that causes us to write them down in the first place? In other words, can the process of developing an iterative schema from a recursive schema be automated so that a compiler can do it on the spot instead of having to rely on "divine inspiration" from a library?

Second, there are many cases where the ability to turn recursive definitions into iterative code depends on the *particular* properties of the operations involved in the definition and not on just its "shape". For example, generation of iterative code for a definition may only be possible if a certain operation therein is commutative, or if it has an inverse, etc. Specification of such properties of operations is difficult to do within schemata that are matched against the program by what is essentially syntactic unification. We need semantic unification also. "Semantic unification" is a term that is used in the area of logic programming, see [S&Y84]; the term "narrowing" is also used [G&M84]. For an example of semantic unification, consider the following (almost) Prolog definition of the factorial function:

$$fact(0,1).$$

$$fact(N+1,R) \;\; if \;\; fact(N,X), R \;\; is \;\; (N+1)*X.$$

If the query is $fact(3,What)$ then 3 will have to be unified with $N+1$, which normally could not be done. This requires solution of the equation $N+1 = 3$ which in turn requires knowledge about what "+", "3", and "1" mean. It is possible to effect this with syntactic unification by knowing that "3" can be rewritten as "2+1", which allows us to unify X with 2; it is also possible to rely on methods other than unification that have knowledge about numbers and their properties. Although the latter approach may not necessarily be a disadvantage, the approach taken here *attempts* to obviate it.

In the next section, we will introduce the notation, which is FP. In section 3, we will try various strategies to solve the Fibonacci function. In section 4, we do the factorial function. Section 5 will derive a tail-recursive form of the exponential notation that was been used in sections 3 and 4.

2. Notation

The notation used in this paper differs slightly from that described in [Bac78] in two respects. First, instead of naming the selector functions with integers, they will be named $p1$, $p2$, etc. You can read "p" as "part", or "pick" if you wish. It really stands for **projection**, which is what it really is (well not quite, we're still using the same name for functions with different domains).

Second, we use exponential notation to indicate repeated composition of functions. This will be done in such a matter that it can be considered merely an abbreviation for something much longer that we would rather not write. In other words, exponential notation will be used as a shorthand within, but not a change to, FP.

The main algebraic rules of FP that we will be using are these:

$$(A \circ B) \circ C = A \circ (B \circ C) \qquad \text{associative}$$

$$A \circ id = A = id \circ A \qquad \text{identity}$$

$$[A,B] \circ C = [A \circ C, B \circ C] \qquad \text{right distributive}$$

$$(A \rightarrow B; C) \circ D = (A \circ D \rightarrow B \circ D; C \circ D)$$

$$p1\circ[A,B] = A \qquad \text{how projections work}$$
$$p2\circ[A,B] = B.$$

Note that I am being cavalier about questions of strictness; this is part of the informality. You can assume that we're talking about an FP system that evaluates from the outside in, or lazily, if that helps.

3. The Fibonacci function.

Here's the first recursively defined function that we'll try to solve, the Fibonacci function. We will characterize it thus:

$$F_0 = \bar{0}$$
$$F_1 = \bar{1}$$
$$F_{n+2} = +\circ[F_{n+1},F_n].$$

Note that this is really an infinite number of equations (a tabulation of the results of the Fibonacci function); it characterizes an infinite number of functions. Each F_x computes the value of the Fibonacci function for the argument x. Actually, each of these functions is a constant function; it returns the same result for all arguments in its domain. Later, we will construct another function that agrees with each of these functions at the appropriate domain-codomain pair; this will be the function that we want. This function will be equivalent to the following, which is a *recursive* definition of the Fibonacci function in legal FP:

$$f = (eq0 \rightarrow \bar{0};\ eq1 \rightarrow \bar{1};\ +\circ[f\circ sub1\circ sub1, f\circ sub1])$$

The reason we're doing it with the F_x's is to get the combining form **conditional** out of the way. If we tried to work with it, the algebra would get very messy.

What we want to do is write each F_x in iterative form. As mentioned in the introduction, one of the first tricks we learn in differential equations is to write down a more or less general solution; that is, we write an expression that looks like what we think the solution ought to be, except that it has undetermined parts (i.e. variables). Then we substitute this expression in the differential equation, do some algebra, and discover details about the undetermined parts.

The convention we will use is that capital letters represent as yet undetermined function-valued expressions; in other words, they are the variables we will use during "unification". Lower case letters and special symbols represent constants (i.e., specific functions). Note that the F_x's above are such variables. We want to get to the point where we can write each F_x as some expression involving constants that is also in iterative form.

3.1. Fibonacci, first attempt.

Now, what does a function look like if it's in iterative form? One way we iterate is to build an initial state from the argument with some function S, then we apply a transformation G (another function) x times, where x is the argument, and finally we extract the result out of the resulting state with yet another function A. Here's how this could be written in FP:

$$F_x = A\circ G^x\circ S. \qquad (1)$$

G^x is an abbreviation of x-fold **composition** of G; e.g., G^3 means $G\circ G\circ G$. (1) abbreviates the series of equations

$$F_0 = A\circ S$$
$$F_1 = A\circ G\circ S$$
$$F_2 = A\circ G\circ G\circ S$$

and so forth. What we are going to do is substitute this expression for F_x in the definitions above and then do some algebra to see what we can discover about A, G, and S. In the beginning, we will

go very slowly and justify each step. This is so that you can get a feeling for the algebra involved; later we will pick up the pace. First we substitute into the recurrence relation:

$$A \circ G^{n+2} \circ S = +\circ[A \circ G^{n+1} \circ S, A \circ G^n \circ S].$$

We can use the distributive law to factor the S's out of the **construction** to give

$$= +\circ[A \circ G^{n+1}, A \circ G^n] \circ S$$

and then rewrite the $n+1$-fold **composition** of G to yield

$$= +\circ[A \circ G \circ G^n, A \circ G^n] \circ S$$

and we can now use the distributive law again on G^n and also factor G^{n+2} to give

$$A \circ G^2 \circ G^n \circ S = +\circ[A \circ G, A] \circ G^n \circ S.$$

Now we cancel the common expressions off the right. Note that we are working backwards here; if the equation below represents a solution, then the equation above certainly does also since **composition** of equals are equal. There is no guarantee that this will lead to a solution; we may be cancelling out something that is essential to the solution. For example, we may end up with each F_χ equal to 0, which does satisfy the recurrence relation, but it doesn't satisfy the initial values. Nevertheless, it seems like a reasonable thing to try. We now have

$$A \circ G^2 = +\circ[A \circ G, A]. \tag{2}$$

We have certainly simplified matters. By eliminating the $G^n \circ S$, we have left what is in some vague sense the essence of the iterative loop. We could call (2) the *characteristic equation* of the recurrence relation. Since we want to find out about G, it might help if it only appeared on one side of the equation. Unfortunately, we can't use the same distributive law to do any more factoring — or can we? A is as yet undetermined. What if we assumed that it is of the form

$$A = B \circ G \ ? \tag{3}$$

B is a new variable that we are introducing so that A has the desired form. In other words, we are speculating about the details of A. Now, an obvious thing to try would be to substitute this for all occurrences of A, but if we do that, we'll just get equation (2) back again, albeit with A renamed to B (try it). What we do instead is substitute for only the rightmost A; it's the one causing the problem; thus

$$A \circ G^2 = +\circ[A \circ G, B \circ G].$$

Now we can use the distributive law again to get

$$A \circ G \circ G = +\circ[A, B] \circ G$$

and cancelling another G gives

$$A \circ G = +\circ[A, B] \tag{4}$$

which has G isolated on one side as desired. Since we now just have a **composition** on each side, we might guess that the functions being composed can be identified; i.e., we can do *syntactic* unification now. This gives the partial solution

$$A = +$$

$$G = [A, B]$$

$$= [+, B].$$

Now we can work on B. We can substitute our solutions for A and G in (3) to yield

$$+ = B \circ [+, B]$$

and then make the two sides equal by picking the + out of the right side with the guess

$$B = p1$$

which gives us

$$G = [+, p1].$$

G is the function that represents "once around the loop". What this says is that each time around the loop, we add two variables (the $+$) and copy the first to the second (the $p1$ in the second position). Since we all know that this is a way to compute the Fibonacci function with iterative code, we have reason to believe we're getting somewhere.

In [B&D77], Burstall and Darlington start with the definitions

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n+2) = f(x+1) + f(x)$$

and end with the key results

$$f(x+2) = u+v \textbf{ where } <u,v> = g(x)$$
$$g(0) = <1,0>$$
$$g(x+1) = <u+v,u> \textbf{ where } <u,v> = g(x)$$

where they define g with the "eureka" step

$$g(x) = <f(x+1), f(x)>.$$

Compare our G with their $<u+v,u>$; they're both saying exactly the same thing about how to compute the next iteration. What happened to the "eureka"? If it appears above, it seems to be difficult to find. We certainly don't have any step above that *exactly* corresponds to their "eureka". They needed to guess that a pairing of two values will be appropriate and needed to guess the assignment of positions in that pairing; the pairing that we arrived at was there from the beginning as what we needed to do to form an argument for $+$. (In all fairness, they remark that this is what motivates them to guess their g as they did, also). We, however, were able to derive the assignments (**projections**) of components of this pairing and in the next section we will derive them in an even "stronger" sense. We also did not do anything what looks like unfolding and folding, with the possible exception of (3), which might be an unfolding in some sense. It seems to me that the use of exponential notation essentially unfolds everything at the beginning. Note also that we did not do the "inductive step" of substituting $n+1$ for n as they do. We will do this below, though.

So far, we have solved the recurrence equation and discovered the body of the iterative loop. Again, just as in differential equations, we now need to worry about the boundary conditions. In this case, they're called initial values, or where we terminate the recursion. Use of the initial values for F_x at 0 and 1 should hopefully let us determine what S has to be. We begin by assuming that the iterative form (1) also works for 0 and 1. For 0, this gives us

$$F_0 = +\circ G^0 \circ S.$$

Since zero **compositions** of G are the identity function, this becomes

$$= +\circ id \circ S$$

and since the identity function is an identity for **composition** we have

$$\bar{0} = +\circ S.$$

The $+$ indicates that S is a pair of numbers so we can expand it by inventing names for its components with

$$S = [S1, S2]$$

which makes

$$\bar{0} = +\circ[S1, S2]. \qquad (5)$$

We'll just remember this for awhile and go on to the value of F_x at 1. Assuming that the iterative form works at 1 gives

$$F_1 = +\circ G^1 \circ S$$

in which we can finish substituting what we know already to get

$$\bar{1} = +\circ[+,p1]\circ[S1,S2]$$

and then use the distributive law (the other way this time) to yield

$$= +\circ[+\circ[S1,S2],p1\circ[S1,S2]]$$

in which we can simplify the application of $p1$ to get

$$= +\circ[+\circ[S1,S2],S1].$$

Now we notice that the left component of the outermost **construction** can be rewritten using (5) to make

$$\bar{1} = +\circ[\bar{0},S1]$$

from which we conclude

$$S1 = \bar{1}$$

since 0 is the identity for addition. Going back to (5), we now have

$$\bar{0} = +\circ[\bar{1},S2]$$

which gives

$$S2 = \overline{-1}.$$

Now we have all variables specified and the result is

$$F_x = +\circ[+,p1]^x\circ[\bar{1},\overline{-1}].$$

If we try this out on a few values for x, we find that is does indeed compute the Fibonacci function. However, something doesn't seem quite right. What's that -1 doing in there? We shouldn't need any -1's to compute the Fibonacci function.

What happened was that we were either too general or too quick. When we initially wrote an expression for an iterative form for F_x, we were just about as general as possible. This generality lead us to solve for A prematurely a while back in (4). We ended up with A, which is the function we use to extract the answer from the final state, being the same as the main function of the loop. This means that we do it (A) one too many times and to adjust for this we have to start the initial state (S) one "farther back". John Williams encountered essentially the same problem in [Wil82] when he needed the notion of "overrun tolerance". In this case, it doesn't hurt, but if we were to try the same thing for the factorial function (see Section 4), we would find ourselves needing to divide by zero.

Anyway, I did it this way on purpose. I think it's useful to see what happens when we go in the wrong (but almost right) direction. Furthermore, the main purpose of the above was to introduce examples of the algebra and the thought process involved. We're going to do it again, go a lot faster, but use a slightly different tactic.

3.2. Fibonacci, second attempt.

What we're going to do is not generalize quite so much. Instead of using any arbitrary function to extract the final result, we'll insist that it be a projection. I'm not that happy with this since it certainly seems possible that there are recursive equations where some final processing needs to be done beyond mere selection of the answer from the state; for instance, we may do a parallel iteration through two sequences and then need to combine the two final results for the answer. Since I'm being quite informal, I won't worry about it now.

We'll start again with an iterative form except change A to P to emphasize that it's a projection. The iterative form is

$$F_x = P \circ G^x \circ S \tag{6}$$

which we substitute into the definition and then do exactly as above down through (4) which looks like

$$\begin{aligned} P \circ G^{n+2} \circ S &= + \circ [P \circ G^{n+1} \circ S, P \circ G^n \circ S] \\ P \circ G^2 &= + \circ [P \circ G, P] \qquad (7) \\ P &= B \circ G \qquad (8) \\ P \circ G^2 &= + \circ [P \circ G, B \circ G] \\ P \circ G \circ G &= + \circ [P, B] \circ G \\ P \circ G &= + \circ [P, B]. \qquad (9) \end{aligned}$$

Since we have G isolated, we would like to get rid of a P. Instead of trying to figure out a way to cancel the one on the left, we get rid of the one on the right by turning $[P,B]$ into the identity (for the domain of two component **constructions**); i.e., we make it $[p1,p2]$ by setting

$$\begin{aligned} P &= p1 \\ B &= p2 \\ p1 \circ G &= + \circ [p1, p2] \\ &= + \circ id \\ &= + \end{aligned}$$

which would make G a **construction** that looks like

$$G = [+, G2].$$

Substituting back in (8), we have

$$\begin{aligned} p1 &= p2 \circ [+, G2] \\ &= G2 \\ G &= [+, p1] \end{aligned}$$

which is the same G we arrived at before (this is not supposed to be a surprise). Our insistence that P be a projection turns out to have been overly strict. All we did was handle (9) differently and derived P as an appropriate projection.

Now we'll handle the boundary conditions. We'll find that having P be a projection simplifies matters here. For 0 we have

$$\begin{aligned} F_0 &= p1 \circ [+, p1]^0 \circ S \\ \bar{0} &= p1 \circ [S1, S2] \\ \bar{0} &= S1 \end{aligned}$$

and so much for $S1$. Then for 1 we get

$$\begin{aligned} F_1 &= p1 \circ [+, p1]^1 \circ [\bar{0}, S2] \\ \bar{1} &= p1 \circ [+, p1] \circ [\bar{0}, S2] \\ \bar{1} &= + \circ [\bar{0}, S2]. \end{aligned}$$

Hence,

$$S2 = - \circ [\bar{1}, \bar{0}] = \bar{1}.$$

Note that the $\bar{1}$ in the final result is not "the same" $\bar{1}$ as F_1; it is $F_1 - F_0$. This is not the same as the "overrun" above where we had to go *beyond* the boundary. Here we do the subtraction as the last step *at* the boundary. However, this still isn't quite right. We can compute the Fibonnaci function by having F_0 and F_1 as the two components of the initial state, so let's make one more quick try.

3.3. Fibonacci, third time's the charm.

The trick is to still get rid of things on the right side by making them the identity function, but to do it earlier. Namely, we can isolate both G and P by proceeding from (7) rather than (9) with

$$\begin{aligned} P \circ G &= p1 \\ P &= p2 \\ p2 \circ G &= p1 \\ G &= [G1, p1] \\ p2 \circ [G1, p1] \circ [G1, p1] &= + \circ id \\ p1 \circ [G1, p1] &= + \\ G &= [+, p1]. \end{aligned}$$

For the boundary conditions we get

$$\begin{aligned} p2 \circ S &= \bar{0} \\ S &= [S1, \bar{0}] \\ p2 \circ [+, p1] \circ S &= \bar{1} \\ p1 \circ S &= \bar{1} \\ S &= [\bar{1}, \bar{0}]. \end{aligned}$$

At last we have the solution

$$F_n = p2 \circ [+, p1]^n \circ [\bar{1}, \bar{0}].$$

The solution above is more general that it appears. I could have used arbitrary numbers for F_0 and F_1 and the above would still be a solution since we never used any of the properties of these numbers (e.g., that 0 is the identity of addition). Neither did we use any properties of addition other than it is a binary function. In short, we have derived iterative code for the generalized Fibonacci function.

4. The factorial function.

Now we'll try to transform a recursive definition of the factorial function into iterative form. Although there will be differences from what we did above, there are also many similarities. Here's the characterization of factorial:

$$\begin{aligned} F_0 &= \bar{1} \\ F_{n+1} &= \times \circ [\overline{n+1}, F_n]. \end{aligned}$$

We use the same iterative form as (6) above and begin by substituting it into the recurrence relation as before to get

$$P \circ G^{n+1} \circ S = \times \circ [\overline{n+1}, P \circ G^n \circ S].$$

Now we immediately seem to be stuck. We have a $G^n \circ S$ that we would like to take out of the **construction** but we don't have an indeterminant function as the other component that we can assume is of this form. Well, who said it *has* to be indeterminant? We're doing semantic unification here. Semantically, and still like we did above in (2), let's assume that $n+1$ can be computed by n applications of G to S followed by A (which we invent) and continue like so:

$$\begin{aligned} \overline{n+1} &= A \circ G^n \circ S \\ P \circ G \circ G^n \circ S &= \times \circ [A \circ G^n \circ S, P \circ G^n \circ S] \\ P \circ G &= \times \circ [A, P]. \end{aligned} \tag{10}$$

Again, just as above, we make $[A,P]$ disappear by turning it into the identity, thus:

$$\begin{aligned} A &= p1 \\ P &= p2 \\ p2 \circ G &= \times \\ G &= [G1, \times]. \end{aligned}$$

Now we have part of G. For the other part, we put what we know in (10) to get

$$\overline{n+1} = p1 \circ [G1, \times]^n \circ S. \tag{11}$$

Now we substitute $n+1$ for n which gives

$$\overline{n+2} = p1 \circ [G1, \times]^{n+1} \circ S.$$

What's happening above and below looks very similar to what we do in an inductive proof (see comments above, also). Anyway, we can get $n+2$ another way. We can also apply the function *add*1 to both sides of (11) to get

$$\begin{aligned} add1 \circ \overline{n+1} &= add1 \circ p1 \circ [G1, \times]^n \circ S \\ \overline{n+2} &= add1 \circ p1 \circ [G1, \times]^n \circ S. \end{aligned}$$

Please note that *add*1 is injective so that when working frontwards (up), we can cancel it from the left in an equation involving **composition.** Note also that we are using properties of numbers and *add*1 now as well as properties of combining forms.

One of the advantages of this algebraic approach seems to be that taking advantage of known properties (commutativity, associativity, existence of identities, etc.) of the primitive functions is done the same way that we take advantage of the properties of the combining forms (**construction, composition**, etc). In either case, we merely use algebraic laws to rewrite terms.

Anyway, we can equate the two ways of computing $n+2$, which gives

$$\begin{aligned} p1 \circ [G1, \times]^{n+1} \circ S &= add1 \circ p1 \circ [G1, \times]^n \circ S \\ p1 \circ [G1, \times] &= add1 \circ p1 \\ G1 &= add1 \circ p1 \\ G &= [add1 \circ p1, \times]. \end{aligned}$$

Now for the boundary conditions. From the definition of F_0 we obtain

$$\begin{aligned} F_0 &= p2 \circ G^0 \circ S \\ \bar{1} &= p2 \circ S \\ S &= [S1, \bar{1}] \end{aligned}$$

and substituting 0 in (11) gives

$$\begin{aligned} \bar{1} &= p1 \circ G^0 \circ [S1, \bar{1}] \\ &= p1 \circ [S1, \bar{1}] \\ &= S1 \end{aligned}$$

which finally results in

$$F_x = p2 \circ [add1 \circ p1, \times]^x \circ [\bar{1}, \bar{1}].$$

Again, the two $\bar{1}$'s above are only coincidentally the same. One is F_0 while the other is the initial value for a counter. Although we used properties of numbers and *add*1 above, we never used any properties of multiplication. The $\times$ above was just a symbol and could be interpreted as any two argument function. Other transformations that have been published [D&B73] require that multiplication be (semi) associative. The reason that this is not required here is that the multiplication is being done "the other way"; i.e., we're doing just like the recursive definition says and computing

$$F_n \times (\cdots \times (F_2 \times (F_1 \times F_0)))$$

instead of

$$(((F_n \times F_{n-1}) \times F_{n-2}) \times \cdots) \times F_0.$$

5. The finished function.

In both cases above, what we did was solve a series of recursive definitions to get a series of iterative functions. For each n, we have a function of the form

$$f_n = p \circ g^n \circ s$$

which agrees with the recursive definition of f_n at argument n. We're using lower case for f, p, g, and s now since we no longer consider them functional expressions whose substructure is to be determined; they're just symbols for arbitrary functions. In other words, we now no longer care about particular recursive functions (like Fibonacci and factorial); we have any function that can be expressed in the exponential notation above. As mentioned in section 2, the above is really an abbreviation for a series of definitions, one for each n. What we want is a *single* function. We want a single function F which, if applied to any natural number n, will compute the same result as f_n would. We can express this in FP by

$$F \circ \bar{n} = f_n \circ \bar{n} = p \circ g^n \circ s \circ \bar{n}. \tag{12}$$

As mentioned above, the exponentiation in g^n is not part of FP. It is merely a notational extension that was introduced, albeit a *very* useful one. In any case, I suppose we could stop right here since everybody already knows how to implement the above with a loop, but we're interested in deriving things today so let's see what we can do to discern what such an F might look like.

We need the combining form **conditional** to do this. We need to test the argument and "pick out" the right F_n. One way we can do this is to put a "counter" for n in the initial state. We still need the state built by s also; therefore, we'll start with a new initial state that we build with $[id,s]$ and assume that we have a function L that will do what we want if applied to this initial state. That is, we want a function L, which will involve p and g, that if applied to any initial state $[\bar{n},s \circ \bar{n}]$, will compute $F \circ \bar{n}$. In symbols, this property is stated

$$L \circ [id,s] \circ \bar{n} = p \circ g^n \circ s \circ \bar{n}. \tag{13}$$

This is our first condition on L; L must satisfy (13) for arbitrary s, but L itself must be free of s's; it can only contain appearances of p and g.

Our second condition on L is that it be an iterative function. This would happen if it were tail-recursive. Therefore, the definition of L should look like

$$L = (A \rightarrow B; L \circ C). \tag{14}$$

Now we can consider how to compute F_{n+1}. If we substitute $n+1$ for n in (12), we get

$$F \circ \overline{n+1} = p \circ g^{n+1} \circ s \circ \overline{n+1}$$

$$F \circ add1 \circ \bar{n} = p \circ g^n \circ (g \circ s \circ add1) \circ \bar{n}. \tag{15}$$

Since L must obey (13) for arbitrary s, the right side of (15) can be rewritten as $L \circ [id, g \circ s \circ add1] \circ \bar{n}$. Another way of saying this is that if we want to compute a result for $n+1$, we can take n, add 1 to it, form the initial state, run that around the loop *once* (this is what $g \circ s \circ add1$ does), and then do the same thing (i.e., L) to the resulting state that we would do with n. We can also compute $F \circ n+1$

with $L\circ[id,s]\circ add1\circ\bar{n}$. Equating these two methods, we have

$$L\circ[id,g\circ s\circ add1] = L\circ[id,s]\circ add1$$

so if we expand the L on the right (just once, and note that we *are* unfolding now), we get

$$\begin{aligned} L\circ[id,g\circ s\circ add1] &= (A \rightarrow B; L\circ C)\circ[id,s]\circ add1 \\ &= (A\circ[add1,s\circ add1] \rightarrow B\circ[add1,s\circ add1]; L\circ C\circ[add1,s\circ add1]) \end{aligned}$$

by using a distributive law for **conditional.** In order to simplify this, and remembering that we are restricting the domain of F to natural numbers, we guess

$$\begin{aligned} A &= eq0\circ p1 \\ eq0\circ p1\circ[add1,s\circ add1] &= eq0\circ add1 \\ &= \overline{\overline{false}} \end{aligned}$$

which simplifies the **conditional** so that we now have

$$L\circ[id,g\circ s\circ add1] = L\circ C\circ[add1,s\circ add1].$$

Now we work on C with

$$\begin{aligned} C &= [C1,C2] \\ L\circ[id,g\circ s\circ add1] &= L\circ[C1,C2]\circ[add1,s\circ add1] \\ &= L\circ[C1\circ[add1,s\circ add1],C2\circ[add1,s\circ add1]] \end{aligned}$$

and we can identify the components of the **constructions** to get

$$\begin{aligned} id &= C1\circ[add1,s\circ add1] \\ C1 &= sub1\circ p1 \\ g\circ s\circ add1 &= C2\circ[add1,s\circ add1] \\ C2 &= g\circ p2. \end{aligned}$$

All that's left is B, which we can discover by substituting 0 for n in (12); i.e.,

$$L\circ[\bar{0},s\circ\bar{0}] = p\circ g^{0}\circ s\circ\bar{0}$$

and since

$$eq0\circ p1\circ[\bar{0},s\circ\bar{0}] = \overline{true} = A\circ[\bar{0},s\circ\bar{0}]$$

we can expand L and simplify the **conditional** to get

$$\begin{aligned} B\circ[\bar{0},s\circ\bar{0}] &= p\circ s\circ\bar{0} \\ B &= p\circ p2. \end{aligned}$$

Ergo, our finished F is

$$F = L\circ[id,s]$$

where

$$L = (eq0\circ p1 \rightarrow p\circ p2; L\circ[sub1\circ p1,g\circ p2]).$$

If, for instance, we put what we found out about the factorial function in this, we will fianlly have the following definition

$$\begin{aligned} fact &= L\circ[id,[\bar{1},\bar{1}]] \\ L &= (eq0\circ p1 \rightarrow p2\circ p2; L\circ[sub1\circ p1,[add1\circ p1,\times]\circ p2]). \end{aligned}$$

Q.E.F.

Note that the state that we iterate around the loop is isomorphic to one containing three components; it starts as $[\bar{n},[\bar{1},\bar{1}]]$. There are two "counters" therein; one counting up and the other counting down. One is used to terminate the loop, the other to keep track of what to multiply by. In this respect, the fact that we did the multiplication the other way is not quite as efficient as other transformations. Although, if the initial state (s) contains *id* and if it's always accessed with $sub\,1\circ px$, then we can use it as the counter and not have to construct one.

6. Arm waving proofs.

Since we made lots of guesses above, it behooves us to prove that the iterative solutions that result really do satisfy the recursive specifications. In a somewhat jocular sense, I can claim that they've already been proved; just turn the paper upside down! For instance, to prove the validity of the iterative form of the Fibonacci function, we would substitute the iterative form for the F_x's in the recursive definition, do some algebraic massaging, and verify that the equations are indeed satisfied by the iterative form. Analogously, we verify a proposed solution to a differential equation by plugging it back in and doing the differentiation and algebra.

The point is that if this is done, the algebraic manipulations will be essentially what you get if you read upwards. Verifying the finished function seems to require an inductive proof, but the same observation still holds. The key step in the inductive part is essentially the relationship above between $F\circ\bar{n}$ and $F\circ\overline{n+1}$. I will confess that I knew what the answer was and did the inductive proof in order to find that relationship. Before that I made lots of poor guesses; there seems to be plenty of room for them in this case. However, as was mentioned above, we probably would not want a compiler to effect the derivation of the finished function. Once we have instantiated the variables in the exponential form (1) or (6), we know how to generate code.

There does seem to be an observation to made here. If we have a recursive schema and a computationally equivalent iterative schema and we are trying to find out how the latter can be derived from the former, we can get hints by looking at the proof of equivalence and inverting it. In loose words, deriving a program is sort of the inverse of proving it.

7. Relation to other work.

The above relationship between deriving and proving is very much like the strategy used by Manna and Waldinger to synthesize programs [M&W80]. They attach *output expressions* to assertions about the input and goal of the output and then construct a proof that the assertions imply the goals. During construction of this proof, the *output expressions* become instantiated to code that can be used to transform the inputs into the outputs.

The work of Bellegarde [Bel84], Givler and Kieburtz [G&K84], Goguen and Meseguer [G&M84], and many others is also relevant. What we need to do during semantic unification is solve equations. If we have a confluent and terminating set of rewrite rules for the operations involved, equations can be solved by reducing each side to normal form and then doing *syntactic* unification. Unfortunately, the use of **conditional** makes a set of rewriting rules with these properties difficult to obtain. The phrase used by Givler and Kieburtz is "its projected running time spans the lifetimes of stars". The use of exponentiation to eliminate it creates non-FP, so the rewrite rules for FP less **conditional** aren't usable. I do not know whether rewrite rules can be expanded to effectively handle exponentiation.

The work of Cohen [Coh83] also seems similar to what we're doing here. What we compute with S and manipulate with G is essentially what he calls a *tabulation*. A major difference, though, is that the use of exponentiation here is quite limited. That is, we begin by making the assumption that a recursive function of n can be computed by iterating a loop n times. We are making a *significant* simplifying assumption about what Cohen calls *descent conditions*; namely, we are limiting them to the function $sub\,1$. If we wished to handle arbitrary descent functions, we might, for instance, characterize the Fibonacci function thus:

$$F\circ\bar{0} = \bar{0}$$

$$F\circ\bar{1} = \bar{1}$$

$$F\circ\bar{n} = +\circ[F\circ sub\,1^2, F\circ sub\,1]\circ\bar{n}.$$

This has not been investigated any farther.

On the other hand, the advantage of the approach here is that we were able to derive iterative functions with mere "source to source" transformations; we did not need to leave the realm of FP and appeal to DAG's or descent trees.

8. Retrospection and Conclusions.

It seems to me that one of the most interesting observations to be made about what was done above is that both the Fibonacci and factorial functions were handled in very similar ways. The only essential difference seems to be the "induction" that we did to solve (11). This is a consequence of not having sufficient algebraic laws to deal with exponents. We always worked from "first principles" above. It is possible to derive new algebraic laws that allow us to move exponents in and out of constructions. Such laws have not been investigated enough to be included herein, but here are some examples:

$$[A\circ B^n\circ C, D\circ E^n\circ F] = [A\circ p1, D\circ p2]\circ[B\circ p1, E\circ p2]^n\circ[C, F]$$

$$p1\circ[A\circ p1, B]^n\circ[C, D] = A^n\circ C.$$

If we write $\overline{n+1}$ as $add1^n\circ\bar{1}$, the second law above allows a rapid solution of (11).

An obvious question to be asked about the two different iterative methods of computing the factorial function is whether the result that we derived can be algebraically transformed into the other. The author is embarassed by the fact that he cannot supply an affirmative answer.

In any case, the only iteration that we handle is what could be called *primeval* iteration. Not only do we know an upper bound for a loop before we execute it; we know exactly how many times we're going to execute it. There is no mechanism by which a computation within the loop itself can determine how many times it will be executed.

Finally, FP is a tool that seems to be particularly well suited to expressing the relationships that are important in program transformation, and perhaps more. It can be used to deal with the algebra of computation on varying scales. To get an idea how large the range of scales can be, compare **composition**, **construction**, and **conditional** with **join**, **include**, and **or** in [Mar85], where they are applied from a system-wide level down to the most detailed level.

9. Acknowledgements.

I am indebted to Tom Reps for many stimulating conversations, for pointing out some different views of what I am doing here, and for holding a carrot in front of me so that I would stay up long hours to get this written.

10. References.

[D&B73] Darlington, J. & Burstall, R. "A System which Automatically Improves Programs". Proceedings of 1973 International Conference on Artificial Intelligence.

[B&D77] Burstall, R. & Darlington, J. "A Transformation System for Developing Recursive Programs". JACM 24,1. Jan 77.

[Bac78] Backus, J. "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". CACM 21,8. Aug 78.

[H&L78] Huet, G. & Lang, B. "Proving and Applying Program Transformations Expressed with Second-Order Patterns". Acta Informatica 11. 1978.

[M&W80] Manna, Z. & Waldinger, R. "A Deductive Approach to Program Synthesis". TOPLAS 2,1. Jan 80.

[Wil82] Williams, J. "On the Development of the Algebra of Functional Programs". TOPLAS 4,4. Oct 82.

[Coh83] Cohen, N. "Eliminating Redundant Recursive Calls". TOPLAS 5,3. Jul 83.

[Bel84] Bellegarde, F. "Rewriting Systems of FP Expressions that Reduce the Number of Sequences they Yield". Proceedings of the 1984 Conference on LISP and Functional Programming. Aug 84.

[G&K84] Givler, J. & Kieburtz, R. "Schema Recognition for Program Transformations". Proceedings of the 1984 Conference on LISP and Functional Programming. Aug 84.

[G&M84] Goguen, J. & Meseguer, J. "Equality, Types, Modules, and Generics for Logic Programming". Second International Logic Programming Conference. Jul 84.

[S&Y84] Subrahamanyam, P. & You, J-H. "Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming". Proceedings of the 1984 International Symposium on Logic Programming. Feb 84.

[Mar85] Martin, J. *System Design from Provably Correct Constructs.* Prentice-Hall. 1985.