Theory of Linear Equations Applied to Program Transformation

Uday S.Reddy
University of Utah, Salt Lake City
Barat Jayaraman
University of North Carolina at Chapel Hill

Abstract

In this paper is presented a technique for transforming a class of recursive equations called *linear equations* into iterative equations. Linear equations are characterized by involving at the most one recursive call for any invocation. In contrast to the conventional techniques, the scheme of program transformation presented here involves finding the solution of the given linear equation and transforming this solution. The solutions of linear equations can always be expressed using a construct called *abstract sequence.* Two classes of abstract sequence programs are identified: *right-associative* and *left-associative* sequence programs. The former are obtained by solving linear equations and the latter correspond to iterative programs. The task of transforming linear recursive programs into iterative programs is thus reduced to the task of transforming right-associative sequence programs into left associative ones. Various transformation rules are developed based on an algebra of functional programs.

I Introduction

Since the pioneering work of [Darlington 1976, Burstall 19771, program transformation has come to be widely recognized as a program development tool. In this paper, we are interested in the application of program transformation to develop iterative programs from recursive programs. A generalized calculus for such transformations called *unfold-fold* method was given in I Burstall 19771. Given a recursive equation to compute a function *f* one finds another function /', such that

1. there exists a recursive equation for *f* that is in iterative form, and
2. /"can be defined in terms of/"' without using recursion.

Once such a function *f'* is found, the unfold-fold method can be used to systematically develop the recursive equation for *f* from that for *f* The main problem with the use of this method is to find the target recursive function *f.*

We submit that the cause of the problem is that the unfold-fold method attempts to transform one recursive equation into another without regard to what the functions defined by these equations are. If we can find the solution of the given recursive equation, the function *f,* it may be possible to systematically develop the function *f* for which an iterative equation exists. But, unfortunately, it is not known how to express the solution of an arbitrary recursive equation. Even though the solution of any recursive equation can be specified as the limit of a monotonically increasing sequence of functions |Scott 1970, Manna 1972], the limit itself is, in general, not "expressible", i.e., cannot be specified using a closed form expression involving other functions. Backus 119781 initiated the development of notation and theory to formally derive and express the solutions of certain classes of recursive equations. Further developments of this approach can be found in I Backus 1979, Backus 1981, Williams 1982).

In this paper, we shall consider the class *of linear equations* as defined in IBackus 19811. The solutions of these equations can always be expressed using a construct called *abstract sequence* [Reddy 1982a I. Further, the solutions fall into a class of abstract sequence programs called *right-associative sequence programs.* We will identify another class called *left—associative sequence programs* which are equivalent to iterative programs We will then present examples of transformation rules to transform right—associative sequence programs into left associative ones, based on the "algebra of functional programs" introduced in [Backus 1978].

II Notation

We shall use the FP system given in [Backus 19781 as the language for presenting the transformations. An object in an FP system is either the undefined object (I), or an atom or a sequence of objects. The atoms include boolean values (T, F) and numbers. Sequences are enclosed in angular brackets (...) and the empty sequence is denoted by 0. A sequence containing I is equal to 1. All functions accept a single object as argument and yield a single object as result, either or both of which can be sequences. The application of a function on an object is denoted by the operator ":". All functions yield 1 when applied to I (i.e. they are *strict).* Unlike other functional languages, in an FP system, only first-order functions are defined. Using a small set of higher order functions called *functional forms,* functions can be defined without using X-abstraction. Such definitions are called *function-level definitions* in contrast to the *object level definitions* of the λ-calculus style.

Appendix I contains a list of Backus's functions and functional forms which we shall use in this paper. Some new functional forms that will be used are given below.

Abstract Sequence

$$\text{seq}\ r\ p{:}x = \langle x, r^1{:}x, \ldots, r^n{:}x \rangle \quad \text{if } p{:}r^n{:}x = T \text{ and } p{:}r^i{:}x = F \text{ for all } i<n$$

$$\bot \quad \text{if no such } n \text{ exists}$$

Insert from left

$\textbf{insertl}\ h : \langle z, \langle s_1,\ldots,s_n\rangle\rangle = (\ldots((z\ h\ s_1)\ h\ s_2)\ h\ \ldots)\ h\ s_n)$

$\textbf{insertl}\ h : \langle z, \varnothing\rangle = z$

$\backslash h : \langle s_1,\ldots,s_n\rangle = (\ldots(s_1\ h\ s_2)\ h\ \ldots)\ h\ s_n)$

$\backslash h : \langle s_1\rangle = s_1$

$\backslash h : \varnothing =$ left identity of h if it exists and is unique

$\bot$ otherwise

Insert from right

$\textbf{insertr}\ h : \langle\langle s_1,\ldots,s_n\rangle, z\rangle = (s_1\ h\ (s_2\ h\ (\ldots\ h\ (s_n\ h\ z)\ldots))$

$\textbf{insertr}\ h : \langle\varnothing, z\rangle = z$

Cumulate from left

$\textbf{cuml}\ h : \langle z, \langle s_1,\ldots,s_n\rangle\rangle = \langle t_0, t_1, \ldots, t_n\rangle$

where $t_i = \textbf{insertl}\ h : \langle z, \langle s_1,\ldots,s_i\rangle\rangle$

Cumulate from right

$\textbf{cumr}\ h : \langle\langle s_1,\ldots,s_n\rangle, z\rangle = \langle t_0, t_1, \ldots, t_n\rangle$

where $t_i = \textbf{insertr}\ h : \langle\langle s_1,\ldots,s_i\rangle, z\rangle$

Example 1: The factorial function can be defined in FP using these functional forms as

factorial = (**insertr** ×) ° [(**seq** pred eq1), $\bar{1}$]

pred = − ° [id, $\bar{1}$]

eq1 = eq ° [id, $\bar{1}$]

The application of this function to 3, for instance, yields

factorial : 3 = **insertr** × : ⟨**seq** pred eq1:3, 1⟩

= **insertr** ×: ⟨⟨3,2,1⟩, 1⟩

= 6

□

For the sake of convenience, we shall use some purely syntactic extensions to the FP notation. Firstly, we shall use *object-level definitions* as in λ-calculus based languages. Such definitions can be translated into pure FP definitions in much the same way as extended definitions discussed in [Backus 1981]. Another notational extension we shall find useful is infix notation for binary functions. We shall write *f:(x,y)* as *x f y* using infix notation. Prefix applications of functions have precedence over infix applications.

The seq functional form was introduced in [Reddy 1982a] where a sequence yielded by the seq functional form was called an *abstract sequence.* As they play a pivotal role in our manipulations, we shall introduce informal notation to denote abstract sequences. In this notation,

$\langle x_0, \ldots, x_n\rangle$

where $x_0 = t{:}x$

$x_{i+1} = r{:}x_i$

$p{:}x_n$

denotes (**seq** r $p{:}t{:}x$). The symbols n, i, and x_i are all formal parameters in this notation. It is implicitly understood that $p{:}x_i$ does not hold for all $i<n$. Using these syntactic extensions, the definition of factorial in example 1 can be written as

factorial : n = **insertr** ×:⟨⟨n_0, ..., n_k⟩, 1⟩

where $n_0 = n$; $n_{i+1} = n_i - 1$; n_k eq 1

The functions tl, tlr, distl, distr, all g, cuml h preserve abstract sequences, i.e., applying any of them on an abstract sequence produces another abstract sequence that can be expressed using the functional form **seq**. Therefore, we shall use liberalized abstrace sequence notation to denote the expressions in which the above functions are applied on other abstract sequences. For instance,

$\langle x_0, \ldots, x_{n-1}\rangle$

where $x_0 = t{:}x$; $x_{i+1} = r{:}x_i$; $p{:}x_n$

denotes (tl:**seq** r $p{:}t{:}x$).

III Linear Functionals and Canonical Linear Forms

Definition: A *functional* of arity n is a higher-order function, that maps every tuple of n functions into a function. □

A functional is an abstract object. A representation of a functional that can be defined in FP is called a form.

Definition: [Backus 1981] Let V = $\{f_1 \ldots f_n\}$ be a set of (function) variables. Then we shall say that $Ef_1 \ldots f_n$ is a *form*, or E is a form in V, if exactly one of the following holds:

1. $Ef_1 \ldots f_n$ = r for some function r, or
2. $Ef_1 \ldots f_n = f_i$ for some i, or
3. There are forms $E_1 \ldots E_k$ in V and a primitive form (functional form) Γ with k parameters such that $Ef_1 \ldots f_n$ = $\Gamma(E_1,\ldots,E_k)$.

□

Consider a recursive equation of the form

$f = p \rightarrow q; Hf$

where p and q are functions and H is a functional. Using the approach of [Kleene 1952], the solution of such an equation is the limit of the sequence of functions:

$f_0 = \bar{\bot}$

$f_1 = p \rightarrow q;\ H\bar{\bot}$

$f_2 = p \rightarrow q; H(p \rightarrow q; H\bar{\bot})$

$f_3 = p \rightarrow q; H(p \rightarrow q; H(p \rightarrow q; H\bar{\bot}))$

...

Finding the limit of such a sequence of functions is a nontrivial exercise owing to the nesting of the functional H in the approximating functions. If there exists another functional H_t such that

$H(p \rightarrow q; r) = H_t p \rightarrow Hq; Hr$

then, each of the approximating functions can be simplified to

$f_n = p \rightarrow q; H_t p \rightarrow Hq; \ldots ; H_t^{n-1} p \rightarrow H^{n-1} q; H^n \bar{\bot}$

The limit of this sequence of functions can be easily found.

Definition: [Backus 1981] A functional Hf is *linear* if there exists another functional $H_t f$ called its *predicate transformer*, so that

1. for all functions a, b and c,

   $H(a \rightarrow b; c) = H_t a \rightarrow Hb; Hc$
2. for all objects x, if $H\bar{\bot}{:}x \neq \bot$, then for all functions a,

   $H_t a{:}x$ = T

A form that represents a linear functional is a *linear form.* □

Example 2: The following recursive equation for factorial

f = eq0 → $\bar{1}$; × ° [id, f °pred]

involves a linear functional

Hf = × ° [id, f °pred]

with the predicate transformer

$H_t f = f$ ° pred

□

It has been proved by [Backus 1981] that, whenever H is a linear functional, the recursive equation

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

for any p and q, has as its solution, the linear expansion,

$$f = p \longrightarrow q; H_p p \longrightarrow Hq; \ldots; H_p^k p \longrightarrow H^k q; \ldots$$

Backus then goes on to characterize some of the forms that can be constructed in FP as linear. In the following, we shall give a much simpler characterization of linear functionals, which is essentially the same as that of linear recursive schema of [Walker 1973].

Definition: A form of the kind

$$Hf = p \longrightarrow g;\ h \circ [\mathrm{id}, f \circ r]$$

is called a *canonical linear form.* □

It can be verified that a canonical linear form represents a linear functional with the predicate transformer, $H_p f = f \circ r$.

Theorem III.1: [Reddy 1983] A functional is linear if and only if there exists a canonical linear form that represents it. □

Using this theorem, we can also formulate a simple operational test to check if a form is linear.

Corollary III.2: [Reddy 1983] A form H is linear if and only if for all functions b and objects x, either the computation of $Hb{:}x$ involves a b–application on at most one distinct value, or $Hb{:}x = \bot$. □

Example 3: The form used in the factorial equation of Example 2 is a canonical linear form. As a more complex example, consider the one given in [Backus 1981].

$$Hf = a \longrightarrow f \circ b;\ h \circ [(e \longrightarrow d; f \circ g), (i \longrightarrow j; k \circ [m, f \circ g])]$$

It is a linear form, with the predicate transformer

$$H_p f = a \longrightarrow f \circ b;\ \mathrm{and} \circ [e, i] \longrightarrow \mathrm{T}; f \circ g$$

Note that the computation of $Hf{:}x$ for any x would involve only one f–application, either on $b{:}x$ or on $g{:}x$.

The following canonical linear form $H'f$ is equal to Hf.

$$H'f = \alpha \longrightarrow \beta;\ \gamma \circ [\mathrm{id}, f \circ \delta]$$

where

$$\alpha = \mathrm{and} \circ [\mathrm{not} \circ a, \mathrm{or} \circ [e, i]]$$

$$\beta = h \circ [d, j]$$

$$\gamma = a \longrightarrow 2;\ h \circ [(e \circ 1 \longrightarrow d \circ 1; 2), (i \circ 1 \longrightarrow j \circ 1; k \circ [m \circ 1, 2])]$$

$$\delta = a \longrightarrow b; g$$

□

Even though the test given in corollary III.2, cannot be performed by an algorithm, it is possible to mechanically identify most useful linear forms and also to transform them to canonical linear forms.

IV Solutions of Linear Equations

A *linear equation* is a recursive equation of the form

$$f = a \longrightarrow b; Hf$$

where H is a linear form. Using theorem III.1, we can see that every such equation can be rewritten in the form

$$f = p \longrightarrow q;\ h \circ [\mathrm{id}, f \circ r]$$

We shall call the function r, the *reduction function* of this equation, because it reduces the problem of computing $f{:}x$ to that of computing $f{:}r{:}x$. As shown in [Backus 1978], this equation has as its solution, the infinite expansion:

$$f = p \longrightarrow q; p \circ r \longrightarrow q_1; \ldots; p \circ r^n \longrightarrow q_n; \ldots$$

$$\text{where } q_i = /h \circ [\mathrm{id}, r, r^2, \ldots, r^i, q \circ r^{i+1}]$$

If we define

$$R_i = [\mathrm{id}, r, r^2, \ldots, r^i] \text{ for all } i \geq 0$$

then

$$q_i = /h \circ \mathrm{apndr} \circ [\mathrm{tlr}, q \circ 1\mathrm{r}] \circ R_i = \textbf{insertr}\ h \circ [\mathrm{tlr}, q \circ 1\mathrm{r}] \circ R_i$$

$$q = /h \circ \mathrm{apndr} \circ [\mathrm{tlr}, q \circ 1\mathrm{r}] \circ R_0 = \textbf{insertr}\ h \circ [\mathrm{tlr}, q \circ 1\mathrm{r}] \circ R_0$$

The sequences yielded by R_i are called *reduction sequences*. We can now rewrite the solution using the reduction sequences as:

$$f = p \longrightarrow h' \circ R_0; p \circ r \longrightarrow h' \circ R_1; \ldots; p \circ r^n \longrightarrow h' \circ R_n; \ldots$$

$$\text{where } h' = \textbf{insertr}\ h \circ [\mathrm{tlr}, q \circ 1\mathrm{r}]$$

or as

$$f = h' \circ (p \longrightarrow R_0; p \circ r \longrightarrow R_1; \ldots; p \circ r \longrightarrow R_n; \ldots)$$

The infinite conditional in the parentheses is itself the solution of the linear equation

$$R = p \longrightarrow [\mathrm{id}];\ \mathrm{apndl} \circ [\mathrm{id}, R \circ r]$$

Using the functional form **seq** defined as

$$\textbf{seq}\ r\ p = p \longrightarrow [\mathrm{id}];\ \mathrm{apndl} \circ [\mathrm{id}, (\textbf{seq}\ r\ p) \circ r]$$

we can express R without recursion.

Theorem IV.1: [Backus 1978] The solution of the linear equation

$$f = p \longrightarrow q;\ h \circ [\mathrm{id}, f \circ r]$$

is, for all functions p, q, h, and r,

$$f = \textbf{insertr}\ h \circ [\mathrm{tlr}, q \circ 1\mathrm{r}] \circ \textbf{seq}\ r\ p$$

□

Example 4: The definition of factorial in example 1 is nothing but the solution of the factorial equation given in example 2. □

V Left–associative and Right–associative Sequence Programs

Definition: A *right–associative (sequence) program* is one of the two forms

$$f{:}x = \textbf{insertr}\ h : \langle \textbf{seq}\ r\ p{:}x, f_0{:}x \rangle$$

$$f{:}x = /h : \textbf{seq}\ r\ p : x$$

Similarly, a *left–associative (sequence) program* is one of the two forms

$$f{:}x = \textbf{insertl}\ h : \langle f_0{:}x, \textbf{seq}\ r\ p{:}x \rangle$$

$$f{:}x = \backslash h : \textbf{seq}\ r\ p : x$$

□

Note that an abstract sequence can be transformed into both a right–associative program and a left–associative program. So can a program in which a right selector, such as 1r, which is applied on an abstract sequence. For instance:

$$1\mathrm{r} : \textbf{seq}\ r\ p : x = /2 : \textbf{seq}\ r\ p : x$$

$$= \backslash 2 : \textbf{seq}\ r\ p : x$$

Every left–associative program can also be trivially transformed into a right–associative one.

The solution of a linear recursive equation is a right–associative program and *vice versa*. On the other hand, every left–associative sequence program can be transformed into an iterative (tail–recursive) equation. Suppose

$$f{:}x = \textbf{insertl}\ h : \langle f_0{:}x, \textbf{seq}\ r\ p{:}x \rangle$$

We can find an iterative function simply by generalizing $f_0{:}x$.

$$f'{:}\langle y, x \rangle = \textbf{insertl}\ h : \langle y, \textbf{seq}\ r\ p{:}x \rangle$$

$$f{:}x = f' : \langle f_0{:}x, x \rangle$$

The iterative equation for f' is

$f':(y,x) = p:x \longrightarrow h:(y,x);\ f':(h:(y,x),\ r:x)$

If $f_0:x$ is defined by another left–associative program, it can be redefined using another iterative equation. All iterative equations can also be trivially transformed into left–associative programs.

Thus, we have a kind of strong equivalence between linear equations and right–associative programs, on the one hand, and between iterative equations and left–associative programs, on the other. The problem of transforming a linear equation into an iterative one is therefore reduced to the problem of transforming a right–associative program into a left–associative program.

Example 5: We have seen that the solution of the following factorial equation of example 2 is the right–associative program given in example 1. This can be directly transformed to the following left–associative program using the fact that $\times$ is associative and commutative:

factorial:n = **insertl** $\times$: $\langle 1, \langle n_0,\ldots,n_k\rangle\rangle$

This, in turn, can be transformed into an iterative program.

factorial:n = fact:$\langle 1,n\rangle$

fact:$\langle p,n\rangle$ = n eq $1 \longrightarrow p \times 1$; fact:$\langle p \times n,\ n-1\rangle$

□

VI Right–associative–To–Left–associative Transformations

There are basically three methods to transform a right–associative sequence program into a left–associative one:

1. Using a stack
2. Reduction inversion
3. Associative Duals

Using a stack, all right–associative programs can be transformed into left–associative ones. This is not surprising since all recursive equations can be transformed into iterative ones using a stack. We shall concentrate on the latter two methods.

A. Reduction Inversion

If f is defined by the right–associative program

$f{:}x$ = **insertr** h : $\langle R{:}x,\ f_0{:}x\rangle$

$R{:}x$ is the reduction sequence of the corresponding linear equation and is defined as an abstract sequence. If rev:$R{:}x$ can be defined as an abstract sequence *revR*:x, then $f{:}x$ can be defined by the left–associative program

$f{:}x$ = **insertl** (h ° swap) : $\langle f_0{:}x,\ revR{:}x\rangle$

where swap:$\langle x,y\rangle = \langle y,x\rangle$

This is the simplest technique to use when the reduction function has an inverse. Suppose

$R{:}x = \langle x_0,\ldots, x_n\rangle$

where $x_0 = x;\ x_{i+1} = r{:}x_i;\ p{:}x_n$

and the reduction function has an inverse g so that

$r{:}g{:}x = x$ whenever $p{:}x$ = F

If the last element of the reduction sequence $\langle x_0,\ldots,x_n\rangle$ is $t{:}x$, then,

revR:x = **all** 1 : $\langle\langle y_0,x\rangle,\ldots,\langle y_n,x\rangle\rangle$

where $y_0 = t{:}x;\ y_{i+1} = g{:}y_i;\ y_n$ eq x

The function $t{:}x$ can, in turn, be defined using the left–associative program

$t{:}x$ = 1r : **seq** $r\ p$: x

However, it is not desirable to have two abstract sequences in the transformed program. So, this transformation should not normally be used unless $t{:}x$ can be defined without using an abstract sequence.

Example 6: Consider the very common reduction sequence

$R{:}n = \langle n_0,\ldots, n_k\rangle$

where $n_0 = n;\ n_{i+1} = n_i - 1;\ n_k$ eq 0

The inverted sequence is

revR:n = **all** 1 : $\langle\langle m_0,n\rangle,\ldots,\langle m_k,n\rangle\rangle$

where $m_0 = 0;\ m_{i+1} = m_i + 1;\ m_k$ eq n

□

It is sometimes possible to use reduction inversion, even if the reduction function does not have an inverse, but has several right–inverses.

Example 7: Consider the more interesting reduction sequence

$R{:}n = \langle n_0,\ldots, n_k\rangle$

where $n_0 = n;\ n_{i+1} = n_i$ div $2;\ n_k$ eq 0

The halve function has two right–inverses:

double:$n = 2 \times n$

doubleadd:$n = 2 \times n + 1$

Let

ceilingpower:n = the smallest 2^k such that $n < 2^k$

This is nothing but one plus the length of the reduction sequence and can be defined by a left–associative program. The reversed reduction sequence can be defined using the two right–inverses.

revR:n = genseq : $\langle n$, ceilingpower:$n\rangle$

genseq:$\langle n,p\rangle$ = **all** 1:$\langle\langle a_0,n_0,p_0\rangle,\ldots,\langle a_m,n_m,p_m\rangle\rangle$

where $a_0 = 0;\ n_0 = n;\ p_0 = p$

$a_{i+1} = n_i \geq (p_i$ div $2) \longrightarrow$ doubleadd:a; double:a

$n_{i+1} = n_i \geq (p_i$ div $2) \longrightarrow n_i - (p_i$ div $2);\ n_i$

$p_{i+1} = p_i$ div 2

p_m eq 1

□

B. Associative Duals

The method of reduction inversion has only limited applicability. The use of associative duals has much wider applicability and forms the main core of our transformation technique. Consider, again, the right–associative program

$f{:}x$ = **insertr** h : $\langle\langle x_0,\ldots, x_n\rangle,\ f_0{:}x\rangle$

We may be able to find a function h', so that

$f{:}x$ = **insertl** h' : $\langle f_0{:}x,\ \langle x_0,\ldots, x_n\rangle\rangle$

Definition: If, for all sequences s,

insertr h : $\langle z,s\rangle$ = **insertl** h' : $\langle s,z\rangle$

then the functions $\langle h', h\rangle$ are said to be *associative duals* with respect to z. h' is called the *left associative dual* of h with respect to z, and h is called the *right associative dual* of h' respect to z. □

Theorem VI.1: [Reddy 1982b] If $\langle h', h\rangle$ are duals with respect to z, then (h°swap, h'°swap) are duals with respect to z. □

The concept of associative duals was introduced by [Kieburtz 1981]. But, their definition differs from ours. We can show that their definition is a sufficient condition for ours.

Theorem VI.2: [Reddy 1982b] If two functions

$h : A \times B \longrightarrow B$, and

$h' : B \times A \longrightarrow B$

satisfy the following conditions, for some $z \in B$,

1. $a\ h\ z = z\ h'\ a \quad \forall a \in A$

2. $a\ h\ (b\ h'\ c) = (a\ h\ b)\ h'\ c \quad \forall a,c \in A$ and $b \in B$

then (h', h) are duals with respect to z. □

Example 8: The functions (apndr, apndl) are duals with respect to the empty sequence, ∅, because

a apndl ∅ = $\langle a \rangle$ = ∅ apndr a

a apndl $(\langle s_1,...,s_n \rangle$ apndr $c)$ = $(a$ apndl $\langle s_1,...,s_n \rangle)$ apndr c

The function rev, for reversing a sequence can be defined by the linear equation

rev:s = s eq ∅ —> ∅; apndr:(rev:tl:s, 1:s)

Its right associative solution is

rev:s = **insertr** (apndr°swap) : ⟨**all** 1:$\langle s_0,...,s_{n-1} \rangle$, ∅⟩

where $s_0 = s$; s_{i+1} = tl:s_i; s_n eq ∅

Since (apndr, apndl) are duals with respect to ∅, the functions (apndl°swap, apndr°swap) are also duals with respect to ∅, by theorem VI.1. Hence,

rev:s = **insertl** (apndl°swap) : ⟨∅, **all** 1:$\langle s_0,...,s_{n-1} \rangle$⟩

□

A special case of duals occurs when an operation h is associative. If it has an identity I, then h is its own dual with respect to I. If it is associative as well as commutative, then it is its own dual with respect to any value. The functions such as +, −, min, and max are examples of such functions.

Theorem VI.3: [Reddy 1982b] If (h', h) are duals with respect to z then $(h'$ ° $[1, k$°$2]$, h ° $[k$°$1, 2])$ are duals with respect to z, for any function k. □

If a powerful set of properties of associative duals, such as the one of theorem VI.3, is found then the use of duals may be a viable tool in transformations. But, currently we do not know enough useful properties of them. Therefore, instead of directly looking for the associative dual of the function used with insertr, we would like to transform the given right-associative program into another right-associative program, so that the technique of duals can be used with the latter.

VII Right-associative-To-Right-associative Transformations

For most programs, the function h used with insertr functional form, is too complicated to have an associative dual. We then transform it into another rights-associative program in which a simpler function h' is used with insertr. The transformations that are possible for a specific function h are highly sensitive to the form of h and the properties that it satisfies. The following rules identify certain widely applicable forms and properties of h. But there may indeed be several others. The proofs of these rules can be found in [Reddy 1982b].

1. If the function h satisfies the property

$a\ h\ (b\ h\ c) = (a\ h'\ b)\ h\ c$

for some function h'

insertr h : $\langle\langle s_1,..., s_n\rangle, z\rangle$

= $(/h'$: $\langle s_1, s_2,...,s_n\rangle)\ h\ z$

The h' may have a dual or may be simpler than h. For example, consider

h = **exp ° swap**

where exp is the exponentiation function.

$a\ h\ (b\ h\ c) = a\ h$ (c **exp** b) = (c **exp** b) **exp** a

= c exp ($b \times a$) = ($a \times b$) h c

The function × is associative and commutative, whereas h is neither.

2. If h is of the form

h:⟨x,y⟩ = h' : ⟨k:x, y⟩

then

insertr h :⟨⟨s_1,...,s_n⟩, z⟩

= **insertr** h' : ⟨⟨k:s_1,...,k:s_n⟩, z⟩

3. If h is of the form

h:⟨x,y⟩ = h' : ⟨x, k:y⟩

and k distributes over h'

k : (y_1 h' y_2) = k:y_1 h' k:y_2

then

insertr h : ⟨⟨s_1,...,s_n⟩, z⟩

= **insertr** h' :⟨⟨k^0:s_1, k^1:s_2, ..., k^{n-1}:s_n⟩, k^n:z⟩

Example 9: Consider the following linear program from [Arsac 1981].

f:⟨n,b⟩ = n eq 0 —> 0;

$10 \times f$:⟨n div b,b⟩ + n mod b

The right associative solution is

f:⟨n,b⟩ = **insertr** h : ⟨⟨⟨n_0,b⟩,..., ⟨n_{p-1},b⟩⟩, 0⟩

where n_0 = n; n_{i+1} = n_i div b; n_p eq 0

h:⟨⟨n,b⟩, y⟩ = $10 \times y$ + n mod b

Since the n mod b part does not depend on y, we can simplify the function h, using rule 2.

f:⟨n,b⟩ = **insertr** h' : ⟨⟨$n_0$mod b,..., n_{p-1}mod b⟩, 0⟩

h':⟨m,y⟩ = $10 \times y$ + m

Let t10 :y = $10 \times y$. Since it distributes over +, using rule 3,

f:⟨n,b⟩ =

insertr + :⟨⟨t10^0:(n_0 mod b),...,t10^{p-1}:(n_{p-1},b)⟩,0⟩

Now, the function + used with **insertr** is associative and commutative. So, f can be defined by the left–associative program (it is not exactly a program yet)

f:⟨n,b⟩ =

insertl +:⟨0,⟨t10^0:(n_0 mod b),...,t10^{p-1}:(n_{p-1},b)⟩⟩

The t10^i factors can be handled by introducing another parameter c_i in the elements of the sequence, so that t10^i:x = c_i×x

f:⟨n,b⟩ =

insertl + :⟨0, ⟨c_0×(n_0 mod b),...,c_{p-1}×(n_{p-1} mod b)⟩⟩

where c_0 = 1; c_{i+1} = $10 \times c_i$

This can now be rewritten using an iterative equation.

f:⟨n,b⟩ = f':⟨0,1,n,b⟩

f':⟨r,c,n,b⟩ = n eq 0 —> r;

f':⟨r + c×(n mod b), 10×c, n div b, b⟩

□

4. This rule is a generalization of 2 and 3 above. Suppose h is of the form

h:⟨x,y⟩ = h':⟨a:x, k:⟨b:x,y⟩⟩

and k is distributive over h' in the second variable position, i.e.,

k:⟨x,(y_1 h' y_2)⟩ = k:⟨x,y_1⟩ h' k:⟨x, y_2⟩

Further, let k be associative and have an identity I. Then

insertr h : ⟨⟨s_1,..., s_n⟩, z⟩

= **insertr** h' : ⟨⟨s'_1,..., s'_n⟩, z'⟩

where z' = k:⟨\k:⟨b:s_1,...,b:s_n⟩, z⟩

$s'_i = k : (\backslash k:(b{:}s_1,\ldots,b{:}s_{i-1}), a{:}s)$

$= \textbf{insertr}\ h' : (\langle k:(c_1, a{:}s_1),\ldots,k:(c_n, a{:}s_n)\rangle, k:(c_{n+1}, z))$

where $c_1 = 1$

$c_{i+1} = k:(c_i, b{:}s_i)$

5. This is a variant of rule 4. Suppose h is of the form

$h:(x,y) = k:(b{:}x, h':(a{:}x,y))$

k is distributive over h'.

$h:(x,y) = h' : (k:(b{:}x,a{:}x), k:(b{:}x,y))$

Further, assume that k is associative. It need not have an identity.

$\textbf{insertr}\ h : (\langle s_1,\ldots,s_n\rangle, z) =$

$\textbf{insertr}\ h':(\langle k:(c_1,a{:}s_1),\ldots,k:(c_n,a{:}s_n)\rangle, k:(c_n,z))$

where $c_1 = b{:}s_1$

$c_{i+1} = k:(c_i, b{:}s_{i+1})$

6. If h is a conditional of the form

$h:(x,y) = p{:}x \longrightarrow y;\ h':(x,y)$

and h' has a left identity 1,

$1\ h'\ y = y$ for all y

then, we can redefine h as

$h:(x,y) = h' : ((p{:}x \longrightarrow 1; x), y)$

7. This is a generalization of rule 6. Suppose h is of the form

$h:(x,y) = p{:}x \longrightarrow h_1:(x,y);\ h_2:(x,y)$

Suppose

$h_1{:}x = k_{11} : (a_{11}{:}x, k_{12} : (a_{12}{:}x,\ldots,k_{1p} : (a_{1p}{:}x, y)\ldots))$

We are only interested in the sequence of functions

$k_{11}, k_{12}, \ldots, k_{1p}$

Let us call them the *embedded sequence of functions* in h_1. There will be a similar sequence for h_2.

$k_{21}, k_{22}, \ldots, k_{2q}$

We need to find a sequence that generalizes these two sequences, so that,

a. by substituting some of the functions in the generalized sequence by the identity function (id"2) we can obtain each of the original sequences, and

b. each of the functions that need to be substituted has a left identity.

We can then construct a function with the generalized embedded sequence using the left identities of the embedded functions, so that this function equals the conditional h. We can extend this method for any number of conditional branches. This scheme of generalization has been used with unfold-fold method by [Arsac 1982].

Example 10: Consider the linear equation

$\text{mult}:(a,b) = b\ \text{eq}\ 0 \longrightarrow 0;$

$\text{even}{:}b \longrightarrow 2\times(\text{mult}:(a, b\ \text{div}\ 2));$

$\text{mult}:(a,b-1) + a$

Its right-associative solution is

$\text{mult}:(a,b) = \textbf{insertr}\ h : (\langle(a,b_0),\ldots,(a,b_{n-1})\rangle, 0)$

where $b_0 = b$

$b_{i+1} = \text{even}{:}b_i \longrightarrow b_i\ \text{div}\ 2;\ b_i - 1$

$b_n\ \text{eq}\ 0$

$h:((a,b),y) = \text{even}{:}b \longrightarrow 2\times y;\ a + y$

The embedded sequences for the conditional branches are $\times$ and $+$ respectively. The generalized embedded sequence can be either $+,\times$ or $\times,+$. Let us choose the former. We can redefine the function h to have this generalized embedded sequence.

$h:((a,b),y) = t:(a,b) + s:(a,b) \times y$

$t:(a,b) = \text{even}{:}b \longrightarrow 0;\ a$

$s:(a,b) = \text{even}{:}b \longrightarrow 2;\ 1$

Since $\times$ distributes over $+$ we can use the rule 4.

$\text{mult}:(a,b)$

$= \textbf{insertr}\ + : (\langle c_0\times t:(a,b_0),\ldots,c_{n-1}\times t:(a,b_{n-1})\rangle, 0)$

where $c_0 = 1$ (left identity of $\times$)

$c_{i+1} = c_i \times s:(a,b_i)$

$= \textbf{insertl}\ + : (0, \langle c_0\times t:(a,b_0),\ldots,c_{n-1}\times t:(a,b_{n-1})\rangle)$

Note that each of the coefficients c_i is a power of 2. Using a "shift left" operation

$k\ \text{sl}\ x = 2^k \times x$

it can be rewritten as

$\text{mult}:(a,b) =$

$\textbf{insertl}\ + : (0, \langle k_0\ \text{sl}\ t:(a,b_0),\ldots, k_{n-1}\ \text{sl}\ t:(a,b_{n-1})\rangle)$

where $k_0 = 0$

$k_{i+1} = \text{even}{:}b_i \longrightarrow k_i + 1;\ k_i$

It is also possible to obtain a program without the shift left operation, by noting that $\times$ commutes with t. See [Reddy 1982b]. [ ]

VIII Discussion

An automatic transformation system can be designed based on the techniques described here. Such a system would have three stages.

1. Rewrite the linear equation using a canonical linear form and solve it.
2. If the reduction sequence can be inverted, then use it to produce a left-associative program. Otherwise, apply rights-associative-to—right-associative transformations, until the right-associative function is sufficiently simple to have a dual.
3. Transform the left-associative sequence program into an iterative equation or equivalently a loop.

The stages 1 and 3 can be done algorithmically, whereas the stage 2 handles a hard problem. We envisage the best approach for stage 2 to be a user—directed transformation system such as that of | Feather 1982].

The main advantage of our transformation scheme over the unfold-fold scheme IBurstall 1977] is that the target recursive function is not guessed (by the so—called *eureka* steps) but results automatically from the transformation of the solution of the source recursive equation. However, Arsac and Kodratoff [1982) have recently suggested a generalization strategy which can be used to guess the target recursive equation based on the form of the source recursive equation. Even though their strategy is radically different from ours, the effects achieved by them are surprisingly close to ours. More investigation to find any possible relationship of our strategy with theirs is worthwhile.

The main drawback of our transformation scheme is that the algebraic properties of the rights-associative function h have to be restated in a form applicable to sequences, so they can be used in right^associative— to-right-associative transformations. The rules given in section VII are such restatements. It is not always clear how the properties can be so restated. The unfold-fold method, on the other hand, directly uses the algebraic properties of the functions involved. Further development of FP algebra may alleviate this problem.

The right-associative-to-right-associative transformations are proved using an inductive proof similar to the unfold-fold method. The rules used in such proofs are 1 and 2 or 1 and 3 of the following.

1. insertr *h* : (,z) = *z*
2. insertr *h* : *(s* apndr *a, z)* - insertr *h* : *(s, a h z)*
3. insertr *h* : *(a* apndl *s, z)* ~ *ah* (insertr *h* : *(s,z))*

This suggests that it may be possible to apply these transformations directly on a recursive equation using unfold-fold, in effect mimicking the transformation of the sequence programs (because every right-associative sequence program is equivalent to a linear equation and *vice versa).* Such a strategy would eliminate the need to restate the algebraic properties of functions in sequence form, and also integrate our technique with the unfold-fold method which is a much more general technique applicable to any recursive transformation.

If our techniques have to be used for equations other than linear equations, methods to express the solutions of those equations must be found. [Williams 19821 was a step in that direction. Further investigation of recursive equation solutions would facilitate the development of transformation techniques for nonlinear equations.

Acknowledgements

We would like to thank Don Stanat, Manton Matthews and Gyula Mago for several discussions and suggestions.

Appendix I

This appendix contains informal definitions of some FP functions and functional forms that are used in this paper.

Basic functions

identity id:x = x

Basic functional forms

constant $\bar{c}:x = c$ $\bar{c}:\bot = \bot$

$\bar{\bot}$ is therefore the everywhere-undefined function

composition $f \circ g : x = f{:}g{:}x$

construction $[f_1,...,f_n] : x = \langle f_1{:}x,...,f_n{:}x\rangle$

condition $(p \rightarrow f;\, g){:}x$ = if $p{:}x$ = T then $f{:}x$ else if $p{:}x$ = F then $g{:}x$ else $\bot$

Functions on sequences

selectors $1{:}\langle x_1,...,x_n\rangle = x_1$; $2{:}\langle x_1,...,x_n\rangle = x$ etc.

right selectors $1r{:}\langle x_1,...,x_n\rangle = x_n$ etc.

tail $tl{:}\langle x_1,...,x_n\rangle = \langle x_2,...,x_n\rangle$

$tlr{:}\langle x_1,...,x_n\rangle = \langle x_1,...,x_{n-1}\rangle$

append $apndl{:}\langle a, \langle x_1,...,x_n\rangle\rangle = \langle\langle a,x_1\rangle,...,x_n\rangle$

$apndr{:}\langle\langle x_1,...,x_n\rangle, a\rangle = \langle x_1,...,x_n,a\rangle$

Functional forms on sequences

apply to all $\alpha f : \langle x_1,...,x_n\rangle = \langle f{:}x_1,...,f{:}x_n\rangle$

insert $/f : \langle x_1,...,x_n\rangle = f : \langle x_1, /f{:}\langle x_2,...,x_n\rangle\rangle$

References

[1) Arsac.J., Kodratoff, Y. 119821, Some techniques for recursion removal from recursive functions, *ACM Trans, on Prog. Lang, and Systems,* 4, 2, 295-322.

[21 Backus, J.W. [1978], Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Cornm. ACM,* 21, 8, 613—641.

[31 Backus, J.W. 11979], *On extending the concept of program and solving linear functional equations,* IBM Research Division, San Jose.

[41 Backus, J.W. [1981], The algebra of functional programs: functional level reasoning, linear equations and extended definitions, (in) *Formalization of programming concepts,* (ed) Diaz and Ramos, Springer-Verlag.

[5] Burstall, R.M., Darlington, J.[1977], A transformation system for developing recursive programs, *Journal of ACM,* 24, 1,44-67.

[6] Darlington, J., Burstall, R.M. [1976], A system which automatically improves programs, *Acta Informatica,* 6,1, 41-60.

[7J Darlington, J. [1978], A synthesis of several sorting algorithms, *Acta Informatica,* 11,1, 1-30.

18] Feather, M.S. [1982], A system for assisting program transformation, *ACM Trans, on Prog. Lang, and Systems,* 4,1, 1-20.

[9|Kieburtz, R.B., Shultis, J. [1981], Transformation of FP program schemes, *1981 Conf. on Functional Programming Languages and Computer Architecture,* ACM, 41—48.

[10] Manna, Z., Vuillemin, J.[19721, Fixpoint Approach to the theory of computation, *Comm. ACM,* 15, 7, 528-536.

[11] Reddy, U.S. [1982aJ, Programming with sequences, *ACM Southeast regional conference,* Knoxville, Tennessee.

[121 Reddy, U.S. [1982b], *Transformation of linear recursive programs using sequences,* Dep. of Comp. Sci., University of North Carolina at Chapel Hill.

[13] Reddy, U.S. [1983], *A simple characterization of linear recursive equations,* Dep. of Comp. Sci., University of Utah, Salt Lake City.

[14] Scott, D.[1970], *Outline of a mathematical theory of computation,* Programming Research Group Tech. Memo. PRG-2, Oxford University Computing Lab.

[15] Walker, S.A., Strong, H.R. [1973], Characterizations of flowchartable recursions, *J. of Computer and System Sciences,* 7,404-447.

[16] Williams, J.H. [1982], On the development of the algebra of functional programs, *ACM Trans, on Prog. Lang, and Systems,* 4, 4, 733-757.