An Experimental Program Transformation and Synthesis System

John Darlington
Imperial College of Science and Technology, South Kensington, Great Britain

Recommended by Erik Sandewall

ABSTRACT

This paper concentrates on the practical aspects of a program transformation system being developed. It describes the present performance of the system and outlines the techniques and heuristics used.

1. Introduction

Program transformation is a way of aiding the production of reliable, efficient programs. Programmers are encouraged to postpone questions of efficiency and first write their programs as clearly as possible. These programs are then transformed (either manually or automatically) into efficient versions by applying transformations that preserve correctness. We have developed a program transformation system based on transformation methods developed with R.M. Burstall. The programming languages used, and the formal transformation rules together with a brief description of the system, can be found in Burstall and Darlington [4]. In Darlington [7] we show how these same methods can be used to accomplish program synthesis, the conversion of non-executable program specifications into runnable programs. This paper concentrates on the more practical aspects of the system showing how it behaves at present, and describing some of the techniques used to achieve this behaviour. We have attempted to make this paper as self-contained as possible, but for a fuller description of the languages and formal transformations used the reader is referred to the earlier papers.

The system is under continuing development, it is viewed more as a research tool for the investigation of program transformation methodologies than a prototype programming assistant, however, the performance of the system at present gives some hope for the ultimate feasibility of such systems, and we discuss several possible developments in Section 7. Section 2 describes the language used and gives the formal transformation rules the system is based upon. Section 3 outlines the basic structures of the system and gives some examples of its use for simple transformations. Section 4 concerns itself with synthesis, Section 5 shows how the system is able to generate auxiliary functions and synthesise functions from equations giving properties of the desired function, and finally Section 6 describes how several features that previously required user intervention have now been automated.

2. Languages and Formalism Used

2.1. The NPL language

We transform programs written in NPL, Burstall [9], a first order recursion equation language that grew out of our work on program transformation, and is being developed as a usable language by Burstall who has written an interpreter and type checker for it. The output of the transformation process is still an NPL program, but hopefully of a radically different form. It is a thesis of ours that a lot of the important transformations can be most easily expressed and achieved within the framework of recursion equations.

One of the novel features of NPL is the use of multiple left hand sides for the equations to eliminate the need for some conditionals and structure accessing functions. For example, for the traditional factorial function, instead of

$$\begin{aligned}\text{fact}(n) \Leftarrow \textit{if}\; n = 0\; \textit{then}\; 1 \\ \textit{else}\; n * \text{fact}(n-1)\end{aligned}$$

one would write

$$\text{fact}(0) \Leftarrow 1$$

$$\text{fact}(n+1) \Leftarrow n + 1 * \text{fact}(n)$$

The syntax of the main part of NPL is

Primitive functions: a set of primitive function symbols, with zero or more arguments; a subset of the primitive functions are the constructor functions.

Parameters: a set of parameter variables.

Recursive functions: a set of recursive function symbols.

Expressions: an expression is built in the usual way out of primitive function symbols, parameter variables and recursive function symbols.

Left hand expressions: a left hand expression is of the form $f(e1, \ldots, en)$, $n > 0$, where $e1, \ldots, en$ are expressions involving only parameter variables and constructor function symbols.

Right hand expressions: a right hand expression is an expression, or a list of expressions, qualified by conditions, or a where expression.

Recursion equations: A recursion equation consists of a left hand expression and a right hand expression, written $E \Leftarrow F$.
Thus an NPL program is a list of equations such as

$$\begin{aligned}
&\text{member}(x1, x :: X) \Leftarrow \text{true} \quad \textit{if}\; x1 = x \qquad \text{(:: is an infix for cons)}\\
&\qquad\qquad\qquad\qquad\quad \text{member}(x1, X) \quad \textit{if}\; x1 \neq x\\
&\text{member}(x1, \text{nil}) \Leftarrow \text{false}
\end{aligned}$$

Two defined functions are important for the transformation system. As an alternative to the *if* construct a three-placed conditional, cond, is defined,

$$\begin{aligned}
&\text{member}(x1, x :: X) \Leftarrow \text{cond}(x1 = x, \text{true}, \text{member}(x1, X))\\
&\text{member}(x1, \text{nil}) \Leftarrow \text{false}
\end{aligned}$$

and a function Maketuple is provided to simulate the action of tuples (Maketuple should be thought of as variadic although it actually takes a list as argument).

$$\begin{aligned}
&\text{maxandmin}(x :: X) \Leftarrow \text{maketuple}([u1, u2]) \quad \textit{if}\; x < u1 \;\textit{and}\; x > u2\\
&\qquad\qquad\qquad\qquad\quad \text{maketuple}([x, u2]) \quad \textit{if}\; x > u1 \;\textit{and}\; x > u2\\
&\qquad\qquad\qquad\qquad\quad \text{maketuple}([u1, x]) \quad \textit{if}\; x < u1 \;\textit{and}\; x < u2\\
&\qquad\qquad\qquad\qquad\qquad \textit{where}\; \text{maketuple}([u1, u2]) = \text{maxandmin}(X)\\
&\text{maxandmin}(x :: \text{nil}) \Leftarrow \text{maketuple}([x, x])
\end{aligned}$$

In addition, NPL allows certain set and logic constructs to appear on the right hand side, for example

$$\begin{aligned}
&\text{setofeven}(X) \Leftarrow \langle : x : \text{in}\; X \;\&\; \text{even}(x) : \rangle\\
&\quad\text{even}(0) \Leftarrow \text{true}\\
&\quad\text{even}(1) \Leftarrow \text{false}\\
&\text{even}(n + 2) \Leftarrow \text{even}(n)\\
&\text{alleven}(X) \Leftarrow \textit{forall}\; x \;\textit{in}\; X : \text{even}(x)\\
&\text{haseven}(X) \Leftarrow \textit{exists}\; x \;\textit{in}\; X : \text{even}(x)
\end{aligned}$$

The deep structure of these expressions is rather interesting, but we postpone its discussion until Section 4.

NPL is a strongly typed language, but as at present the transformation system makes no use of the typing information we will ignore this aspect of NPL.

2.2. Transformation rules

The transformation rules operate on the equations and produce new equations. They are

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

(ii) *Instantiation*: Introduce a substitution instance of an existing equation.

(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 $F \Leftarrow F''$.

(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 $F \Leftarrow F''$.

(v) *Abstraction*: Introduce a *where* clause, by deriving from a previous equation $E \Leftarrow E'$ a new equation

$$E = E'[u_1/F_1, \dots, u_n/F_n] \quad \textit{where}\ \langle u_1, \dots, u_n\rangle = \langle F_1, \dots, F_n\rangle.$$

(vi) *Laws*: Transform an equation by using on its right hand expression any laws we have about the primitives (associativity, commutativity, etc.) obtaining a new equation.

A rule similar to the folding rule was discovered independently by Manna and Waldinger in their work on program synthesis, Manna and Waldinger [13].

The basic strategy the system implements is based on the observation that a sequence of unfoldings, abstractions and applications of laws followed by foldings is bound to produce a dynamic improvement provided that the abstractions and law applications produce a static (textual) improvement.

3. System Structure

The system is written in POP-2, Burstall et al. [6] on the DEC 10 at Edinburgh University. It is written for clarity and flexibility rather than efficiency. Nevertheless it is not very large, occupying approximately 18K of which the NPL system and interpreter takes up 12K. At the moment it is set up as a semi-interactive system, the user supplying help and advice in certain key areas which will be outlined below.

Before describing the structure of the system and giving some examples of its performance we would like to correct any possible misconceptions about its nature. It is not designed as a prototype programmer's assistant that will enable users to develop their programs via transformations, rather it is a research tool for the development of program transformation methodologies. In several of the examples we shall see that some of the interactions require a knowledge of the internal workings of the system or the final program being aimed for. However, we are of the opinion that the best way to develop and test our understanding of such phenomena is by implementing them in working systems. What we would claim is that the behaviour of the present system demonstrates that the present transformation methodology is capable of making significant improvements in program behaviour without requiring excessive computer resources. All the examples shown have been typed directly from telytype output and although we apologise for the verbosity of some of them, we believe they illustrate our point better than suitably edited or invented dialogues. We have been able to do several larger examples on the system but we have not presented them here as the dialogues become rather long. The next stage in the development is to devise methods (both manual and automatic) whereby programmers could sensibly use these methods for program development. We are working on several ways to achieve this, putting more control with the user and these will be discussed briefly in Section 7.

The system has no quantitative means of evaluating the improvements made, relying on the user to accept or reject them, however, the system has an inbuilt bias towards improvement and, as will be seen in the examples, behaves quite purposefully.

We will display the system's basic behaviour by means of a very simple example. Consider the following program

$$\text{append}(\text{nil}, Y) \Leftarrow Y$$

$$\text{append}(x::X, Y) \Leftarrow x::\text{append}(X, Y)$$

$$g(X, Y, Z) \Leftarrow \text{append}(\text{append}(X, Y), Z)$$

g is slightly inefficient as it builds up an intermediate list that it then iterates along. Given that the user has input the above equations to the NPL system, he could affect an improvement by the following dialogue with the transformation system.

Example 1. *Append.*

```
EQNLIST.PREXP;
APPEND([ ], Y)⇐ Y                                }   —print out of original
APPEND(X1:: Y, W)⇐X1::APPEND(Y, W)              }     equations, [ ] is the
G(X, Y, W)⇐APPEND(APPEND(X, Y), W):             }     empty list nil
:
START:                                              —user starts dialogue
TYPE IN INSTANTIATED L.H.S. BASE
CASES FIRST.                                        —system's response
G(NIL, Y, W) END                                    —user
G([ ], Y, W)⇐APPEND(Y, W)                           —system

NOW TYPE IN RECURSIVE BRANCHES                      —system
G(X1::X, Y, W) END                                  —user
```

```
HOW DOES                              }
G(X1::X, Y, W)⇐X1::G(X, Y, W)         }       —system
LOOK TO YOU                           }
OK                                            —user

FINISHED.CPU TIME TAKEN 0 MINS 2 SECS
:
:
:.PREXP;                                      —print out of modi-
G([ ], Y, W)⇐APPEND(Y, W)                       fied equations
G(X1::X, Y, W)⇐X1::G(X, Y, W)                   handed back to
APPEND([ ], Y)⇐Y                                NPL system
APPEND(X1::Y, W)⇐X1::APPEND(Y, W)
```

The system exits with an indication that it has achieved an acceptable improvement and passses back to the NPL system the modified list of equations

$$\text{append}(\text{nil}, Y) \Leftarrow Y$$
$$\text{append}(x :: X, Y) \Leftarrow x :: \text{append}(X, Y)$$
$$g(\text{nil}, Y, Z) \Leftarrow \text{append}(Y, Z)$$
$$g(x :: X, Y, Z) \Leftarrow x :: g(X, Y, Z)$$

Thus we see that the main user responsibilities are to provide the correct instantiations for the functions he wants to improve, and to accept or reject proposed improvements. Each of the instantiations given above could have been a list of instantiations, the system attempting to find an acceptable equation for every instantiation given. In the above dialogue the system comes up with the 'correct' solution first time for each instantiation. If the user rejects any solution the system continues looking at this case until it runs out of improvements to suggest or a pre-set effort bound is exceeded. If no acceptable improvement is found for any case the system immediately exits with a signal of failure and the equation list remains unaltered.

For the base cases the system relies solely on evaluation. It selects the equation required, instantiates it as requested and unfolds the right hand side completely or until the pre-set effort bound is exceeded.

For the other cases the system implements the strategy outlined earlier, viz. sequences of unfoldings, rewritings and abstractions followed by sequences of foldings. This is embodied in the routine DEVELOP which takes the suitably instantiated equation and searches for a fold.

```
develop(eqn) =
            if toodeep( )
              then return
              else reduce (eqn)→eqn;
                forall eqn1 in allwaysofunfolding (eqn)
                  do tryfindfold (eqn1)→eqn1
                    develop (eqn1)
                  od
            fi
```

toodeep controls how much effort is expanded down any particular branch of the recursion. It is true if the depth of recursion exceeds a pre-set depth bound, false otherwise

reduce applies a set of reduction rules to the equation. These reduction rules are supplied to the system as equations with the convention that they are only to be used in a left to right direction. The system has a kernel of reduction rules and these can be added to by the user for particular problems. Amongst the ones used are

$$\text{Cond}(\text{true}, x, y) \Leftarrow x$$

$$\text{Cond}(\text{false}, x, y) \Leftarrow y$$

$$\text{append}(x, \text{append}(y, z)) \Leftarrow \text{append}(\text{append}(x, y), z)$$

We have experimented with various ways of using these reduction rules. In order of sophistication these have been.

(i) Applying each applicable reduction rule just once at each iteration. This simple method suffices for a lot of smaller examples but for larger ones one has to tailor the rules to fit the transformations required.

(ii) Using the rules to keep the developing equation in a normal form. This is not always successful as the form required is dictated more by the need to find a fold rather than any global criteria.

(iii) Using the rules in a 'means-end' fashion. The rewritings are used to direct the developing equation to a foldable form. Earlier versions of the system used a matching routine, due to Topor [15] whereby certain properties of functions such as commutativity, and associativity were built into the matching routine and only applied when necessary.

The last method is the one to be ultimately preferred giving the most intelligent behaviour but for the moment the first method is preferred for simplicity and was used for most of the examples presented here.

allwaysofunfolding returns a list of the equation given plus all versions of the equation that can be produced by one unfolding using the equations on the present list. A switch enables the user to restrict this to call by value unfoldings if required.

tryfindfold is where most of the cleverness of the system resides. It attempts to find a fold between the developing equation and an equation on the equation list. There are several filters that reject obviously undesirable folds e.g. ones that lead to recursions that definitely do not terminate, or ones that lead to recursive patterns different from those being sought (more about this later). For any possible fold that passes all the filters the equation that would result is printed out for the user's inspection. The user can then do one of three things

(i) Reject this fold by typing NOGOOD whereupon the system ignores this possible fold and continues developing the equation.

(ii) Accept this fold but continue by typing CONTINUE, in this case the system makes the fold and carries on developing the equation.

(iii) Accept the fold as a final version of the equation by typing OK. The system makes the fold and exits directly from develop to select the next instantiation if any remain.

At its simplest, folding consists of searching through the equation list to find an equation an instance of whose right hand side occurs within the right hand side of the developing equation. More interesting behaviour occurs when tryfindfold looks for a forced fold, using information collected from a failure to achieve a simple fold to direct the development of the equation so that a fold can be achieved. The simplest form of this occurs when trying to fold the equation with an equation whose right hand side is a tuple. We will illustrate this by showing the optimisation of the fibonacci function, defined thus

$$\begin{aligned} \text{fib}(0) &\Leftarrow 1 \\ \text{fib}(1) &\Leftarrow 1 \\ \text{fib}(n+2) &\Leftarrow \text{fib}(n+1)+\text{fib}(n). \end{aligned}$$

The key to this optimisation lies in defining the auxiliary function

$$g(n) \Leftarrow \text{maketuple}([\text{fib}(n+1), \text{fib}(n)])$$

The system can now discover such definitions for itself as we shall see in Section 6, however, for the moment let us take it as given. The dialogue for the improvement goes as follows.

EXAMPLE 2. *Fibonacci.*

```
EQNLIST.PREXP;

G(N) ⇐ MAKETUPL([FIB(N + 1), FIB(N)])
FIB((N + 1) + 1) ⇐ FIB(N + 1) + FIB(N)
FIB(1) ⇐ 1                                  —original equations
FIB(0) ⇐ 1:
:
:
: START;
TYPE IN INSTANTIATED L.H.S. BASE CASES FIRST.
G(0) END

G(0) ⇐ MAKETUPL([1, 1])

NOW TYPE IN RECURSIVE BRANCHES
FIB((N + 1) + 1), G(N + 1) END                —two cases to try
HOW DOES

FIB((N + 1) + 1) ⇐ U1 + U2 WHEREP
                                   MAKETUPL([U1, U2]) = = G(N)
                                                —fibonacci in terms of g
LOOK TO YOU
OK

HOW DOES

G(N + 1) ⇐ MAKETUPL([FIB((N + FIB(1)) + FIB(1)), FIB(N) + FIB(1)])

LOOK TO YOU
NOGOOD

HOW DOES

G(N + 1) ⇐ MAKETUPL([FIB((N + FIB(0)) + FIB(0)), FIB(N) + FIB(0)])

LOOK TO YOU
NOGOOD

HOW DOES

G(N + 1) ⇐ MAKETUPL([U3 + U4, U3]) WHEREP
                                         MAKETUPL([U3, U4]) = = G(N)
                                                    recursion for g
```

```
LOOK TO YOU
OK

FINISHED.CPU TIME TAKEN 0 MINS 11 SECS
:
:
:
:. PREXP;                               —print out of resulting equations
 FIB(0)⇐1
 FIB(1)⇐1
 FIB((N + 1) + 1)⇐U1 + U2 WHEREP MAKETUPL([U1, U2]) = = G(N)
 G(0)⇐MAKETUPL([1, 1])
 G(N + 1)⇐MAKETUPL([U3 + U4, U3]) WHEREP
                                       MAKETUPL([U3, U4]) = = G(N)
```

Forced folding comes into play during the optimisation of $g(n+1)$, simple unfolding produces the equation

$$g(n+1) \Leftarrow \langle \text{fib}(n+1) + \text{fib}(n), \text{fib}(n) \rangle \qquad Ⓐ$$

and when tryfindfold attempts to fold this with

$$g(n) \Leftarrow \langle \text{fib}(n+1), \text{fib}(n) \rangle \qquad Ⓑ$$

the matching routine fails to find a direct match but notices that all the components necessary to match Ⓑ are present within Ⓐ so it forces the rearrangement of Ⓐ to

$$g(n+1) \Leftarrow \langle u1 + u2, u1 \rangle \textit{ where } \langle u1, u2 \rangle = \langle \text{fib}(n+1), \text{fib}(n) \rangle$$

which folds with Ⓑ giving

$$g(n+1) \Leftarrow \langle u1 + u2, u1 \rangle \textit{ where } \langle u1, u2 \rangle = g(n)$$

as above.

This technique of forced folding, an application of means end analysis, contributes greatly to any sense of purpose the system possesses. Forced folding is also used in the automatic generation of sub-functions, but we will postpone this discussion until Section 5, after synthesis, as it is used in both transformation and synthesis. As mentioned earlier, forced folding was also used to restrict the application of rewrite rules to those that were immediately beneficial in achieving folds, but for the moment this has been dropped in favour of a simpler method.

EXAMPLE 3. *Conversion to iterative and bottom up form. Sum.*

3.1. *Normal recursion to iteration*
The transformation system enables us to convert normal recursive forms to iterative forms that use an accumulator. The user has to supply a definition for the iterative form in terms of the recursive.

```
EQNLIST.PREXP:

SUMIT(I, ACC) = ACC + SUM(I)                 original equations
SUM(N + 1) = (N + 1) + SUM(N)                including definition
SUM(0) = 0:                                  of iterative form
:
: START;                                     sum(N) = Σ_{i=0}^{N} i
TYPE IN INSTANTIATED L.H.S. BASE CASES FIRST.
SUMIT(0, ACC) END
SUMIT(0, ACC) = ACC
NOW TYPE IN RECURSIVE BRANCHES.
SUMIT(I + 1, ACC), SUM(N + 1) END
HOW DOES

SUMIT(I + 1, ACC) = SUMIT(I, (I + 1) + ACC)
LOOK TO YOU
OK

HOW DOES

SUM(N + 1) = SUMIT(N, N + 1)
LOOK TO YOU
OK
FINISHED. CPU TIME TAKEN 0 MINS 3 SECS

:.PREXP;
SUM(0)⇐0
SUM(N + 1)⇐SUMIT(N, N + 1)
SUMIT(I + 1, ACC)⇐SUMIT(I, (I + 1) + ACC)
SUMIT(0, ACC)⇐ACC
:
```

3.2. *Top down recursion to bottom up recursion*
We can also convert the normal 'going down' recursion to a 'going up' form, again

by supplying a definition

```
EQNLIST.PREXP;
SUMUP(I, N) ⇐ SUM(N) − SUM(I)
SUM(N + 1) ⇐ (N + 1) + SUM(N)
SUM(0) ⇐ 0:
:
:
:START;
TYPE IN INSTANTIATED L.H.S. BASE CASES FIRST.
SUMUP(N, N) END

SUMUP(N, N) ⇐ 0

NOW TYPE IN RECURSIVE BRANCHES.
SUMUP(I, N,), SUM(N + 1) END

HOW DOES

SUMUP(I, N) ⇐ (SUM(N) − SUM(I + 1)) + (I + 1)

LOOK TO YOU
CONTINUE

HOW DOES

SUMUP(I, N) ⇐ SUMUP(I + 1, N) + (I + 1)

LOOK TO YOU
OK

HOW DOES

SUM(N + 1) ⇐ (N + 1) + (SUM(N) − SUM(0))

LOOK TO YOU
CONTINUE

HOW DOES

SUM(N + 1) ⇐ (N + 1) + SUMUP(0, N)
LOOK TO YOU
OK
```

```
FINISHED CPU TIME TAKEN 0 MINS 4 SECS
:.PREXP;
SUM(0)⇐0
SUM(N + 1)⇐(N + 1) + SUMUP(0, N)
SUMUP(I, N)⇐SUMUP(I + 1, N) + (I + 1)
SUMUP (N, N)⇐0
:
```

4. Synthesis

As mentioned previously, NPL allows certain set and logic constructs on the right hand side of equations. The concrete syntax of these constructs is

```
setexpression     : : = ⟨:expression : generator* :⟩
existsexpression : : = exists generator* : expression
forallexpression : : = forall generator* : expression
generator         : : = variable in expression (range)
                                    & expression (condition).
```

The NPL interpreter evaluates the list of generators in a left to right fashion so conditions can mention any bound variables that occur to their left. The interpreter is able to run many programs written using these facilities, but some programs written are horrendously inefficient while others are not runnable as they stand.

For example a function begins(a, b) to check whether the list a forms the initial segment of list b can be defined thus begins(a, b)⇐*exists* c in universoflists( ): equallist(a⟨ ⟩c, b) where universoflists( ) can either be uninterpreted or return the set of all possible lists up to a certain size given a set of atoms, the former makes the definition unrunnable, the latter makes it very inefficient. The transformation system is able to convert these sorts of definitions to efficient recursions removing the set or logic constructs.
For example

EXAMPLE 4. *Begins.*

```
EQNLIST.PREXP;
BEGINS(L, M)⇐EXISTS L1 IN UNIV : L⟨ ⟩L1 = = M   (⟨ ⟩ is infix for ap-
                                                  pend)
A : : L⟨ ⟩M⇐A : : (L⟨ ⟩M)
[ ]⟨ ⟩M⇐M
A : : L = = B : : M⇐(A = B) AND (L = = M)
[ ] = = B : : M⇐FALSE
A : : L = = [ ]⇐FALSE
```

```
[ ] = = [ ] ⇐ TRUE
:
:
:.START;
TYPE IN INSTANTIATED L.H.S. BASE CASES FIRST.
BEGINS (NIL, M), BEGINS (A :: L, NIL) END

BEGINS ([ ], M) ⇐ TRUE
BEGINS (A :: L, [ ]) ⇐ FALSE

NOW TYPE IN RECURSIVE BRANCHES.
BEGINS (A :: L, B :: M) END
HOW DOES
BEGINS (A :: L, B :: M) ⇐ (A = B) AND BEGINS (L, M)
LOOK TO YOU
OK
FINISHED.CPU TIME TAKEN 0 mins 4 secs
:
:
:
:
:
:
:.PREXP;
  BEGINS ([ ], M) ⇐ TRUE
  BEGINS (A :: L, [ ]) ⇐ FALSE
  BEGINS (A :: L, B :: M) ⇐ (A = B) AND BEGINS (L, M)
```

The system behaves towards these new constructs exactly as it does towards the more normal expressions shown earlier. However, the concept of unfolding has to be extended as often these constructs obviate the need for recursively defined functions to appear on the right hand side of equations. We have developed a set of reduction rules that are used on these constructs, for example

$$\langle{:}f(x){:}\ x \text{ in } \emptyset \ \&\ P(x){:}\rangle \Leftarrow \emptyset$$

$$\langle{:}f(x){:}\ x \text{ in } x1 + X \ \&\ P(x){:}\rangle \Leftarrow \begin{cases} \langle{:}f(x1){:}\rangle \cup \langle{:}f(x){:}\ x \text{ in } X \ \&\ P(x){:}\rangle & \textit{if } P(x1) \\ \langle{:}f(x){:}\ x \text{ in } X \ \&\ P(x){:}\rangle & \textit{otherwise} \end{cases}$$

$$\textit{exists}\ x \text{ in } \emptyset \ \&\ P(x){:}\ Q(x) \Leftarrow \text{False}.$$

These are applied allwaysofundolding (not reduce) always in a left to right manner.

This task has been greatly simplified by use of a unifying deep structure for these constructs that reduces the number of separate reduction rules needed. To illustrate this let us first rewrite our constructs

$$\langle :f(x)\text{: } x \textit{ in } X \,\&\, P(x):\rangle \equiv \bigcup_{x \textit{ in } X \,\&\, P(x)} \langle :f(x):\rangle \quad \text{(union)}$$

$$\textit{forall } x \textit{ in } X \,\&\, P(x)\text{: } Q(x) \equiv \bigwedge_{x \textit{ in } X \,\&\, P(x)} Q(x) \quad \text{(conjunction)}$$

$$\textit{exists } x \textit{ in } X \,\&\, P(x)\text{: } Q(x) \equiv \bigvee_{x \textit{ in } X \,\&\, P(x)} Q(x) \quad \text{(disjunction)}$$

and we see the underlying similarity. Each construct consists of a predicated generation process (x *in* X & $P(x)$), an expression for each element so generated ($\langle :f(x):\rangle, Q(x)$) and an operation to combine these expressions ($\cup, \wedge, \vee$). Each operator has associated identifiers and zeros forming a monoid. These structures are represented by iterative expressions which have the following abstract syntax.

$$\text{iterative expression} ::= \langle \text{monoid, generator}^*, \text{expression}\rangle$$

$$\text{generator} ::= \langle \text{variable, range, condition}\rangle$$

$$\text{monoid} ::= \langle \text{operator, identity, zero}\rangle$$

The monoids for our various constructs are

$$\text{set} = \langle \text{union}, \emptyset, \perp\rangle$$

$$\text{forall} = \langle \text{and, true, false}\rangle$$

$$\text{exists} = \langle \text{or, false, true}\rangle$$

Thus for example

$$\langle :f(x, y)\text{: } x \textit{ in } X \cup Y \,\&\, P(x), y \textit{ in } Y \,\&\, Q(x, y):\rangle$$

is represented as

$$\langle \text{setmonoid}, [\text{gen1}, \text{gen2}], \langle :f(x, y):\rangle\rangle$$

where

$$\text{gen1} = \langle x, X \cup Y, P(x)\rangle$$

$$\text{gen2} = \langle y, Y, Q(x, y)\rangle$$

Thus our rewrite rules are expressed in terms of these iterative expressions.

Some of the ones available to the system at present are

(i) $\langle$monoid, $\langle bv,\ x1+X$, Cond$\rangle$: : genlist, exp$\rangle$ $\quad (x1+X \equiv \langle{:}x1{:}\rangle \cup X)$
$\Leftarrow\langle$monoid, genlist$\{bv/x1\}$[1], exp$\{bv/x1\}\rangle$
monoid.operatorof $\langle$monoid, genlist, exp$\rangle$ *if* Cond$\{bv/x1\}$
$\langle$monoid, genlist, exp$\rangle$ *otherwise*

For example

forall x in $x1+X$ *&* $P(x)$, *y in Y &* $Q(x, y)$: $R(x, y)$

reduces to

forall y in Y& $Q(x1, y)$: $R(x1, y)$
and
forall x in X & $P(x)$, *y in Y &* $Q(x, y)$: $R(x, y)$ *if* $P(x1)$
forall x in X & $P(x)$ *y in Y &* $Q(x, y)$: $R(x, y)$ *otherwise*

(ii) $\langle$monoid, nil, exp$\rangle$

$\Leftarrow$ exp

In conjunction with (i) this reduces

$\langle{:}x{:}$ *x in* $x1+X$ *&* $P(x){:}\rangle$

to

$\langle{:}x1{:}\rangle \cup \langle{:}x{:}$ *x in X &* $P(x){:}\rangle$ *if* $P(x1)$
$\langle{:}x{:}$ *x in X &* $P(x){:}\rangle$ *otherwise*

(iii) $\langle$monoid, $\langle x, \emptyset$, Cond$\rangle$: : genlist, exp$\rangle$
$\Leftarrow$ monoid.identityof

This reduces

$\langle{:}x{:}$ *x in* $\emptyset$ *&* $P(x){:}\rangle$ to $\emptyset$
forall x in $\emptyset$ *&* $P(x)$: $Q(x)$ to true
and *exists x in* $\emptyset$ *&* $P(x)$: $Q(x)$ to false

(iv) $\langle$monoid, $\langle x, S1 \cup S2$, Cond$\rangle$: : genlist, exp$\rangle$
$\Leftarrow\langle$monoid, $\langle x, S1$, Cond$\rangle$: : genlist, exp$\rangle$
monoid.operatorof $\langle$monoid, $\langle x, S2$, Cond$\rangle$: : genlist, exp$\rangle$

This reduces

exists x in $X1 \cup X2$: $R(x)$

to

exists x in $X1$: $R(x)$ or *exists x in* $X2$: $R(x)$

[1] $\mathscr{C}\{\mathrm{bv}/x1\}$ means $\mathscr{C}$ with all occurrences of bv replaced by $x1$.

(v) ⟨monoid, genlist, $e1$ op $e2$⟩
$\Leftarrow$ ⟨monoid, genlist, $e1$⟩ op ⟨monoid, genlist, $e2$⟩

provided op and the monoid operator distribute, i.e. if the monoid operator is opm we need

$(a \text{ op } b) \text{ opm } (c \text{ op } d) \equiv (a \text{ opm } c) \text{ op } (b \text{ opm } d)$

This reduces

exists x in X: $P(x)$ *or* $Q(x)$

to *exists x in X*: $P(x)$ or *exists x in X*: $Q(x)$

and

forall x in X: $P(x)$ *&* $Q(x)$
to *forall x in X*: $P(x)$ *& forall x in X*: $Q(x)$

and

$$\bigcup_{x \text{ in } X} f(x) \cup g(x)$$

to

$$\bigcup_{x \text{ in } X} f(x) \cup \bigcup_{x \text{ in } X} g(x)$$

(vi) ⟨monoid, genlist, $e1$ op $e2$⟩
$\Leftarrow$ $e1$ op ⟨monoid, genlist, $e2$⟩

provided that $e1$ contains no variables bound in genlist. This reduces

forall x in X: $P(y)$ or $Q(x, y)$
to $P(y)$ or *forall x in X*: $Q(x, y)$

5. Generalisation and Sub-function Synthesis

In all the examples shown so far, the recursions introduced have been calls to previously defined functions. An important attribute of the system is its ability to invent new functions for itself, through various forms of generalisation. Switches enable the user to switch the various generalisation facilities on and off.

5.1. Simple generalisation

The simplest form of generalisation used is when a new function is defined to compute some part of the developing equation. For example if we define the

Cartesian product of two sets thus

$$\mathrm{Cart}(X, Y) \Leftarrow \langle : \langle x\ y \rangle : x \mathit{\ in\ } X, y \mathit{\ in\ } Y : \rangle$$

and proceed with a synthesis we get

$$\mathrm{Cart}(\emptyset, Y) \Leftarrow \emptyset$$

and

$$\mathrm{Cart}(x1 + X, Y) \Leftarrow \langle : \langle x1\ y \rangle : y \mathit{\ in\ } Y : \rangle \cup \langle : \langle x\ y \rangle : x \mathit{\ in\ } X, y \mathit{\ in\ } Y : \rangle$$

Using reduction rules (iii) and (i).

The system spots that $\langle : \langle x1\ y \rangle : y \mathit{\ in\ } Y : \rangle$ is going to need computing and asks the user if it can go ahead and define a new function by

$$\mathrm{new}f1(u1, u2, u3, u4) \Leftarrow \langle : \langle u1\ y \rangle : y \mathit{\ in\ } u3 : \rangle \cup \langle : \langle x\ y \rangle : x \mathit{\ in\ } u2, y \mathit{\ in\ } u4 : \rangle$$

If this is accepted by the user, the system postpones consideration of Cart and turns its attention to new$f$1. It asks the user to provide instantiations as before, base cases and main recursions (this time the form of input is slightly different as the system already knows what function is to be so instantiated) and proceeds to evaluate these as before. If it is successful in finding acceptable forms for all the instantiations of the new function it adds these equations to the equation list and returns to working on Cart. If it succeeds with Cart the equations for new$f$1 are retained. If no acceptable forms can be found for new$f$1 the system abandons this generalisation and returns to working on Cart. In this example the system succeeds and the final equations produced are

$$\begin{aligned}
&\mathrm{cart}(\emptyset, Y) \Leftarrow \emptyset\\
&\mathrm{cart}(x1 + X, Y) \Leftarrow \mathrm{new}f1(x1, X, Y, Y)\\
&\mathrm{new}f1(u1, u2, \emptyset, u4) \Leftarrow \mathrm{cart}(u2, u4)\\
&\mathrm{new}f1(u1, u2, y1 + u3, u4) \Leftarrow \langle : \langle u1\ y1 \rangle : \rangle \cup \mathrm{new}f1(u1, u2, u3, u4)
\end{aligned}$$

This system of recursion is rather idiosyncratic. A more orthodox pattern is

$$\begin{aligned}
&\mathrm{cart}(\emptyset, Y) \Leftarrow \emptyset\\
&\mathrm{cart}(x1 + X, Y) \Leftarrow \mathrm{new}f1(x1, Y) \cup \mathrm{cart}(X, Y)\\
&\mathrm{new}f1(u1, \emptyset) \Leftarrow \emptyset\\
&\mathrm{new}f1(u1, y1 + u2) \Leftarrow \langle : \langle u1\ y1 \rangle : \rangle \cup \mathrm{new}f1(u1, u2)
\end{aligned}$$

which can be achieved by defining

$$\mathrm{new}f1(u1, u2) \Leftarrow \langle : \langle u1\ y \rangle : y \mathit{\ in\ } u2 : \rangle$$

We can produce this pattern also and in Section 5.3 we show how it is achieved. Martin Feather wrote much of the code to enable the system to achieve this form of generalisation.

EXAMPLE 5. *Multiple Cartesian product.* In this example we convert the definition of the Cartesian product of three sets to a recursion involving three loops.

In this example the switch NOSETEXPRS is set. This restricts the use of the set and logic constructs to the specification stages and does not allow them in the final versions, cf. the specification and target languages used by Manna and Waldinger [13]. This enables the system to suggest the 'right' version every time.

```
EQNLIST.PREXP;
CART3(X, Y, Z)⇐⟨:MAKETUPL([X1, X2, X3]): X1 IN X X2 IN Y
                    X3 IN Z:⟩:
:
:START;
TYPE IN INSTANTIATED L.H.S. BASE CASES FIRST.
CART3(NILSET, Y, Z) END

CART3(⟨: :⟩, Y, Z)⇐⟨: :⟩

NOW TYPE IN RECURSIVE BRANCHES.
CART3(CONSSET(X4, X), Y, Z) END

COULD GENERALISE FOLLOWING EXPRESSION:
HOW DOES

⟨: MAKETUPL([X4, X2, X3]): X2 IN Y X3 IN Z:⟩+⟨: MAKETUPL([X1,
  X2, X3]): X1 IN X X2 IN Y X3 IN Z:⟩

LOOK TO YOU
OK

AN ATTEMPTING TO SYNTHESISE A FUNCTION NEWF1 DEFINED
IMPLICITLY BY THIS EQUATION


NEWF1 (U6, U5, U4, U3, U2, U1)⇐⟨:MAKETUPL([U3, X2, X3]): X2
IN U1 X3 IN U2:⟩+⟨:MAKETUPL([X1, X2, X3]): X1 IN U4 X2 IN U5
X3 IN U6:⟩


HOW DOES


NEWF1 (U6, U5, U4, U3, U2, U1)⇐⟨:MAKETUPL([U3, X2, X3]): X2
IN U1 X3 IN U2:⟩+⟨:MAKETUPL([X1, X2, X3]): X1 IN U4 X2 IN U5
X3 IN U6:⟩
```

LOOK TO YOU
OK

(There is avoidable redundancy here, having accepted the generalisation the new function will always be accepted.)

WOULD YOU TYPE IN SUITABLE INSTANTIATION PAIRS
BASE CASES FIRST.
:END —'base' case involves a recursion
NOW THE RECURSIONS

:*U*1, NILSET END *U*1, CONSSET(*X*5, *U*1) END END

HOW DOES

NEW*F*1(*U*6, *U*5, *U*4, *U*3, *U*2, ⟨: :⟩)⇐CART3(*U*4, *U*5, *U*6)

LOOK TO YOU
OK

COULD GENERALISE FOLLOWING EXPRESSION:
HOW DOES

(⟨: MAKETUPL([*U*3, *X*5, *X*3]): *X*3 IN *U*2:⟩+⟨: MAKETUPL([*U*3, *X*2, *X*3]): *X*2 IN *U*1 *X*3 IN *U*2:⟩)+⟨: MAKETUPL([*X*1, *X*2, *X*3]): *X*1 IN *U*4 *X*2 IN *U*5 *X*3 IN *U*6:⟩

LOOK TO YOU
OK

AM ATTEMPTING TO SYNTHESISE A FUNCTION NEW*F*2 DEFINED IMPLICITLY BY THIS EQUATION

NEW*F*2(*U*15, *U*14, *U*13, *U*12, *U*11, *U*10, *U*9, *U*8, *U*7)⇐(⟨: MAKETUPL ([*U*8, *U*9, *X*3]): *X*3 IN *U*7:⟩+⟨: MAKETUPL([*U*12, *X*2, *X*3]): *X*2 IN *U*10 *X*3 IN *U*11:⟩)+⟨: MAKETUPL([*X*1, *X*2, *X*3]): *X*1 IN *U*13 *X*2 IN *U*14 *X*3 IN *U*15:⟩

HOW DOES

NEW*F*2(*U*15, *U*14, *U*13, *U*12, *U*11, *U*10, *U*9, *U*8, *U*7)⇐(⟨: MAKETUPL ([*U*8, *U*9, *X*3]): *X*3 IN *U*7:⟩+⟨: MAKETUPL([*U*12, *X*2, *X*3]): *X*2 IN *U*10 *X*3 IN *U*11:⟩)+⟨: MAKETUPL([*X*1, *X*2, *X*3]): *X*1 IN *U*13 *X*2 IN *U*14 *X*3 IN *U*15:⟩

```
LOOK TO YOU
OK

WOULD YOU TYPE IN SUITABLE INSTANTIATION PAIRS. BASE
CASES FIRST.

: END
NOW THE RECURSIONS

: U7, NILSET END U7, CONSSET (X6, U7) END END

HOW DOES

NEWF2(U15, U14, U13, U12, U11, U10, U9, U8, ⟨::⟩) = NEWF1(U15,
U14, U13, U12, U11, U10)

LOOK TO YOU
OK

HOW DOES

NEWF2(U15, U14, U13, U12, U11, U10, U9, U8, CONSSET(X6, U7))
⇐⟨: MAKETUPL([U8, U9, X6]): :⟩ + NEWF2(U15, U14, U13, U12,
U11, U10, U9, U8, U7)

LOOK TO YOU
OK                                          —system climbs back up

HOW DOES

NEWF1(U6, U5, U4, U3, U2, CONSSET(X5, U1))⇐NEWF2(U6, U5,
                                    U4, U3, U2, U1, X5, U3, U2)

LOOK TO YOU
OK

HOW DOES

CART3(CONSSET(X4, X), Y, Z)⇐NEWF1(Z, Y, X, X4, Z, Y)

LOOK TO YOU
OK

FINISHED.CPU TIME TAKEN 0 MINS 33 SECS
```

```
:.PREXP;
NEWF1(U6, U5, U4, U3, U2, ⟨: :⟩)⇐CART3(U4, U5, U6)
NEWF1(U6, U5, U4, U3, U2, CONSSET(X5, U1))⇐NEWF2(U6, U5,
                                        U4, U3, U2, U1, X5, U3, U2)
NEWF2(U15, U14, U13, U12, U11, U10, U9, U8, CONSSET(X6,
U7))⇐⟨: MAKETUPL([U8, U9, X6]): :⟩+NEWF2(U15, U14, U13, U12,
                                          U11, U10, U9, U8, U7)
NEWF2(U15, U14, U13, U12, U11, U10, U9, U8, ⟨: :⟩)⇐NEWF1(U15,
                                        U14, U13, U12, U11, U10)
CART3(CONSSET(X4, X), Y, Z)⇐NEWF1(Z, Y, X, X4, Z, Y)
CART3(⟨: :⟩, Y, Z)⇐⟨: :⟩
:
```

Example 6. *Double Sum.* This example illustrates generalisation at work in a transformation example. Every element of a two level list is doubled by the function double and the resulting two level list summed by the function sum. Both double and sum use subsidary functions db and sm to iterate along the component lists. This loop structure is collapsed to produce a more efficient version that traverses the list only once using an invented subsidary function.

```
EQNLIST.PREXP;
G(XLL)⇐SUM(DOUBLE(XLL))                    e.g. G([[1, 2], [3]]) = 12
DOUBLE(XL: :XLL)⇐DB(XL): :DOUBLE(XLL)
DOUBLE([ ])⇐[ ]
DB(X: :XL)⇐2*X: :DB(XL)
DB([ ])⇐[ ]
SUM(XL: :XLL)⇐SM(XL)+SUM(XLL)
SUM([ ])⇐0
SM(X: :XL)⇐X+SM(XL)
SM([ ])⇐0: START;
TYPE IN INSTANTIATED L.H.S. BASE CASES FIRST.
G(NIL) END
G([ ])⇐0
NOW TYPE IN RECURSIVE BRANCHES.
G(XL: :XLL) END
COULD GENERALISE FOLLOWING EXPRESSION:
HOW DOES

SUM(DOUBLE(XL: :XLL))

LOOK TO YOU
NOGOOD

HOW DOES
```

```
G(XL::XLL)⇐SUM(DOUBLE(XL::XLL))

LOOK TO YOU
NOGOOD

COULD GENERALISE FOLLOWING EXPRESSION:
HOW DOES

SUM(DB(XL)::DOUBLE(XLL))

LOOK TO YOU
NOGOOD

COULD GENERALISE FOLLOWING EXPRESSION:
HOW DOES

SM(DB(XL))+SUM(DOUBLE(XLL))

LOOK TO YOU
OK

AM ATTEMPTING TO SYNTHESISE A FUNCTION      —again redundant
NEWF1 DEFINED IMPLICITLY BY THIS EQUATION

NEWF1(U2, U1)⇐SM(DB(U1))+SUM(DOUBLE(U2))

HOW DOES

NEWF1(U2, U1)⇐SM(DB(U1))+SUM(DOUBLE(U2))

LOOK TO YOU
OK

WOULD YOU TYPE IN SUITABLE INSTANTIATION PAIRS.
BASE CASES FIRST.
: END                              (newf2(u2, nil) involves a recursion)
NOW THE RECURSIONS

U1, NIL END U1, X::U1 END END

HOW DOES

NEWF1(U2, [ ])⇐SM(DB([ ]))+G(U2)
```

LOOK TO YOU
NOGOOD

HOW DOES

NEW$F$1($U$2, [ ])⇐SM(DB(DOUBLE([ ])))+SUM(DOUBLE($U$2))

LOOK TO YOU
NOGOOD

HOW DOES

NEW$F$1($U$2, [ ])⇐SM(DB(DB([ ])))+SUM(DOUBLE($U$2)) (nearly there!)

LOOK TO YOU
NOGOOD

HOW DOES

NEW$F$1($U$2, [ ])⇐SUM([ ]::DOUBLE($U$2))

LOOK TO YOU
NOGOOD

COULD GENERALISE FOLLOWING EXPRESSION:

HOW DOES

SM([ ])+SUM(DOUBLE($U$2))

LOOK TO YOU
NOGOOD

HOW DOES

NEW$F$1($U$2, [ ])⇐G($U$2)

LOOK TO YOU
OK

HOW DOES main recursion for new$f$1

NEW$F$1($U$2, X::$U$1)⇐SM(DB(X::$U$1))+G($U$2)

```
LOOK TO YOU
NOGOOD

HOW DOES

NEW(U2, X :: U1) ⇐ SUM(DB(X :: U1) :: DOUBLE(U2))

LOOK TO YOU
NOGOOD

COULD GENERALISE FOLLOWING EXPRESSION:

HOW DOES

SM(DB(X :: U1)) + SUM(DOUBLE(U2))

LOOK TO YOU
NOGOOD

HOW DOES

NEWF1(U2, X :: U1) ⇐ SM(DB(X :: U1)) + SUM(DOUBLE(U2))

LOOK TO YOU
NOGOOD

HOW DOES

NEWF1(U2, X :: U1) ⇐ SUM((2 * X :: DB(U1)) :: DOUBLE(U2))

LOOK TO YOU
NOGOOD

COULD GENERALISE FOLLOWING EXPRESSION:

HOW DOES

SM(2 * X :: DB(U1)) + SUM(DOUBLE(U2))

LOOK TO YOU
NOGOOD

HOW DOES
```

```
NEWF1(U2, X :: U1)⇐(2 * X)+NEWF1(U2, U1)

LOOK TO YOU
OK

HOW DOES                                        back to top level

G(XL :: XLL)⇐NEWF1(XLL, XL)

LOOK TO YOU
OK

FINISHED.CPU TIME TAKEN 0 MINS 23 SECS

.PREXP;
                                                (resulting equations)
NEWF1(U2, X :: U1)⇐(2 * X)+NEWF1(U2, U1)
NEWF1(U2, [ ])⇐G(U2)
SM([ ])⇐0
SM(X :: XL)⇐X + SM(XL)
SUM([ ])⇐0
SUM(XL :: XLL)⇐SM(XL)+SUM(XLL)
DB([ ])⇐[ ]
DB(X :: XL)⇐2 * X :: DB(XL)
DOUBLE([ ])⇐[ ]
DOUBLE(XL :: XLL)⇐DB(XL) :: DOUBLE(XLL)
G(XL :: XLL)⇐NEWF1(XLL, XL)
G([ ])⇐0:
:
```

5.2. Forced folding leading to sub-function definition

A more sophisticated method of introducing new functions arises from extending the notion of forced folding. We will illustrate this technique using the following example that employs binary trees.
Given

$$\begin{aligned}&\text{flatten}(\text{atom}(a))\Leftarrow[a]\\&\text{flatten}(\text{consbt}(X, Y))\Leftarrow\text{flatten}(X)\langle\rangle\text{flatten}(Y)\\&\text{nil}\langle\rangle Y\Leftarrow Y\\&(x::X)\langle\rangle Y\Leftarrow x::(X\langle\rangle Y)\\&\text{length}(\text{nil})\Leftarrow 0\\&\text{length}(x::X)\Leftarrow 1+\text{length}(X)\end{aligned}$$

can we optimise

$$g(X)\Leftarrow\text{length}(\text{flatten}(X))$$

The system proceeds as follows; straight evaluation gives

$$g(\text{atom}(a)) \Leftarrow 1$$

but trying to evaluate $g(\text{consbt}(X, Y))$ it gets as far as

$$g(\text{consbt}(X, Y)) \Leftarrow \text{length}(\text{flatten}(X)\langle\rangle\text{flatten}(Y)) \quad Ⓐ$$

and no further unfolding or immediate folds are possible. However, on attempting to fold the above with $g(X) \Leftarrow \text{length}(\text{flatten}(X))$ the system automatically synthesises an equation defining a new subsidiary function that can be used to rewrite the above into a form that will fold. In this case the equation produced is

$$\text{new}f1(\text{length}(u1), \text{length}(u2)) \Leftarrow \text{length}(u1\langle\rangle u2) \quad Ⓑ$$

We call these new style equations with complicated left hand sides (which cannot be run in NPL) *implicit* equations. Using Ⓑ the system can rewrite Ⓐ as

$$g(\text{consbt}(X, Y)) \Leftarrow \text{new}f1(\text{length}(\text{flatten}(X)), \text{length}(\text{flatten}(Y)))$$

which will fold immediately giving

$$g(\text{consbt}(X, Y)) \Leftarrow \text{new}f1(g(X), g(Y))$$

But first the system must synthesise proper recursions for new$f$1. It firsts asks the user to accept or reject this proposed new function and if it is accepted asks for suitable instantiations. In this example the system performs the following deductions.

$$\begin{array}{rll}
\text{new}f1(\text{length}(\text{nil}), \text{length}(u2)) & \Leftarrow \text{length}(\text{nil}\langle\rangle u2) & \\
& \Leftarrow \text{length}(u2) & \text{Unfolding the right hand side} \\
\text{new}f1(0, \text{length}(u2)) & \Leftarrow \text{length}(u2) & \text{Unfolding the left hand side} \\
\text{new}f1(0, u3) & \Leftarrow u3 & \text{Generalising}
\end{array}$$

and

$$\begin{array}{rl}
\text{new}f1(\text{length}(x::u1), \text{length}(u2)) & \Leftarrow \text{length}((x::u1)\langle\rangle u2) \\
& \Leftarrow 1 + \text{length}(u1\langle\rangle u2) \\
& \quad \text{Unfolding the right hand side} \\
& \Leftarrow 1 + \text{new}f1(\text{length}(u1), \text{length}(u2)) \\
& \quad \text{Folding with Ⓑ} \\
\text{new}f1(1 + \text{length}(u1), \text{length}(u2)) & \Leftarrow 1 + \text{new}f1(\text{length}(u1), \text{length}(u2)) \\
& \quad \text{Unfolding the left hand side} \\
\text{new}f1(1 + u3, u4) & \Leftarrow 1 + \text{new}f1(u3, u4) \\
& \quad \text{Generalising.}
\end{array}$$

Thus the system has synthesised a recursion for +. This could be tidied up by applying the schemata reduction rule

$$\begin{aligned}&f(b,c)\Leftarrow d\\&f(a1 \text{ op } a2, b)\Leftarrow a1 \text{ op } f(a2,b)\\&\text{reduces to}\\&f(a,b)\Leftarrow a \text{ op } b, \text{ provided } d=b \text{ op } c \text{ and op is associative.}\end{aligned}$$

Thus in the above $u3=0+u3$ and $+$ is associative so new$f1(g(X), g(Y))$ is $g(X)+g(Y)$.

Discovery of implicit equations. How does the system come up with the implicit equation of Ⓑ? The mechanism is rather complex but basically relies on mismatching and forced folding. To view the process in a little more detail let us look more closely at the example above. The implicit equation Ⓑ arose from attempting to fold Ⓐ with $g(X)\Leftarrow$ length(flatten(X)) in particular in trying to match length(flatten(X)) with length(flatten(X)⟨ ⟩flatten(Y)). The matching routine proceeds top down and succeeds at the first recursion matching length with length. At the next recursion it attempts to match flatten and gets instead ⟨ ⟩, however, it does not give up but looks if the remaining expression it is trying to match, flatten(X), occurs within the rest of the other expression, which it does, twice. These remaining instances of the sought for sub-expression are then replaced by new variables and the implicit equation constructed to 'move' these variables to the correct position to allow the match to succeed and achieve the fold. Any remaining sub-expressions are also generalised out to get the implicit equation in its simplest form.

Alternative forms. Many solutions involving new sub-functions require the form

$$\begin{aligned}&g(x)\Leftarrow g(\text{new}f1(x))\\&\text{new}f1(x)\Leftarrow \text{new}f1(k(x))\end{aligned}$$

Methods have been developed to achieve this form identical to the one given above except that in this case the implicit equation is of the form

$$\mathscr{C}1(\text{new}f1(u))\Leftarrow \mathscr{C}2(u)$$

This method has not been implemented yet, although it is actually computationally simpler than the previous method, so we will not discuss it here. All these patterns can also be achieved using pattern directed generalisation which we discuss in Section 5.3.

EXAMPLE 8. *Reverse flatten.* In this example the frontier of a binary tree is produced and then reversed. This is first transformed to a version that walks over the tree in reverse order and joins the resulting frontiers together. This new version involves an invented subsidary function which is simply append.

Given that we can recognise this we can go on to convert this to a semi-iterative version, see Burstall and Darlington [4, Section 7], which accumulates the result an element at a time while walking over the tree.

```
EQNLIST.PREXP;
G2(X)⇐REVERSE(FLAT(X))
REVERSE(X1::XL)⇐REVERSE(XL)⟨⟩[X1]
REVERSE([])⇐[]
A::L⟨⟩M⇐A::(L⟨⟩M)
[]⟨⟩M⇐M
FLAT(CONSBT(X, Y))⇐FLAT(X)⟨⟩FLAT(Y)
FLAT(ATM(X1))⇐[X1]:
:
:START;
TYPE IN INSTANTIATED L.H.S. BASE CASES FIRST.
G2(ATM(X1)) END

G2(ATM(X1))⇐X1::[]

NOW TYPE IN RECURSIVE BRANCHES.
G2(CONSBT(X, Y)) END

AM ATTEMPTING TO SYNTHESISE A FUNCTION NEWF1 DEFINED
IMPLICITLY BY THIS EQUATION

NEWF1(REVERSE(U2), REVERSE(U1))⇐REVERSE(U1⟨⟩U2)

HOW DOES

NEWF1(REVERSE(U2), REVERSE(U1))⇐REVERSE(U1⟨⟩U2)

LOOK TO YOU
OK

WOULD YOU TYPE IN SUITABLE INSTANTIATION PAIRS.
BASE CASES FIRST.
:U1, NIL END U1, X1::NIL END END

NEWF1(U3, [])⇐U3

NEWF1(U4, X1::[])⇐U4⟨⟩X1::[]

NOW THE RECURSIONS
```

```
:U1, X1::U1 END END

AM ATTEMPTING TO SYNTHESISE A FUNCTION NEWF2 DEFINED
IMPLICITLY BY THIS EQUATION

NEWF2(U7, REVERSE(U6))⇐REVERSE(U7::U6)

HOW DOES

NEWF2(U7, REVERSE(U6))⇐REVERSE(U7::U6)

LOOK TO YOU
NOGOOD

HOW DOES

NEWF1(REVERSE(U2), REVERSE(X1::U1))⇐NEWF1(REVERSE(U2),
                                       REVERSE(U1))⟨⟩X1::[]

LOOK TO YOU
OK

HOW DOES

G2(CONSBT(X, Y))⇐NEWF1(G2(Y), G2(X))

LOOK TO YOU
OK

FINISHED.CPU TIME TAKEN 0 MINS 11 SECS

.PREXP;
NEWF1(U3, [])⇐U3
NEWF1(U4, X1::[])⇐U4⟨⟩X1::[]
NEWF1(U8, U9⟨⟩U10)⇐NEWF1(U8, U9)⟨⟩U10
REVERSE(X1::XL)⇐REVERSE(XL)⟨⟩[X1]
REVERSE([])⇐[]
A::L⟨⟩M⇐A::(L⟨⟩M)
[]⟨⟩M⇐M
FLAT(CONSBT(X, Y))⇐FLAT(X)⟨⟩FLAT(Y)
FLAT(ATM(X1))⇐[X1]
G2(CONSBT(X, Y))⇐NEWF1(G2(Y), G2(X))
G2(ATM(X1))⇐X1::[]
```

If we recognise that new$f1$ is ⟨⟩ and give an extra definition for git we can convert this further.

```
EQNLIST.PREXP;
G2(CONSBT(X, Y))⇐G2(Y)⟨⟩G2(X)
G2(ATM(X1))⇐[X1]
GIT(X, ACC)⇐G2(X)⟨⟩ACC
A::L⟨⟩M⇐A::(L⟨⟩M)
[]⟨⟩M⇐M:
:
START;
TYPE IN INSTANTIATED L.H.S. BASE CASES FIRST.
: GIT(ATM(X1), ACC) END

GIT(ATM(X1), ACC)⇐X1::ACC

NOW TYPE IN RECURSIVE BRANCHES.
GIT(CONSBT(X, Y), ACC) END

HOW DOES

GIT(CONSBT(X, Y), ACC)⇐G2(Y)⟨⟩GIT(X, ACC)

LOOK TO YOU
CONTINUE

HOW DOES

GIT(CONSBT(X, Y), ACC)⇐GIT(Y, GIT(X, ACC))

LOOK TO YOU
OK

FINISHED.CPU TIME TAKEN 0 MINS 9 SECS

PREXP;
G2(CONSBT(X, Y))⇐G2(Y)⟨⟩G2(X)   (G2(X) is GIT(X, NIL))
G2(ATM(X1))⇐[X1]

A::L⟨⟩M⇐A::(L⟨⟩M)
[]⟨⟩M⇐M
GIT(CONSBT(X, Y), ACC)⇐GIT(Y, GIT(X, ACC))
GIT(ATM(X1), ACC)⇐X1::ACC
```

EXAMPLE 9. *Descending.* Ascending checks whether a list of numbers is in ascending order. We define descending as ascending of the reverse list and convert this to a simple recursion.

```
EQNLIST.PREXP;
DESCENDI(X) ⇐ ASCENDIN(REVERSE(X))
ASCENDIN(X1::(X2::X)) ⇐ X1 < X2 AND ASCENDIN(X2::X)
ASCENDIN(X1) ⇐ TRUE
ASCENDIN([ ]) ⇐ TRUE
A::L⟨ ⟩M ⇐ A::(L⟨ ⟩M)
[ ]⟨ ⟩M ⇐ M
REVERSE(X1::XL) ⇐ REVERSE(XL)⟨ ⟩[X1]
REVERSE([ ]) ⇐ [ ]:

:START;
TYPE IN INSTANTIATED L.H.S. BASE CASES FIRST.
DESCENDING(NIL), DESCENDING(X1::NIL) END

DESCENDI([ ]) ⇐ TRUE
DESCENDI([X1]) ⇐ TRUE

NOW TYPE IN RECURSIVE BRANCHES.
DESCENDING(X3::(X4::X)) END

HOW DOES

DESCENDI(X3::(X4::X)) ⇐ ASCENDIN(REVERSE(([ ]⟨ ⟩X3)::
                        (([ ]⟨ ⟩X4)::([ ]⟨ ⟩X))))

LOOK TO YOU
NOGOOD

HOW DOES

DESCENDI(X3::(X4::X)) ⇐ ASCENDIN(REVERSE(([ ]⟨ ⟩X4)::([ ]⟨ ⟩X))
                        ⟨ ⟩([ ]⟨ ⟩X3)::([ ]⟨ ⟩[ ]))

LOOK TO YOU
NOGOOD

HOW DOES

DESCENDI(X3::(X4::X)) ⇐ ASCENDIN(REVERSE(X3::(X4::X)))

LOOK TO YOU
NOGOOD
```

```
HOW DOES

DESCENDI(X3::(X4::X))⇐ASCENDIN(REVERSE(X4::X)⟨ ⟩X3::
                                REVERSE([ ]))


LOOK TO YOU
NOGOOD

AM ATTEMPTING TO SYNTHESISE A FUNCTION NEWF1 DEFINED
IMPLICITLY BY THIS EQUATION

NEWF1(U3, U2, ASCENDIN(U1))⇐ASCENDIN((U1⟨ ⟩U2)⟨ ⟩U3)

HOW DOES

NEWF1(U3, U2, ASCENDIN(U1))⇐ASCENDIN((U1⟨ ⟩U2)⟨ ⟩U3)

LOOK TO YOU
OK

WOULD YOU TYPE IN SUITABLE INSTANTIATION PAIRS.
BASE CASES FIRST.

:U1, NIL END END

NEWF1(U3, U2, TRUE)⇐ASCENDIN(U2⟨ ⟩U3)

NOW THE RECURSIONS

U1, X3::(X4::U1) END END

HOW DOES

NEWF1(U3, U2, ASCENDIN(X3::(X4::U1)))
    ⇐ASCENDIN(((([ ]⟨ ⟩X3)::(([ ]⟨ ⟩X4)::([ ]⟨ ⟩U1))⟨ ⟩([ ]⟨ ⟩U2))⟨ ⟩([ ]⟨ ⟩U3))

LOOK TO YOU
NOGOOD

HOW DOES

NEWF1(U3, U2, ASCENDIN(X3::(X4::U1)))
  ⇐ASCENDIN((X3::(X4::U1)⟨ ⟩U2)⟨ ⟩U3)
```

```
LOOK TO YOU
NOGOOD

HOW DOES

NEWF1(U3, U2, ASCENDIN(X3::(X4::U1)))
  ⇐ASCENDIN(X3::(X4::((U1⟨ ⟩U2)⟨ ⟩U3)))

LOOK TO YOU
NOGOOD

HOW DOES

NEWF1(U3, U2, ASCENDIN(X3::(X4::U1)))
  ⇐X3<X4 AND ASCENDIN(X4::(U1⟨ ⟩U2)⟨ ⟩U3)

LOOK TO YOU
CONTINUE

HOW DOES

NEWF1(U3, U2, ASCENDIN(X3::(X4::U1)))
  ⇐([ ]⟨ ⟩X3)<([ ]⟨ ⟩X4) AND ASCENDIN((([ ]⟨ ⟩X4)::(([ ]⟨ ⟩U1)
    ⟨ ⟩([ ]⟨ ⟩U2))⟨ ⟩([ ]⟨ ⟩U3))

LOOK TO YOU
NOGOOD

HOW DOES

NEWF1(U3, U2, ASCENDIN(X3::(X4::U1)))
  ⇐X3<X4 AND ASCENDIN((X4::U1⟨ ⟩U2)⟨ ⟩U3)

LOOK TO YOU
CONTINUE

HOW DOES

NEWF1(U3, U2, ASCENDIN(X3::(X4::U1)))
  ⇐([ ]⟨ ⟩X3)<([ ]⟨ ⟩X4 AND ASCENDIN((([ ]⟨ ⟩X4)::([ ]⟨ ⟩U1)
    ⟨ ⟩([ ]⟨ ⟩U2))⟨ ⟩([ ]⟨ ⟩U3))

LOOK TO YOU
NOGOOD
```

```
HOW DOES

NEWF1(U3, U2, ASCENDIN(X3::(X4::U1)))
  ⇐X3<X4 AND NEWF1(U3, U2, ASCENDIN(X4::U1))
                                    Cleaned up will be recursion for and!

LOOK TO YOU
OK

HOW DOES

DESCENDI(X3::(X4::X))⇐NEWF1(X3::[ ], X4::[ ], DESCENDI(X))

LOOK TO YOU
OK

FINISHED.CPU TIME TAKEN 1 MINS 39 SECS

.PREXP;
NEWF1(U3, U2, TRUE)⇐ASCENDIN(U2⟨⟩U3)
NEWF1(U3, U2, U13 AND U14)⇐U13 AND NEWF1(U3, U2, U14)
                        (newf1(x, y, t)⇐y<x and t)

ASCENDIN(X1::(X2::X))⇐X1<X2 AND ASCENDIN(X2::X)
ASCENDIN([X1])⇐TRUE
ASCENDIN([ ])⇐TRUE
A::L⟨⟩M⇐A::(L⟨⟩M)
[ ]⟨⟩M⇐M
REVERSE(X1::XL)⇐REVERSE(XL)⟨⟩[X1]
REVERSE([ ])⇐[ ]
DESCENDI(X3::(X4::X))⇐NEWF1(X3::[ ], X4::[ ], DESCENDI(X))
DESCENDI([ ])⇐TRUE
DESCENDI([X1])⇐TRUE
:
```

5.2.1. *Synthesis of inverses*

The ability to synthesise functions defined by implicit equations enables the system to produce recursions for functions defined by their properties. In particular, it can synthesise the inverse of given functions. For example, given a function, double, that doubles every element of a list

$$\text{double}(\text{nil}) \Leftarrow \text{nil}$$
$$\text{double}(x::X) \Leftarrow 2 * x::\text{double}(X)$$

we can define the inverse of double, doubleinv,

$$\text{doubleinv}(\text{double}(X)) = X \quad Ⓐ$$

and a recursion for doubleinv can be synthesised thus

$\text{doubleinv}(\text{double}(\text{nil})) \Leftarrow \text{nil}$

$\text{doubleinv}(\text{nil}) \Leftarrow \text{nil}$ Unfolding right hand sides

$\text{doubleinv}(\text{double}(x::X)) = x::X$

$\text{doubleinv}(2 * x::\text{double}(X) = x::\text{doubleinv}(\text{double}(X))$

Unfolding the left hand side, folding the right hand side.

$\text{doubleinv}(2 * x::u1) = x::\text{doubleinv}(u1)$

Generalising

A function, tryfindinverse, is provided for the user. This takes an equation such as Ⓐ and then uses the same mechanism that is used to derive recursions from the implicit equations as in Section 5.2. This ability to synthesise inverses enables us to mechanise more of the data representation technique outlined in Burstall and Darlington [4, Section 8 and the second example following shows the 'twistree' example given there].

EXAMPLE 10. *Inverse of reverse.*

```
EQNLIST.PREXP;
REVERSE(X1:X)⇐APPEND(REVERSE(X), [X1])
                                          equations for reverse
REVERSE([ ])⇐[ ]
APPEND([ ], Y)⇐Y
APPEND(X1::X, Y)⇐X1::APPEND(X, Y):
```

(Needs fact on reduction rule list, that reverse(append(X, Y)) ⇐ append(reverse(Y), reverse(X)))

```
EQ1.PREXP;
REVINV(REVERSE(X))⇐X:                     —definition of inverse of reverse

:
EQ1.TRYFINDINVERSE;                        —function provided for user

AM ATTEMPTING TO SYNTHESISE A             —redundant print out (uses
FUNCTION REVINV DEFINED                     same part of system as
IMPLICITLY BY THIS EQUATION                 used for subsidary functions)

REVINV(REVERSE(X))⇐X
```

```
HOW DOES

REVINV(REVERSE(X)) ⇐ X

LOOK TO YOU
OK

WOULD YOU TYPE IN SUITABLE INSTANTIATION PAIRS.
BASE CASES FIRST.

:X, NIL END END

REVINV([ ]) ⇐ [ ]

NOW THE RECURSIONS

X, APPEND(X, [X1]) END END

HOW DOES

REVINV(REVERSE(APPEND(X, [X1])))            printed out in
  ⇐ APPEND(REVINV(REVERSE(X)),              uncleaned up form
     REVINV(REVERSE(X1)) : :                to save (compu-
     REVINV(REVERSE([ ])))                  tation) time

LOOK TO YOU
OK
:
:
:.PREXP;

REVINV(X1 : : U0) ⇐ APPE      REVINV(U0), X1 : : [ ])   —cleaned up form

REVINV([ ]) ⇐ [ ]
```

Example 11. *Twistree* (see [4, Section 8]).

```
EQNLIST.PREXP;
REP(CONSP(A, CONSP(B, C)))                  equations for
  ⇐ CONSTREE(A, REP(B), REP(C))             twist and representation
REP(NILP) ⇐ NILTRE                          function (mapping
CONCTWIS(A) ⇐ REPINV(TWIST(REP(A)))         pairs onto trees)
TWIST(CONSTREE(A, T1, T2)) ⇐
        CONSTREE(A, TWIST(T2), TWIST(T1))
TWIST(NILTRE) ⇐ NILTRE:
```

Stage 1: Production of inverse of representation function.

```
EQ1.PREXP;
REPINV(REP(A))⇐A:
:
:EQ1.TRYFINDINV;

AM ATTEMPTING TO SYNTHESISE A FUNCTION REPINV DEFINED
IMPLICITLY BY THIS EQUATION

REPINV(REP(A))⇐A

HOW DOES

REPINV(REP(A))⇐A

LOOK TO YOU
OK

WOULD YOU TYPE IN SUITABLE INSTANTIATION PAIRS.
BASE CASES FIRST.
:A, NILP END END

REPINV(NILTRE)⇐NILP
NOW THE RECURSIONS
A, CONSP(A, CONSP(B, C)) END END

HOW DOES

REPINV(REP(CONSP(A, CONSP(B, C))))⇐CONSP(REPINV(REP(A)),
CONSP(REPINV(REP(B)), REPINV(REP(C))))

LOOK TO YOU
OK
:
:
:.PREXP;
REPINV(CONSTREE(A, U0, U1))⇐CONSP(A, CONSP(REPINV(U0),
REPINV(U1)))
REPINV(NILTRE)⇐NILP:
```

Stage 2: Synthesis of concrete twist function, given inverse of representation function.

```
EQNLIST.PREXP;
```

```
REPINV(CONSTREE(A, T1, T2)) ⇐ CONSP(A, CONSP(REPINV(T1),
                                       REPINV(T2)))
REPINV(NILTRE) ⇐ NILP
REP(CONSP(A, CONSP(B, C))) ⇐ CONSTREE(A, REP(B), REP(C))
REP(NILP) ⇐ NILTRE
CONCTWIS(A) ⇐ REPINV(TWIST(REP(A)))
TWIST(CONSTREE(A, T1, T2))
  ⇐ CONSTREE(A, TWIST(T2), TWIST(T1))
TWIST(NILTRE) ⇐ NILTRE:
:
:
:
:
:START;
TYPE IN INSTANTIATED L.H.S. BASE CASES FIRST.
CONCTWIST(NILP) END
CONCTWIS(NILP) ⇐ NILP

NOW TYPE IN RECURSIVE BRANCHES:
CONCTWIST(CONSP(A, CONSP(B, C))) END

HOW DOES

CONCTWIS(CONSP(A, CONSP(B, C))) ⇐ CONSP(A, CONSP(CONCTWIS
                                        (C), CONCTWIS(B)))

LOOK TO YOU
OK

FINISHED.CPU TIME TAKEN 0 MINS 3 SECS

:.PREXP;
REPINV(CONSTREE(A, T1, T2)) ⇐ CONSP(A, CONSP(REPINV(T1),
                                       REPINV(T2)))
REPINV(NILTRE) ⇐ NILP
REP(CONSP(A, CONSP(B, C))) ⇐ CONSTREE(A, REP(B), REP(C))
REP(NILP) ⇐ NILTRE
TWIST(CONSTREE(A, T1, T2)) ⇐ CONSTREE(A, TWIST(T2),
                                     TWIST(T1))

TWIST(NILTRE) ⇐ NILTRE
CONCTWIST(NILP) ⇐ NILP
CONCTWIST(CONSP(A, CONSP(B, C))) ⇐ CONSP(A, CONSP(CONCTWIS
                                         (C), CONCTWIS(B))):
```

5.3. Pattern directed generalisation

The transformation system is greatly assisted if it knows the form of acceptable solutions to the problem it is tackling. One rudimentary way of achieving this at present is via the switch ONLYTOPFOLD. If this is set by the user the system only looks for solutions of the form

$$g(X) \Leftarrow \mathscr{E}(\cdots g(k(X)) \cdots)$$

where $\mathscr{E}$ may involve subsidiary recursive functions but these are similarly restricted to being defined by this pattern of recursion. If the user knows in advance that the solution is of this form the amount of searching the system has to do during folding is greatly reduced.

This idea of indicating solution patterns is being developed by a student, Martin Feather, and although it will be reported elsewhere when completed, I will give a brief account of it here. The facility is built on top of the present system and the dialogue follows the same pattern as before except that for each case the system asks the user to indicate the pattern of the anticipated result. The system then identifies which parts of the developing equation are to be associated with the components of the pattern and attempts to produce solutions to match this pattern. The patterns are expressions built up using existing function names and TOPFN and SYNFN. TOPFN matches with any function and SYNFN indicates that part of the developing expression is to be taken as the definition of a subsidary function. This facility seems particularly useful during synthesis and approximates to a top down programming style.

EXAMPLE 13. *Cartesian product using pattern directed generalization.* Cartesian product is specified as before, this time only for two sets.

```
EQNLIST.PREXP;
CART(X, Y)⇐⟨:MAKETUPL([X1::Y1]): X1 IN X
                                  Y1 IN Y:⟩:    —original equation
:
:START;
TYPE IN LIST OF SUITABLY INSTANTIATED LEFT HAND SIDES
(SEPARATED BY COMMAS, TERMINATED BY END)
:
:CART(NILSET, Y), CART(CONSSET(X3, X), Y) END

AM SEEKING SOLUTION FOR FOLLOWING EQUATION:
CART(⟨::⟩, Y)⇐⟨::⟩
AM TRYING FOLLOWING PATTERN              —system suggests pattern
⟨::⟩                                      automatically
CART(⟨::⟩, Y)⇐⟨::⟩
```

```
AM SEEKING SOLUTION FOR FOLLOWING EQUATION:
CART(CONSSET(X3, X), Y)⇐⟨:MAKETUPL([X3, Y1]): Y1 IN Y :⟩+
⟨MAKETUPL([X1, Y1]): X1 IN X Y1 IN Y :⟩
TYPE IN RECURSIVE CASES (END WITH END)
:NIL END                              —rejected by giving no pattern
TRY ANOTHER PATTERN?
:YES
TYPE IN SUITABLE PATTERN (END WITH END)
:TOPFN(CART(X, Y), SYNFN(X3, Y)) END—user

CART(CONSSET(X3, X), Y)⇐NEWF1(X3, Y)+CART(X, Y)
AND HAVE THE FOLLOWING NEW FUNCTION TO SYNTHESIZE:

NEWF1(X3, Y)⇐⟨:MAKETUPL([X3, Y1]): Y1 IN Y :⟩
NOW TACKLING FOLLOWING EQUATION:

NEWF1(X3, Y)⇐⟨:MAKETUPL([X3, Y1]): Y1 IN Y:⟩
TYPE IN A LIST OF SUITABLY INSTANTIATED LEFT HAND SIDES
(SEPARATED BY COMMAS, TERMINATED BY END)
:NEWF1(X3, NILSET), NEWF1(X3,          —user
                          CONSSET(Y3, Y)) END
AM SEEKING SOLUTION FOR FOLLOWING EQUATION:

NEWF1(X3, ⟨::⟩)⇐⟨::⟩
AM TRYING FOLLOWING PATTERN
⟨::⟩

NEWF1(X3, ⟨::⟩)⇐⟨::⟩
AM SEEKING SOLUTION FOR FOLLOWING EQUATION:

NEWF1(X3, CONSSET(Y3, Y))⇐SGTON(MAKETUPL([X3, Y3]))
                              +⟨:MAKETUPL([X3, Y1]):
                                      Y1 IN Y :⟩
TYPE IN RECURSIVE CASES (END WITH END)
:NEWF1(X3, Y) END
AM TRYING FOLLOWING PATTERN
TOPFN(Y3, NEWF1(X3, Y), X3)

NEWF1(X3, CONSSET(Y3, Y))⇐SGTON(MAKETUPL([X3, Y3]))
                              +NEWF1(X3, Y)
:
:
:EQNLIST.PREXP;
```

```
NEWF1(X3, CONSSET(Y3, Y))⇐SGTON(MAKETUPL([X3, Y3]))
                                   +NEWF1(X3, Y)
NEWF1(X3, ⟨::⟩)⇐⟨::⟩
CART(CONSSET(X3, X), Y)⇐NEWF1(X3, Y)+CART(X, Y)
CART(⟨::⟩, Y)⇐⟨::⟩
```

EXAMPLE 14. *Diagonal Search.* We wish to find all pairs of members of a set that satisfy some reflexive, symmetric relation R. Using pattern directed generalisation the system is able to optimise a simple specification to a set of equations that perform diagonal search. As the equations as typed out are rather hard-going, we have departed from our policy of presenting the telytype output unedited. However, the dialogue is just as presented, we have only 'pretty-printed' the equations.

```
EQNLIST.PREXP;
G(X)⇐{⟨x1, y1⟩ | x1∈X, y1∈X & R(x1, y1)}
                              (the facts about R are given by the redlist)
:START;
TYPE IN LIST OF SUITABLY INSTANTIATED LEFT HAND SIDES
(SEPARATED BY COMMAS, TERMINATED BY END)
:G(nilset), G(consset(x2, X)) END
AM SEEKING SOLUTION FOR FOLLOWING EQUATION
G({})⇐{}
HAVE FOUND BASECASE
G({})⇐{}

AM SEEKING SOLUTION FOR FOLLOWING EQUATION:

G(consset(x2, X))⇐{⟨x2, x3⟩}
                    +{⟨x2, y1⟩ | y1∈X & R(y1, x2)}
                    +{⟨x1, x2⟩ | x1∈X & R(x1, x2)}
                    +{⟨x1, y1⟩ | x1∈X, y1∈X & R(x1, y1)}

TYPE IN SUITABLE PATTERN (END WITH END)
:TOPFN(G(X), SYNFN(X2, X)) END
HAVE FOUND FOLLOWING SOLUTION

G(consset(x2, X))⇐{⟨x2, x2⟩} + (u1 + u2 + G(X))
                wherep ⟨u1, u2⟩ = = newf2(x2, X)

AND HAVE THE FOLLOWING NEW FUNCTIONS TO SYNTHESISE

newf2(x2, X)⇐⟨{x2, y1⟩ | y1∈X & R(y1, x2)}
              {⟨x1, x2⟩ | x1∈X & R(x1, x2)}⟩
```

```
TYPE IN LIST OF SUITABLY INSTANTIATED LEFT HAND SIDES
SEPARATED BY COMMAS TERMINATED BY END

:newf2(x2, nilset), newf2(x2, consset(x3, X)) END

AM SEEKING SOLUTION FOR FOLLOWING EQUATION

newf2(x2, { })⇐⟨{ }. { }⟩

HAVE FOUND BASECASE

newf2(x2, { })⇐⟨{ }, { }⟩

AM SEEKING SOLUTION FOR FOLLOWING EQUATION

newf2(x2, consset(x3, X))⇐
     if R(x2, x3)
     then ⟨{⟨x2, x3⟩} + {⟨x2, y1⟩ | y1 ∈ X & R(y1, x2)}
            {⟨x3, x2⟩} + {⟨x1, x2⟩ | x1 ∈ X & R(x1, x2)}⟩
     else ⟨{⟨x2, y1⟩ | y1 ∈ X & R(y1, x2)}
            {⟨x1, x2⟩ | x1 ∈ X & R(x1, x2)}⟩

TYPE IN SUITABLE PATTERN (END WITH END)

:topfn(newf2(x)) END

HAVE FOUND FOLLOWING SOLUTION

newf2(x2, consset(x3, X))⇐
            if R(x2, x3)
            then ⟨{⟨x2, x3⟩} + u3, {⟨x3, x2⟩} + u4⟩
            wherep ⟨u3, u4⟩ = = newf2(x2, X)
            else newf2(x2, X)

FINISHED ALL FUNCTIONS


:EQNLIST.PREXP;

newf2(x2, consset(x3, X))⇐
            if R(x2, x3)
            then ⟨{⟨x2, x3⟩} + u3, {⟨x3, x2⟩} + u4⟩
                  wherep ⟨u3, u4⟩ = = newf2(x2, X)
            else newf2(x2, X)
```

$\text{new}f2(x2, \{\}) \Leftarrow \langle\{\}, \{\}\rangle$
$G(\text{consset}(x2, X)) \Leftarrow \{\langle x2, x2\rangle\} + (u1 + u2 + G(X))$
$\qquad \textit{wherep } \langle u1, u2\rangle = = \text{new}f2(x2, X)$
$G(\{\}) \Leftarrow \{\}$

6. Eureka Elimination

In Burstall and Darlington [4], several of the examples required the user to provide auxiliary definitions of 'eurekas' to assist the system. Since then we have mechanised the invention of auxiliary functions involving tuples such as we used in the fibonacci example. Here the key lay in introducing the auxiliary definition

$$g(n) \Leftarrow \langle \text{fib}(n+1), \text{fib}(n)\rangle$$

A function findclevertuple is now provided for the user which when given the main equation for fibonacci

$$\text{fib}(n+2) \Leftarrow \text{fib}(n+1) + \text{fib}(n)$$

produces just the equation above.

Example 15. *Invention of auxilary fibonacci function.*

```
E3.PREXP;
FIB((N + 1) + 1)⇐FIB(N + 1) + FIB(N):
:E3.FINDCLEVERTUPLE;
:
HOW DOES

NEWF3(N)⇐MAKETUPLE([FIB(N + 1), FIB(N)])

LOOK TO YOU
OK
:
```

The system achieves this by implementing the process described in Burstall and Darlington [4], that is looking for matching tuples in the expanding computation tree for fib($n+2$). An interesting point is that when it has achieved the definition for fib($n+2$) it has done most of the work required for the complete optimisation of fibonacci. However, we have not as yet merged these two processes.

Of the other areas of user assistance still required, I feel that it should be fairly easy to remove from the user the task of giving the correct instantiations.

The system could store a number of well-orderings with appropriate base cases, for example $x::X$ and nil, $x1::x2::X$, $x::$nil and nil and $n+1$ and 0, and could then use the typing information that NPL provides to select those appropriate. Using this process it should be possible, in the majority of cases, to ensure that the algorithms produced definitely terminate. However, this would involve the system in extra search for no useful purpose at the moment. Of the other 'eurekas' the main one is involved in supplying auxiliary definitions to enable the translation of recursive definitions to iterative form. These auxiliary definitions are closely related to the invariants required in the proofs of iterative programs and although much promising work has been done in this area [1, 14] we are not yet in a position to be able to automate their invention, although we now have enough experience to produce them without too much thought when required.

7. Conclusion

The conclusion of this report gives the state of the system as of Summer 1977. As stated the system is regarded as an experimental tool and will continue to develop.

We hope that program transformation techniques can be utilised to provide a practical programming tool and have started the design of systems to accomplish this. These systems will put more control back with the user requiring him to have an intuitive idea of the development of his algorithm leaving to the system of task of implementing this development correctly. At the moment we see the progression from specification to efficient algorithm in two stages, firstly a top-down production of a tree structured program and a transformation process that optimises this program, intertwinning the various computations that take place.

ACKNOWLEDGMENTS

The transformations and the basic system were developed during joint work with R.M. Burstall at Edinburgh University and I am grateful to him for the years of fruitful and enjoyable collaboration. We benefited greatly from interactions with R. Boyer and J. Moore whose LISP theorem prover provided many ideas for the transformation system and Manna and Waldinger whose work on program synthesis encouraged us to continue with this aspect of our work. I would like to thank all my colleagues in the Department of Artificial Intelligence, Edinburgh University for their help and encouragement. Special thanks are due to M. Feather for his work on the system. The studies of R. Kowalski and K. Clark at Imperial College into the use of predicate logic as a programming language has provided valuable new insights into our work. The excellent typing was done by Eleanor Kerse at Edinburgh and by Elizabeth Larch at Imperial College. The work was supported by grants from the Science Research Council.

REFERENCES

1. Aubin, R., Some generalisation heuristics in proofs by induction, *Proc. IRIA Symposium on Proving and Improving Programs*, Arcet-Senans, France (1975) pp. 197–208.

2. Boyer, R.S. and Moore, J.S., Proving theorems about LISP functions, *J. Assoc. Comput. Mach.* **22** (1) (1975) 129–144.
3. Burstall, R.M., Design considerations for a functional programming language, *Proc. Infotech State of the Art Conf.*, Copenhagen (1977).
4. Burstall, R.M. and Darlington, J., A transformation system for developing recursive programs, *J. Assoc. Comput. Mach.* **24** (1) (1977) 44–67.
5. Burstall, R.M. and Goguen, J.A., Putting theories together to make specifications, invited paper Fifth International Joint Conf. on Artificial Intelligence, Boston, MA. (August 1977).
6. Burstall, R.M., Collins, J.S. and Popplestone, R.J., *Programming in POP-2 Edinburgh* (Edinburgh University Press, 1971).
7. Darlington, J., Application of program transformation to program synthesis, *Proc. IRIA Symposium on Proving and Improving Programs*, Arc-et-Senans, France (1975) pp. 133–144.
8. Darlington, J., A synthesis of several sorting algorithms, Research Report No. 23, Department of Artificial Intelligence, University of Edinburgh (1976).
9. Darlington, J., The formal development of a priority queue algorithm, in preparation.
10. Goguen, J.A., Thatcher, J.W. and Wagner, E.G., An initial approach to the specification, correctness and implementation of abstract data types, in: R.T. Yeh, Ed., *Current Trends in Programming Methodology, Vol. 3, Data Structuring* (Prentice-Hall, Englewood Cliffs, NJ, 1977).
11. Guttag, J.V., The specification and application to programming of abstract data types, CSRG-S9, University of Toronto (1975).
12. Kowalski, R., Predicate logic as a programming language, in: *Proc. of IFIP Congress 1974* (North-Holland, Amsterdam, 1974) pp. 569–574.
13. Manna, Z. and Waldinger, R., Knowledge and reasoning in program synthesis, *Artificial Intelligence* **6** (2) (1975) 175–208.
14. Moore, J.S., Introducing iteration into the pure LISP theorem prover, CSL-74-3, Xerox Palo Alto Res. Center, Palo Alto, CA (1975).
15. Topor, R.W., Interactive program verification using virtual programs, Thesis, Department of Artificial Intelligence, University of Edinburgh (1975).

Received 21 May 1979