TOWARDS THE COMPILATION OF ANNOTATED LOGIC PROGRAMS

Steve Gregory

Research Report DOC 80/16

June 1980

ABSTRACT

IC-PROLOG provides a simple system of annotations to specify the control component of a program. These are used to instruct the interpreter in the manner in which the program is to be executed. We consider these annotations and investigate the "compilation" of annotated programs into strictly sequential programs, which can then be either interpreted or compiled further. Lastly, a semi-automatic system is presented, which effects the compilation of simple annotated programs.

Department of Computing
Imperial College of Science and Technology
University of London
180 Queen's Gate
London SW7 2BZ

Telephone: 01-589 5111 Telex: 261503

CONTENTS

APPENDIX 1

Listing of the program

APPENDIX 2

Example runs

INTRODUCTION

Two important methods have been developed in recent years for modifying the behaviour of recursive programs, in particular those written in the PROLOG logic programming language. One is the technique of program transformation, developed originally by Burstall and Darlington [1]. The other is the use of control annotations, as incorporated in the IC-PROLOG system [5].

At the same time, a PROLOG system has been implemented at Edinburgh [21,22] which differs from most PROLOGs insofar as it partially compiles programs. This approach to PROLOG implementation has obvious advantages regarding execution speed.

The aim of the present project is to investigate the feasibility of "compiling" logic programs bearing the control annotations of IC-PROLOG into sequential logic programs, using transformation techniques. We will see that annotated logic programs can be semi-automatically compiled by means of the same abstract machine as used for interpretation.

Chapter 1 outlines the control annotations available in IC-PROLOG plus some which are shortly to be implemented. It closes with a description of a useful point of logic programming methodology.

Chapter 2 begins with an outline of program transformation and its application to Horn clause programs. We find that transformation consists largely of symbolic execution and that this can be controlled by annotations. The chapter closes with a discussion of a class of programs which cannot be transformed in the usual manner; these require the problem to be reformulated.

Chapter 3 describes SCALP (a System to Compile Annotated Logic Programs) which has been implemented. This system performs transformation by coroutined symbolic execution. We describe the implementation of the system and finally suggest extensions to its capabilities.

The appendices include the program listing and several example runs. Most of these are examples which have been transformed in chapter 2.

CHAPTER 1

THE CONTROL COMPONENT OF LOGIC PROGRAMS

Ever since Kowalski [15] proposed that

Algorithm = Logic + Control

the advantages of separating the logic and control components of a program have been well known. Ideally, the logic component should be a clear and obviously correct statement of the problem while the control component is responsible for the efficiency of the algorithm.

In this chapter we examine methods of specifying the control component, particularly of PROLOG programs.

1.1 Notation and terminology

In this report, a **clause** is generally taken to mean a Horn clause, with one consequent atom and a number of antecedents. A **goal clause** is a clause with no consequent, while a **goal atom** is the sole atom in a goal clause with only one antecedent.

In a procedural context, a **procedure** is a clause, with a **procedure head** (the consequent) and a **procedure body** consisting of a number of **procedure calls**. We shall refer to the independent execution of a call as a **process**.

A **search tree** has nodes each labelled by either a goal clause or X (signifying failure) or □ (success). The offsprings of a given node denote all possible results of executing one step of that node's selected call.

A **proof tree** has nodes each labelled by an atom. The offsprings of a node denote each of the antecedents of the clause whose consequent unified with that node's atom. There is a proof tree associated with each branch of a search tree.

These terms are defined more fully in [3].

The syntax of IC-PROLOG will be adopted in the examples, i.e. a clause is written in the form

$$B <- A_1 \& A_2 \& \ldots \& A_n$$

variables, functors and most predicates are written in lower case, while constants and "primitive" predicates are in upper case.

1.2 Specifying the control component

The **computation rule** determines which atom of a goal clause is to be selected for execution. This defines a search tree for a given program and goal clause. It also determines the order in which proof trees are constructed. The **search rule** determines the order in which the search tree is constructed.

Most PROLOG systems have an implicit control component in the form of a default search rule and computation rule. The normal default search rule is to try each of the clauses for the selected call's predicate in the order in which they are written, thus constructing the search tree left - right depth first. The default computation rule is the selection of the leftmost call in each goal clause - corresponding to the left - right depth first construction of the proof tree.

Although some PROLOGs, including IC-PROLOG, provide such facilities as indexing of clauses to restrict the search tree, the most important feature of IC-PROLOG for our purposes is the ability to modify the computation rule by means of annotations. These features are outlined in subsequent sections and more fully in [3,5], but first we consider an earlier scheme of control annotations.

Control annotations in recursion equations

Schwarz [19] has proposed an elaborate system of annotations to control the behaviour of programs written in the recursion equation formalism. These enable the programmer to select various evaluation mechanisms such as:

Call-by-value / -need / -opportunity.

Lazy evaluation.

"Almost tail recursion".

Destructive operations.

1.3 Dataflow coroutining

Liberating sequential execution

The performance of a given logic program can often be improved by constructing the proof tree other than in the default left - right depth first order. This means that a call will no longer necessarily be run to completion before the next call is entered. For example, suppose we have the procedure

```
sort(x,y) <- perm(x,y) & ord(y)
```

If we wish to sort the list x by executing the goal <- sort(x,y) where only x is instantiated, the sort procedure will generate all permutations one by one and then find whether each is ordered. It would be preferable to coroutine between the perm and ord calls in order that the generation of each permutation by perm is terminated as soon as ord fails. Coroutining between calls can also sometimes make a nondeterministic program deterministic. For example, consider the procedure

```
front(n,x,z) <- append(x,y,z) & length(x,n)
```

where front(n,x,z) holds when x is the initial sublist of length n, of list z. If the goal <- front(n,x,z) is executed, where n and z are instantiated, then a succession of values for x will be generated by the append call and each checked by the length call (similar to the sort example above). In this case, however, coroutining between the append and length calls intertwines the two executions such as to eliminate the nondeterminism.

Eager consumers and lazy producers

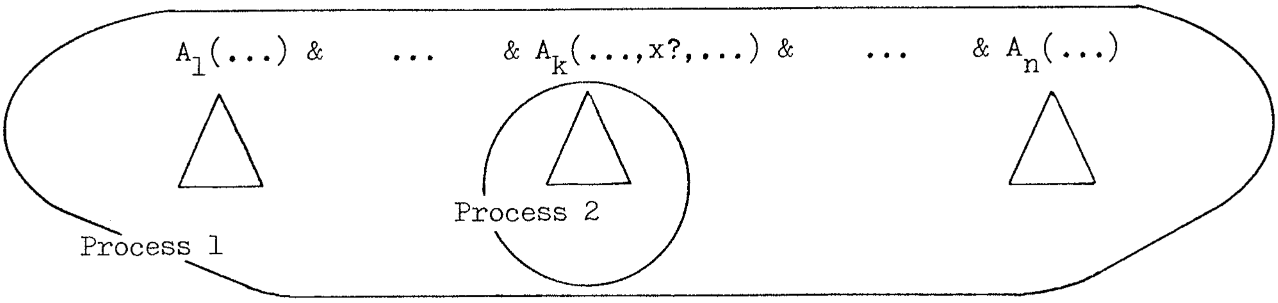
In IC-PROLOG we can program a coroutining interaction by suffixing an argument of a call by a "?" or "↑" annotation. For the sake of simplicity and without loss of generality, we shall henceforth assume that all terms annotated by "?" or "↑" are variables. Suppose there is a procedure whose body is

$$A_1(t_{11},\ldots,t_{1m_1}) \ \& \ \ldots \ \& \ A_k(t_{k1},\ldots,x,\ldots,t_{km_k}) \ \& \ \ldots \ \& \ A_n(t_{n1},\ldots,t_{nm_n})$$

where the variable x is shared with at least one of the calls $A_1,\ldots,A_{k-1}$. We can make A_k an **eager consumer** of x by attaching a "?" annotation:

$$A_1(t_{11},\ldots,t_{1m_1}) \ \& \ \ldots \ \& \ A_k(t_{k1},\ldots,x?,\ldots,t_{km_k}) \ \& \ \ldots \ \& \ A_n(t_{n1},\ldots,t_{nm_n})$$

The effect of this is to allow the subtree of the proof tree rooted at the A_k atom to be constructed independently. We can regard the resulting computation as comprising two processes, one constructing the whole proof tree and another constructing the subtree rooted at A_k:



Although each process still constructs its proof tree (or subtree) in left - right depth first order, the execution of the processes is interleaved according to the binding of certain variables. In the situation depicted above, a coroutining jump or return will take place whenever the variable x - or any variable of a term to which x is bound - becomes bound to a non-variable term. Whether a jump or a return is made depends upon the current point in the proof tree:

If the current point is not in the subtree rooted at A_k, the current point is saved and a coroutining jump made to that subtree, at its last suspension point.

If the current point is in the subtree rooted at A_k, the current point is saved and a return is made to the "return address" saved at the time of the last jump to this subtree. Before returning, the last execution step - which caused the binding - must be undone.

The rationale behind this is that, since A_k is an eager consumer of x, it is not permitted to bind x. When it needs x to be bound, this must be done by one of the calls $A_1,\ldots,A_{k-1}$. When one of these calls supplies a binding for x, the execution of A_k is immediately resumed.

Complementary to this is the ability to specify that A_k is a **lazy producer** of x, by attaching a "↑" annotation:

$$A_1(t_{11},\ldots,t_{1m_1}) \ \& \ \ldots \ \& \ A_k(t_{k1},\ldots,x\uparrow,\ldots,t_{km_k}) \ \& \ \ldots \ \& \ A_n(t_{n1},\ldots,t_{nm_n})$$

In this case, whenever one of the calls $A_1,\ldots,A_{k-1}$ needs a binding for x it suspends whilst A_k executes. When A_k supplies the required binding, the execution of $A_1,\ldots,A_{k-1}$ resumes.

The action to be taken whenever x is bound, again depends upon the current point:

If the current point is not in the subtree rooted at A_k, the current point is saved and a coroutining jump made to that subtree, at its last suspension point. Before jumping, the last execution step - which caused the binding - must be undone.

If the current point is in the subtree rooted at A_k, the current is saved and a return is made to the "return address" saved at the time of the last jump to this subtree.

There may of course be any number of annotated calls in a given procedure body, and coroutining may occur at several levels. However, we impose the following restrictions:

a) No call may contain more than one annotated argument.

b) No variable may be annotated more than once in a procedure body.

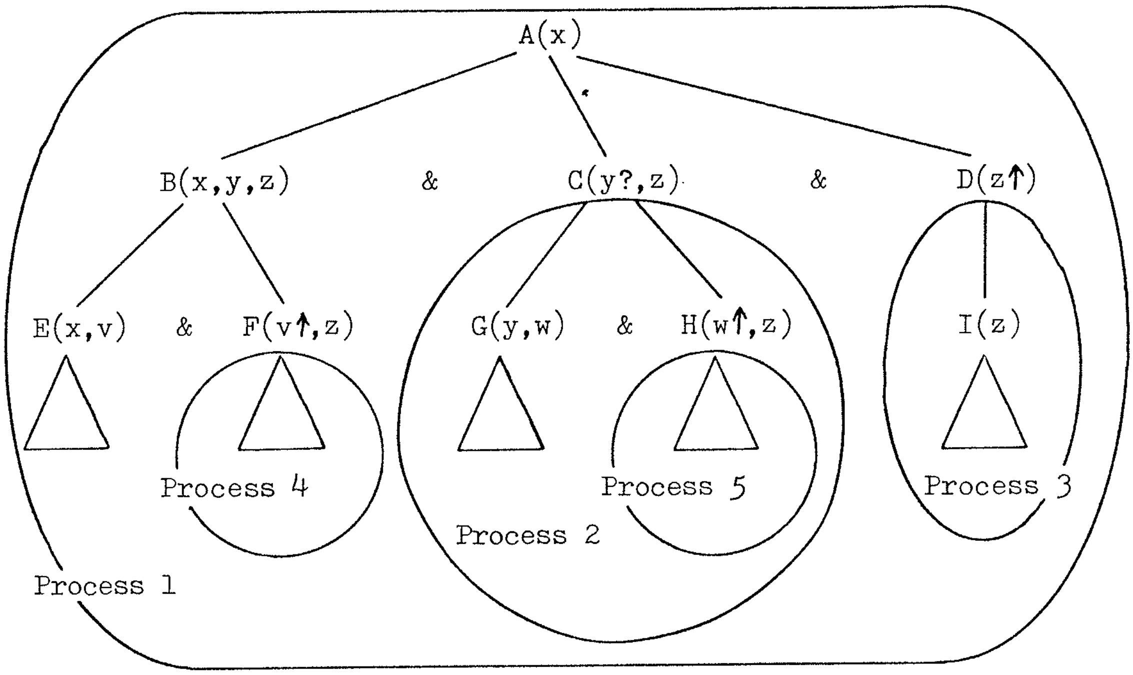
c) For any two annotated variables x and y, y must not occur in any term to which x is bound. This is a generalization of b).

In practice, any limitation resulting from the above restrictions often be overcome by the use of the parallel scheme, to be described in section 1.4.

We mentioned earlier the role of processes in coroutining. To illustrate this, we shall take as an example the following procedure set:

```
A(x) <- B(x,y,z) & C(y?,z) & D(z↑)
B(x,y,z) <- E(x,v) & F(v↑,z)
C(y,z) <- G(y,w) & H(w↑,z)
D(z) <- I(z)
```

A process is created for each annotated atom placed in the proof tree and is responsible for the construction of the subtree rooted at that atom. The structure of processes involved in the example is depicted below:



The rules governing coroutining interaction may now be stated in the framework of processes:

When a variable, say x, is bound to a non-variable term, determine whether there exists a process engaged in the construction of the subtree rooted at an atom in which y is annotated, where either y is x or y is bound to a term containing x. (The restrictions quoted earlier ensure that there is not more than one such process.) If such a process exists, determine whether it is a consumer (its root atom contains a "?") or a producer (its root atom contains a "↑"). Also determine whether we are currently constructing the subtree rooted at that process's root atom - in which case the process is **active** - or elsewhere - the process is **passive**.

1) If the process is an active consumer or passive producer, undo the last execution step.

2) If the process is passive, suspend the current process and jump. Else (the process is active), return from the current process.

In the above example, suppose that process 5 is currently executing the H call. If w becomes bound to a non-variable, a return will be effected to process 2 which is executing the C call. If instead z is bound by process 5, the binding will be undone and a jump effected to process 3, returning as soon as the latter supplies the required binding of z.

Two questions remain unanswered:

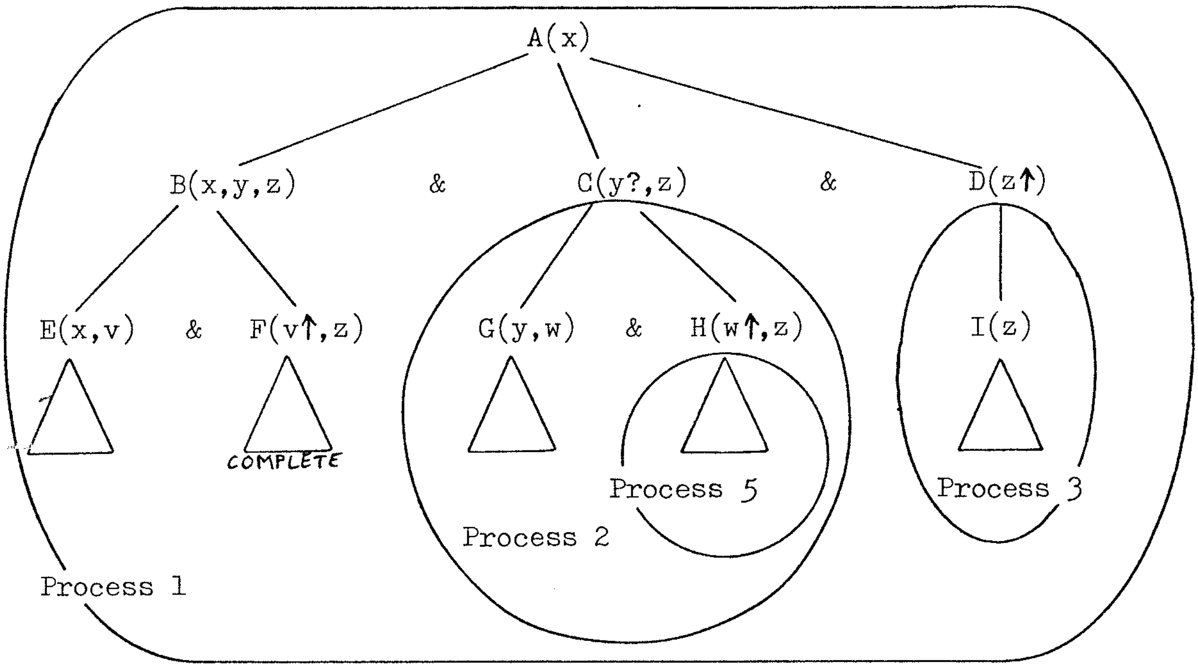
a) What happens when a process finishes, i.e. when its proof tree is complete? In the conjunction

$$A_1(x) \mathbin{\&} \ldots \mathbin{\&} A_{k-1}(x) \mathbin{\&} A_k(x?) \mathbin{\&} \ldots \mathbin{\&} A_n(x)$$

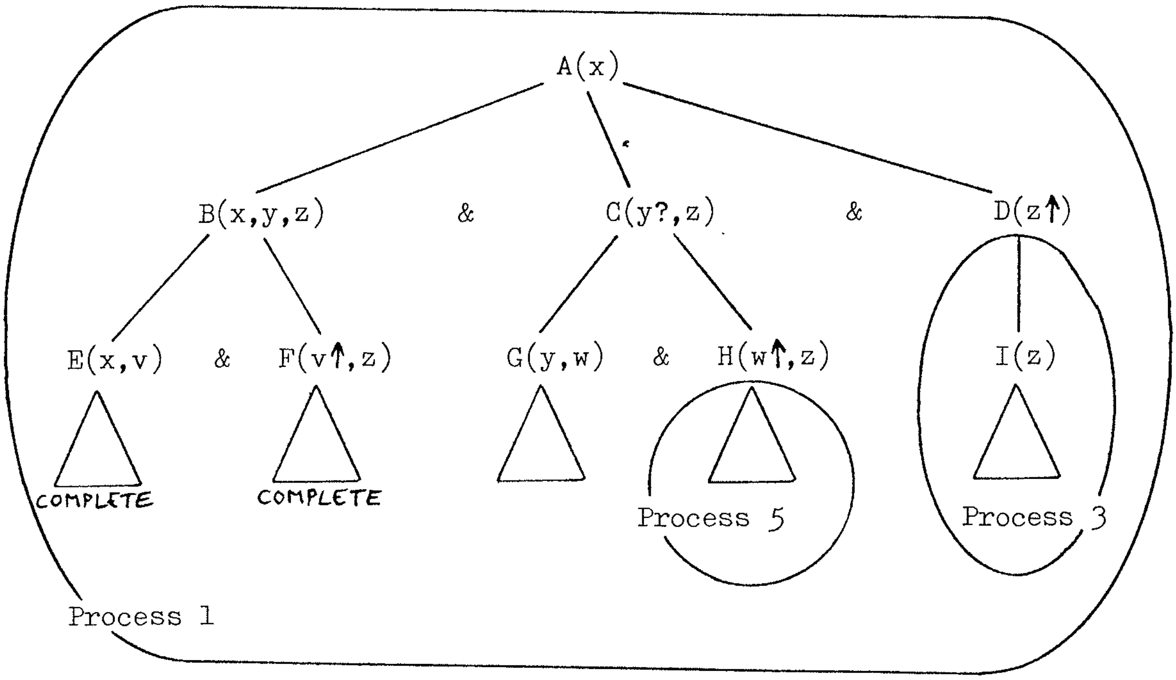
this would occur when the execution of A_k is complete.

b) What happens when a process constructing a proof tree attempts to enter a subtree which is being constructed by an inner process? This would occur when the execution of all of the calls $A_1,\ldots,A_{k-1}$ is complete.

When a process finishes, a return is forced and the process is removed, thus terminating the associated coroutining interaction. In the example above, suppose that process 4 finishes owing to the completion of the execution of F. Process 4 is removed and the execution of E is resumed:



If a process tries to enter an annotated call, i.e. one whose subtree is being constructed by an inner process, the inner process is removed and the construction of its subtree is taken over by the outer process. This terminates the corresponding coroutining interaction. In the example, suppose that the execution of the E call is now completed and hence so is that of B: process 1 now tries to enter the C call. Process 2 is removed and the execution of C is continued by process 1:



Delaying the data transfer

The **clause bar** is often useful in the execution of coroutined programs. This annotation, provided in IC-PROLOG, is a ":" which is written in place of "&" at no more than one point in a procedure body. It has the effect of delaying any coroutining jump or return until all calls to the left of the clause bar have been run to completion.

An example

Example 1 illustrates the features described so far, including an eager consumer, lazy producer and clause bar.

Example 1

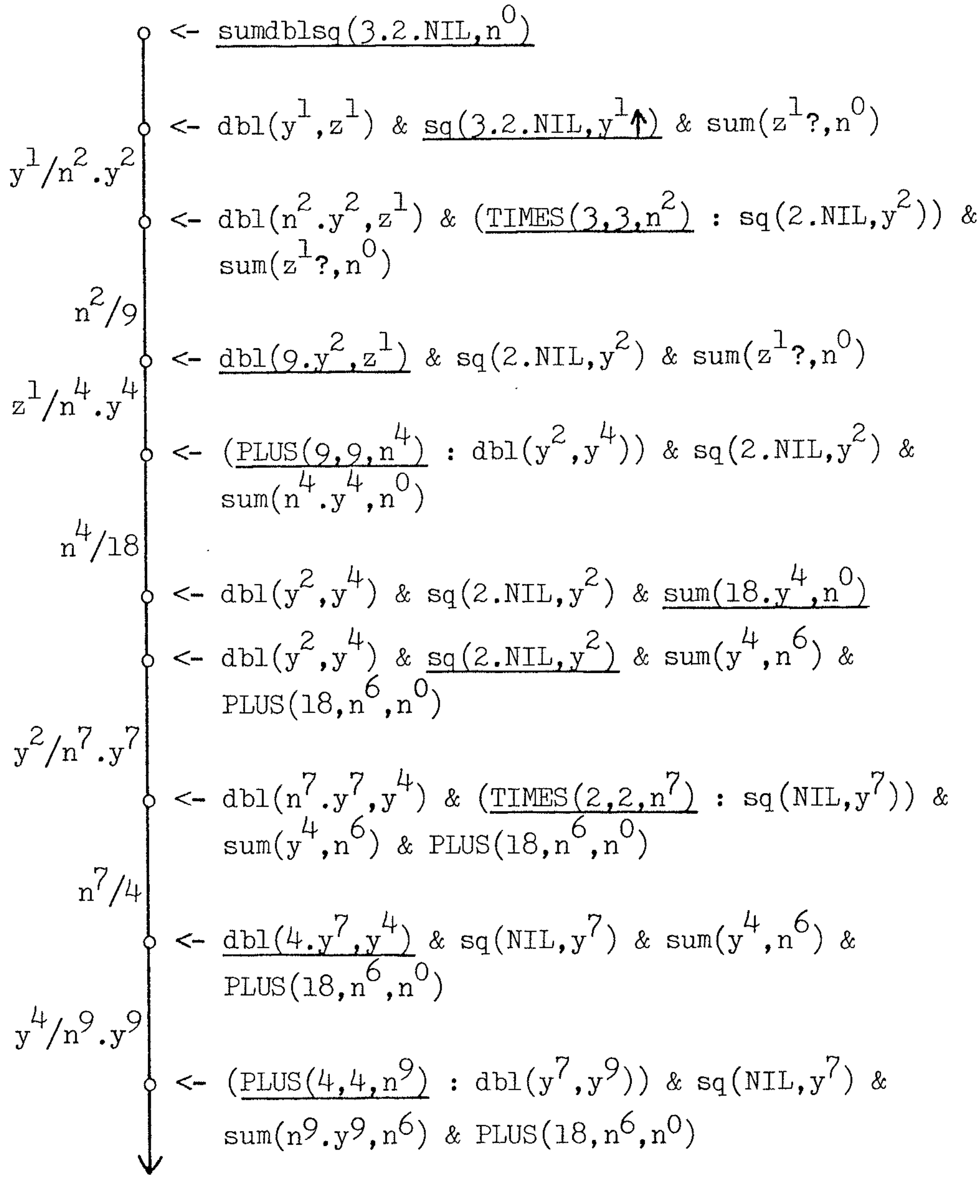
```
sumdblsq(x,n) <- dbl(y,z) & sq(x,y↑) & sum(z?,n)

dbl(NIL,NIL) <-
dbl(m.x,n.y) <- PLUS(m,m,n) : dbl(x,y)

sq(NIL,NIL) <-
sq(m.x,n.y) <- TIMES(m,m,n) : sq(x,y)

sum(NIL,0) <-
sum(m.x,p) <- sum(x,n) & PLUS(m,n,p)
```

This is a program for sumdblsq, where sumdblsq(x,n) holds when n is the sum of the doubles of squares of integers in the list x, i.e.

$$n = \sum_{k \text{ in } x} 2k^2 .$$

It works by coroutining together the separate activities of squaring, doubling and summing a list. The clause bar in the dbl procedure forces the PLUS to be computed before jumping; similarly for the sq procedure. Without the clause bar, all PLUS and TIMES calls would be saved until the end of the execution.

We exhibit the search tree of the execution of the goal <- sumdblsq(3.2.NIL,n) , which happens to be deterministic. Note that the variables in each procedure are superscripted by the "level number" to ensure they are distinct; this method is adopted in all subsequent examples. Also, a fail node (X) is only shown when there are no procedures that unify with the selected call.



<- $\underline{\text{sumdblsq}(3.2.\text{NIL},n^0)}$

<- $\text{dbl}(y^1,z^1)$ & $\underline{\text{sq}(3.2.\text{NIL},y^1\uparrow)}$ & $\text{sum}(z^1?,n^0)$

$y^1/n^2.y^2$

<- $\text{dbl}(n^2.y^2,z^1)$ & $(\underline{\text{TIMES}(3,3,n^2)}$: $\text{sq}(2.\text{NIL},y^2))$ & $\text{sum}(z^1?,n^0)$

$n^2/9$

<- $\underline{\text{dbl}(9.y^2,z^1)}$ & $\text{sq}(2.\text{NIL},y^2)$ & $\text{sum}(z^1?,n^0)$

$z^1/n^4.y^4$

<- $(\underline{\text{PLUS}(9,9,n^4)}$: $\text{dbl}(y^2,y^4))$ & $\text{sq}(2.\text{NIL},y^2)$ & $\text{sum}(n^4.y^4,n^0)$

$n^4/18$

<- $\text{dbl}(y^2,y^4)$ & $\text{sq}(2.\text{NIL},y^2)$ & $\underline{\text{sum}(18.y^4,n^0)}$

<- $\text{dbl}(y^2,y^4)$ & $\underline{\text{sq}(2.\text{NIL},y^2)}$ & $\text{sum}(y^4,n^6)$ & $\text{PLUS}(18,n^6,n^0)$

$y^2/n^7.y^7$

<- $\text{dbl}(n^7.y^7,y^4)$ & $(\underline{\text{TIMES}(2,2,n^7)}$: $\text{sq}(\text{NIL},y^7))$ & $\text{sum}(y^4,n^6)$ & $\text{PLUS}(18,n^6,n^0)$

$n^7/4$

<- $\underline{\text{dbl}(4.y^7,y^4)}$ & $\text{sq}(\text{NIL},y^7)$ & $\text{sum}(y^4,n^6)$ & $\text{PLUS}(18,n^6,n^0)$

$y^4/n^9.y^9$

<- $(\underline{\text{PLUS}(4,4,n^9)}$: $\text{dbl}(y^7,y^9))$ & $\text{sq}(\text{NIL},y^7)$ & $\text{sum}(n^9.y^9,n^6)$ & $\text{PLUS}(18,n^6,n^0)$

$n^9/8$

$$\leftarrow \text{dbl}(y^7,y^9)\ \&\ \text{sq}(\text{NIL},y^7)\ \&\ \underline{\text{sum}(8.y^9,n^6)}\ \&\ \text{PLUS}(18,n^6,n^0)$$

$$\leftarrow \text{dbl}(y^7,y^9)\ \&\ \underline{\text{sq}(\text{NIL},y^7)}\ \&\ \text{sum}(y^9,n^{11})\ \&\ \text{PLUS}(8,n^{11},n^6)\ \&\ \text{PLUS}(18,n^6,n^0)$$

y^7/NIL

$$\leftarrow \underline{\text{dbl}(\text{NIL},y^9)}\ \&\ \text{sum}(y^9,n^{11})\ \&\ \text{PLUS}(8,n^{11},n^6)\ \&\ \text{PLUS}(18,n^6,n^0)$$

y^9/NIL

$$\leftarrow \underline{\text{sum}(\text{NIL},n^{11})}\ \&\ \text{PLUS}(8,n^{11},n^6)\ \&\ \text{PLUS}(18,n^6,n^0)$$

$n^{11}/0$

$$\leftarrow \underline{\text{PLUS}(8,0,n^6)}\ \&\ \text{PLUS}(18,n^6,n^0)$$

$n^6/8$

$$\leftarrow \underline{\text{PLUS}(18,8,n^0)}$$

$n^0/26$

$$\square$$

1.4 Parallel execution

A new computation rule is shortly to be implemented in IC-PROLOG. A new annotation "//", used in place of "&", is interpreted declaratively as conjunction but dictates that calls thus conjoined are to be executed concurrently. In a multi-processor environment, the annotation could be used to reduce execution time by means of real parallelism. Hogger [14] gives as an example the goal clause

<- member(E,A) // member(E,B)

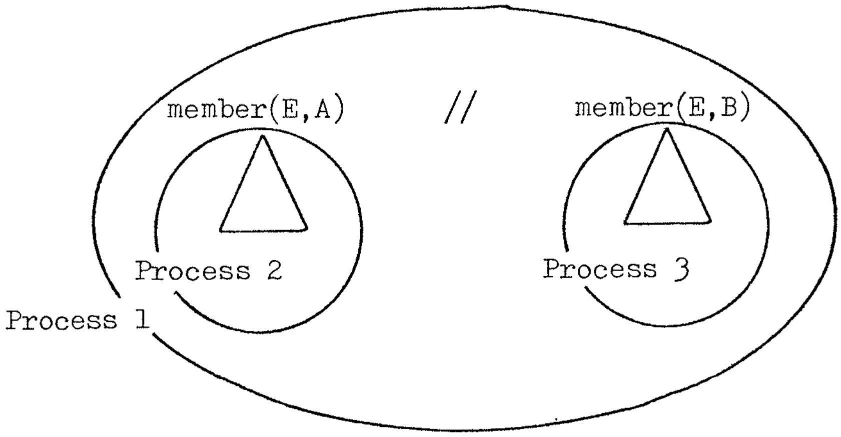
for determining whether E is a member of the intersection of sets A and B. The two member calls would be run on separate processors to halve the maximum execution time. This example is very simple in that the calls do not manipulate shared variables, but in general provision must be made for synchronization and communication between parallel processes.

In the present project, we are concerned only with pseudo-parallelism as a means of modifying the behaviour of programs. There would be no advantage in running the above goal clause in pseudo-parallel; the benefits to be obtained arise from the same considerations of "intertwining" that inspire coroutining. We shall see examples of this later.

The mechanism to be described here for pseudo-parallelism is a slightly modified version of that to be employed by IC-PROLOG. The way in which the goal clause

<- member(E,A) // member(E,B)

would be executed is by initiating a process to construct the proof tree rooted at each of the member atoms:



The two processes (2 and 3) are interleaved by executing one step of each in turn until one of them finishes, whereupon the other is run to completion. When both are finished, process 1 takes over (and promptly finishes).

Suspension of parallel processes

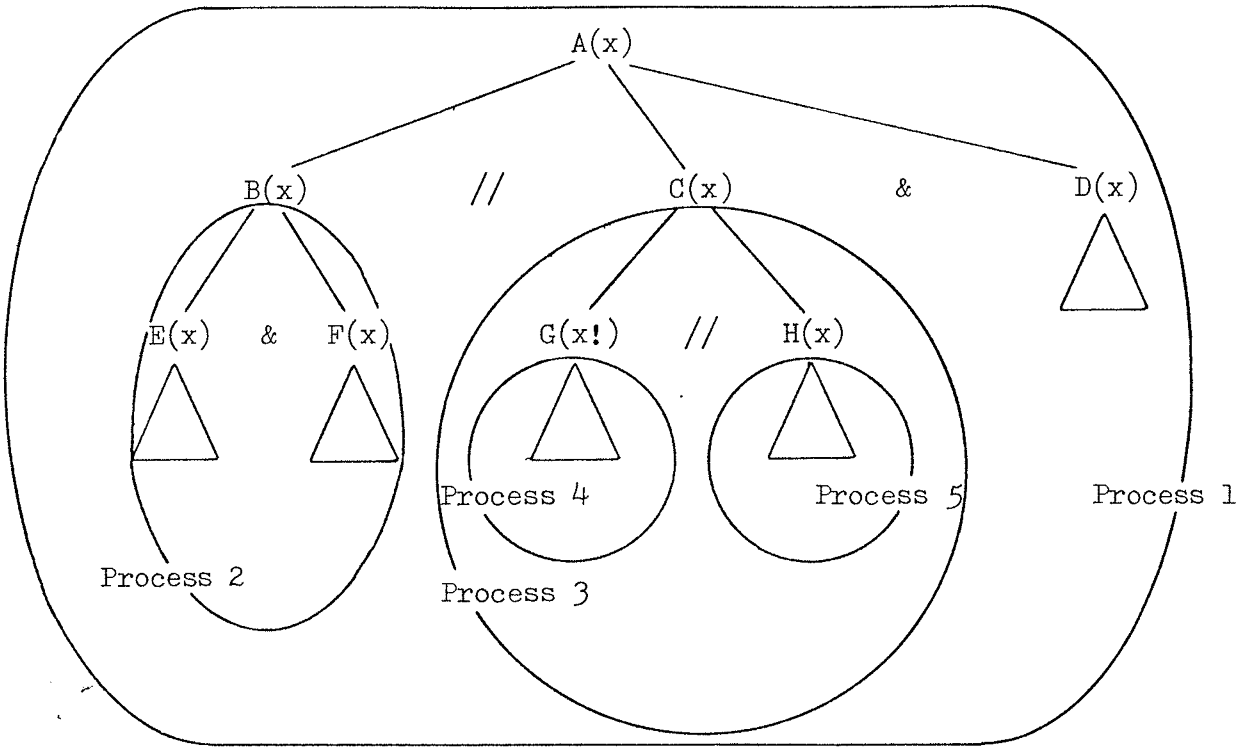
A simple means of synchronizing processes executing in pseudo-parallel is the "!" annotation. By attaching a "!" annotation to one or more arguments of a call we can specify that the call should be executed only if each such argument is bound to a non-variable term. If not, then the process suspends until this requirement is satisfied.

In this scheme, processes are created whenever the current goal clause in the search tree begins with $(C_1//\ldots//C_n)$ where each C_i is a conjunction - we assume, for the sake of simplicity, that each C_i is a single call. A process P_i is created to execute each C_i and these processes are run in (pseudo-)parallel. The process which was previously executing suspends until all processes $P_1,\ldots,P_n$ have finished. The other reason for which a process may suspend is as a result of the "!" annotation mentioned above.

For example, consider the following simple example:

```
A(x) <- (B(x) // C(x)) & D(x)
B(x) <- E(x) & F(x)
C(x) <- G(x!) // H(x)
```

and the corresponding proof tree at some point in the computation:



The first step was for process 1 to execute one step of the A call, which brought into being processes 2 and 3 and caused process 1 to suspend. Process 2 then executed one step of the B call and process 3 one step of the C call. This latter action created processes 4 and 5 and caused process 3 to suspend. In the state depicted, there are five processes of which three are suspended: processes 1 and 3, waiting for their subsidiary processes to finish, and process 4, waiting for x to be instantiated. If process 2 now binds x to a non-variable, process 4 will become active again, joining processes 2 and 5.

Suppose now that process 4 finishes: it is removed and processes 2 and 5 continue in parallel. If process 5 then finishes also, it too is removed and process 3 is no longer suspended. Since there are no more calls for process 3 to execute, it immediately finishes, leaving process 2 to execute alone. Finally, when process 2 finishes, it is removed and process 1 resumes, whereupon it begins to execute the D call.

As an illustration of the parallel control rule, example 2 is one possible program to solve the problem analyzed by Dijkstra [8] who attributes it to R.W. Hamming. This problem is "to generate in increasing order the sequence of all numbers divisible by no primes other than 2, 3 or 5". That is, to generate the set

$$\{x: x = 2^i 3^j 5^k \;\&\; i \geqslant 0 \;\&\; j \geqslant 0 \;\&\; k \geqslant 0\}$$

in ascending order. The following solution happens not to remove duplicates but could easily be extended to do so. It is a good example of the potential clarity of logic programs (c.f. Dijkstra's solution in his imperative language).

Example 2

```
generate(1.x) <- multlist(2,1.x,r) // multlist(3,1.x,s) //
                 multlist(5,1.x,t) // merge(r!,s!,t!,x)

multlist(n,u.x,v.y) <- TIMES(n,u,v) & multlist(n,x!,y)

merge(u.x,v.y,w.z,u.t) <- u ⩽ v & u ⩽ w & merge(x!,v.y,w.z,t)
merge(u.x,v.y,w.z,v.t) <- v ⩽ u & v ⩽ w & merge(u.x,y!,w.z,t)
merge(u.x,v.y,w.z,w.t) <- w ⩽ u & w ⩽ v & merge(u.x,v.y,z!,t)
```

Here, generate(x) holds when x is the list representing the required set. The computation comprises four processes throughout: three which are continuously running multlist and one executing merge. Each multlist process multiplies as much of its incoming list as is instantiated and passes this data to the merge process. merge runs as soon as all three incoming lists are instantiated, and the resulting merged list is fed back to each of the other processes.

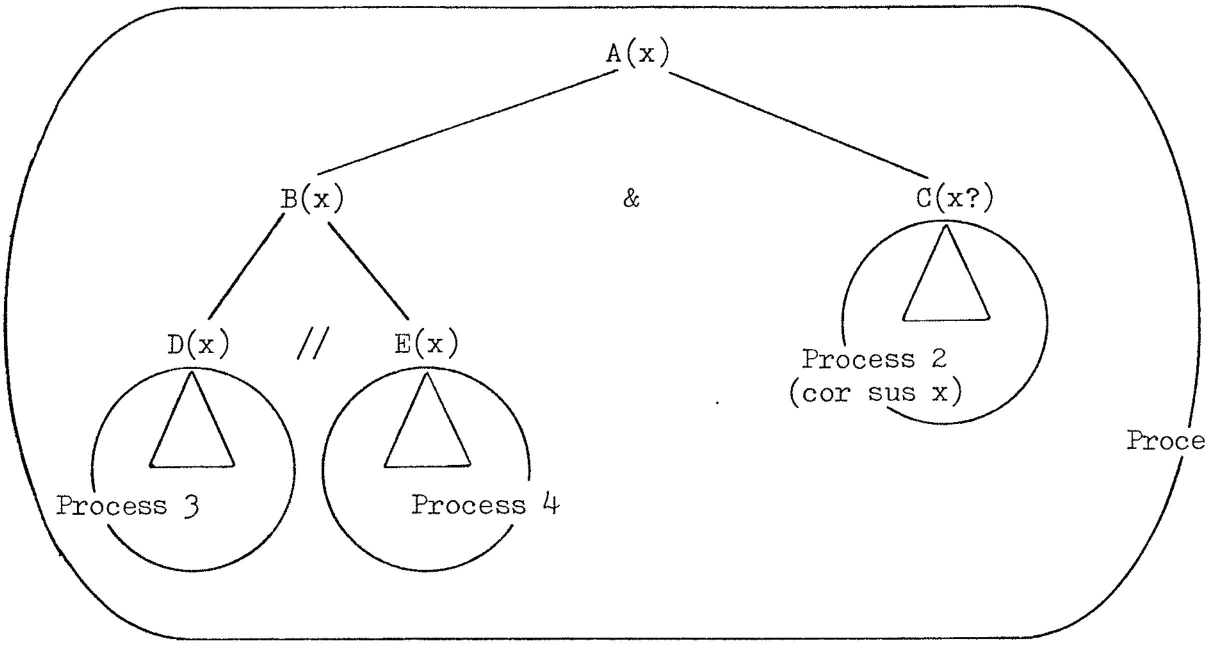
Combining parallelism and coroutining

We can obtain a fairly powerful control strategy by combining coroutining and parallelism. This requires a generalization of the two mechanisms described so far.

We distinguish between **coroutine processes** and **parallel processes**, according to the circumstances in which a process is created. We also need the concept of the **current process set**, which contains a number of processes being executed concurrently. Let us consider a simple example:

```
A(x) <- B(x) & C(x?)
B(x) <- D(x) // E(x)
C(x) <- F(x) // G(x)
```

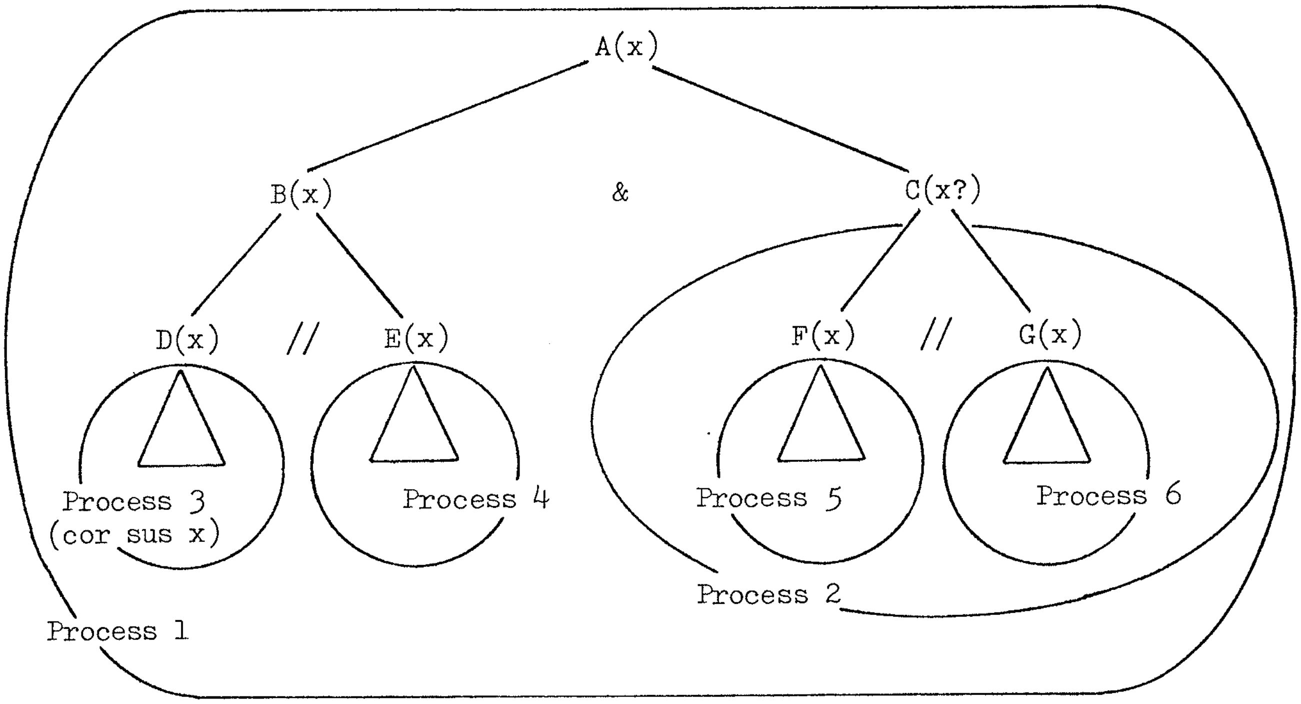
After the B call has executed one step, the situation is:



Apart from the main process (1), there are two parallel processes (3 and 4) and one coroutine process (2). Processes 1, 3 and 4 are in the current process set, while process 2 is **coroutine suspended** on the variable x - since x is the annotated variable at the root of its subtree. Process 1 is **temporarily suspended** since it is waiting for two subsidiary processes to finish. A process may also become temporarily suspended owing to the "!" annotation as described earlier. Processes which are temporarily suspended remain in the current process set; those which are coroutine suspended do not. Deadlock occurs if all processes in the current process set are temporarily suspended.

Now, since we have parallelism, for any annotated variable there may be a number of processes which are consumers and a number which are producers. This slightly complicates the earlier method of coroutine jumps and returns. Formerly, the single consumer and single producer of each annotated variable took turns to be suspended, but how do we perform jumps and returns now?

Returning to the example, suppose that process 3 binds x. We know that this is a producer of x because it is not descended from a call containing "x?". (A process would also be a producer of x if it were descended from a call containing "x↑".) Therefore we do not need to undo the binding. Process 3 then becomes coroutine suspended on x, and since there is a consumer (process 2) already suspended thus, the latter is resumed, i.e. brought into the current process set. If process 2 now executes one step, we have:



Now, processes 1 and 2 are temporarily suspended (waiting for subsidiary processes to finish), processes 4, 5 and 6 are executing, and process 3 is coroutine suspended on x. Suppose now that process 5 tries to bind x. We find that this process is a consumer (since it is descended from C(x?)) so the binding is undone and process 5 becomes coroutine suspended on x. Simultaneously process 3 is resumed since it is a producer suspended on x. If process 6, another consumer, now also tries to bind x, the binding is again undone and the process joins process 5 in being suspended on x. No producer is resumed this time.

This continues until one of the processes finishes. If the last of a number of sibling parallel processes finishes the parent will be released from its temporary suspension. If a coroutine process finishes, a "forced return" is effected and the process removed. That is, if a consumer, any producers suspended on its variable are resumed, and vice versa.

In general, then, the rules governing coroutining interaction are as follows:

When a variable x is bound to a non-variable term, determine first whether x is annotated in some call. If so, determine whether the current process is a consumer or a producer of x, according to whether or not it is descended from the annotated call. If there are processes of the opposite type coroutine suspended on x, resume them. Then suspend the current process on x, joining any others of the same type that may already be there.

It will be seen that a jump or return is no longer necessarily a single event, but consists of several acts of suspension and resumption. There may be a number of consumers and a number of producers in the current process set, but notice that at any given time, all processes of one type are current while one or more processes of the other type are suspended. It should also be pointed out that the mechanism described is not entirely symmetrical: if a number of producers become suspended in succession, they may produce more data than is required by the consumers.

Finally, it is clear that in the absence either of coroutining or of parallelism this method reduces to those described earlier for each strategy.

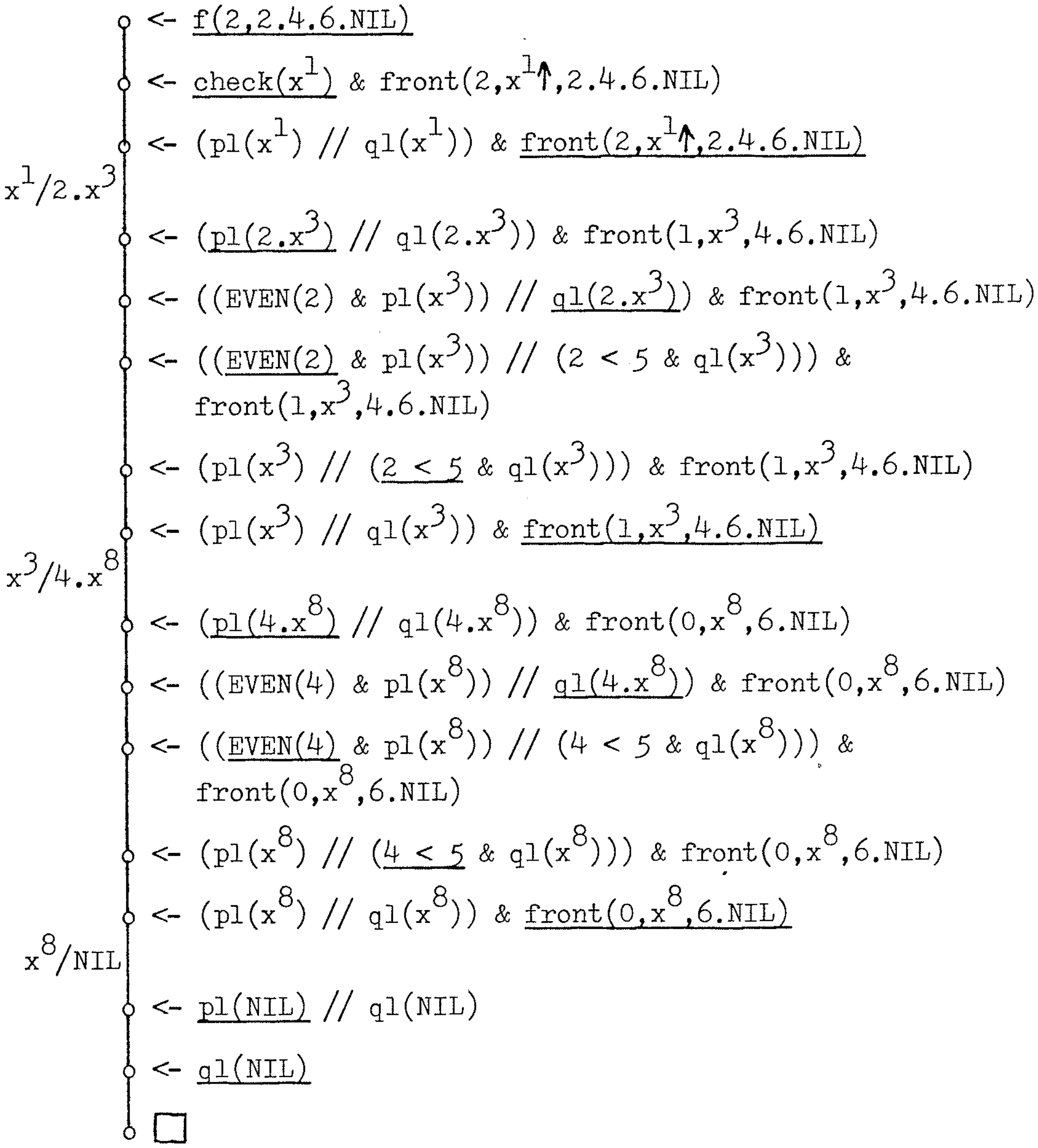
Let us now see a real example using coroutining and parallelism. Example 3 illustrates a common class of programs in which a consumer performs a number of operations in parallel upon each item of data supplied by the producer.

Example 3

```
f(n,z) <- check(x) & front(n,x↑,z)

check(x) <- pl(x) // ql(x)

pl(NIL) <-
pl(u.x) <- p(u) & pl(x)

ql(NIL) <-
ql(u.x) <- q(u) & ql(x)

front(0,NIL,z) <-
front(s(n),u.x,u.z) <- front(n,x,z)
```

f(n,z) holds when both p and q are true of each of the first n elements of the list z. The test is performed by running, in parallel, processes to test the list for each property.

Programs of this kind could not be written using only the coroutining computation rule. If the "//" were a "&", only the first call pl(x) would be executed each time before resuming the producer, leaving the second call ql(x) undone. To show how this scheme works, we exhibit the search tree for the goal <- f(2,2.4.6.NIL) and take p(u) and q(u) to be EVEN(u) and $u < 5$ respectively.



$\leftarrow \underline{f(2,2.4.6.NIL)}$

$\leftarrow \underline{check(x^1)}\ \&\ front(2,x^1\uparrow,2.4.6.NIL)$

$\leftarrow (pl(x^1)\ //\ ql(x^1))\ \&\ \underline{front(2,x^1\uparrow,2.4.6.NIL)}$

$x^1/2.x^3$

$\leftarrow (\underline{pl(2.x^3)}\ //\ ql(2.x^3))\ \&\ front(1,x^3,4.6.NIL)$

$\leftarrow ((EVEN(2)\ \&\ pl(x^3))\ //\ \underline{ql(2.x^3)})\ \&\ front(1,x^3,4.6.NIL)$

$\leftarrow ((\underline{EVEN(2)}\ \&\ pl(x^3))\ //\ (2 < 5\ \&\ ql(x^3)))\ \&\ front(1,x^3,4.6.NIL)$

$\leftarrow (pl(x^3)\ //\ (\underline{2 < 5}\ \&\ ql(x^3)))\ \&\ front(1,x^3,4.6.NIL)$

$\leftarrow (pl(x^3)\ //\ ql(x^3))\ \&\ \underline{front(1,x^3,4.6.NIL)}$

$x^3/4.x^8$

$\leftarrow (\underline{pl(4.x^8)}\ //\ ql(4.x^8))\ \&\ front(0,x^8,6.NIL)$

$\leftarrow ((EVEN(4)\ \&\ pl(x^8))\ //\ \underline{ql(4.x^8)})\ \&\ front(0,x^8,6.NIL)$

$\leftarrow ((\underline{EVEN(4)}\ \&\ pl(x^8))\ //\ (4 < 5\ \&\ ql(x^8)))\ \&\ front(0,x^8,6.NIL)$

$\leftarrow (pl(x^8)\ //\ (\underline{4 < 5}\ \&\ ql(x^8)))\ \&\ front(0,x^8,6.NIL)$

$\leftarrow (pl(x^8)\ //\ ql(x^8))\ \&\ \underline{front(0,x^8,6.NIL)}$

x^8/NIL

$\leftarrow \underline{pl(NIL)}\ //\ ql(NIL)$

$\leftarrow \underline{ql(NIL)}$

□

1.5 Control alternatives

IC-PROLOG allows the computation rule within a procedure to be determined by its use, i.e. the input/output pattern of arguments in the call. This may be done by supplying a list of **control alternatives**:

$$[P_1,\ldots,P_n]$$

These alternatives should all be copies of a procedure P, to which the whole list is equivalent declaratively. They differ only in the annotations and ordering of calls in the body, and in their **head annotations**. The head annotations specify which alternative should be used in a particular case, depending upon "?" and "↑" annotations attached to terms in the procedure heads. The exact use is explained in [3,5] but essentially a "?"-annotated head term means that the term should be instantiated, whereas a "↑"-annotated head term should not, in order for the alternative to be used.

It is interesting to consider the effect of using different logic in the alternatives, so that the logic depends upon use. This can be dangerous if used in combination with negation as failure and therefore is not permitted by IC-PROLOG. Otherwise, however, such a program can be read declaratively as though the head annotations were absent; these having the pragmatic effect of restricting the search according to use. We shall see in section 1.6 that this can be useful.

1.6 Processes as data structures

Hoare [12] gives an interesting algorithm for representing a set of integers, in his notation for Communicating Sequential Processes. There is an array of processes, each of which contains at most one integer, and these are arranged in ascending order. The set operations "insert" and "has" are performed by messages sent from the calling process to the first process of the array. Each process passes on the message to the next until the appropriate process is reached. An "insert" message received by a process will cause that process to adopt the inserted integer and the rest of the set will be shifted one place along the array of processes. A "has" enquiry received by an appropriate process will send a "true" or "false" response to the calling process. It is not clear how "delete" would be implemented in Hoare's algorithm.

Now that we have the concept of processes in PROLOG, can we reproduce the above behaviour in our formalism?

Suppose that the "set" data type is required, having the following operations:

insert: integer x set -> set.
Inserts the integer into the set, having no effect if already present.

delete: integer x set -> set.
Removes the integer from the set, having no effect if already absent.

has: integer x set -> boolean.
Finds whether or not the integer is in the set.

Suppose further that the set is to be represented by an ordered list (like Hoare's). Example 4 shows the conventional PROLOG program to implement this data type.

Example 4

```
insert(m,NIL,m.NIL) <-
insert(n,n.x,n.x) <-
insert(m,n.x,n.y) <- n < m & insert(m,x,y)
insert(m,n.x,m.n.x) <- m < n

delete(m,NIL,NIL) <-
delete(n,n.x,x) <-
delete(m,n.x,n.y) <- n < m & delete(m,x,y)
delete(m,n.x,n.x) <- m < n

has(m,NIL,FALSE) <-
has(n,n.x,TRUE) <-
has(m,n.x,t) <- n < m & has(m,x,t)
has(m,n.x,FALSE) <- m < n
```

Here, the representation is a term (using NIL and the "." functor) and the operations are procedures. We can invert this solution so that the ordered list is represented by processes and the operations constitute a term (a list of commands). This program is shown in example 5.

Example 5

```
empty(insert(m).x,y) <- item(m,x,w) & empty(w?,y)
empty(delete(m).x,y) <- empty(x,y)
empty(has(m).x,FALSE.y) <- empty(x,y)
empty(TRUE.x,TRUE.y) <- empty(x,y)
empty(FALSE.x,FALSE.y) <- empty(x,y)

item(n,insert(n).x,y) <- item(n,x,y)
item(n,insert(m).x,insert(m).y) <- n < m & item(n,x,y)
item(n,insert(m).x,y) <- m < n & item(m,x,w) & item(n,w?,y)

item(n,delete(n).x,x) <-
item(n,delete(m).x,delete(m).y) <- n < m & item(n,x,y)
item(n,delete(m).x,y) <- m < n & item(n,x,y)
item(n,has(n).x,TRUE.y) <- item(n,x,y)
item(n,has(m).x,has(m).y) <- n < m & item(n,x,y)
item(n,has(m).x,FALSE.y) <- m < n & item(n,x,y)

item(n,TRUE.x,TRUE.y) <- item(n,x,y)
item(n,FALSE.x,FALSE.y) <- item(n,x,y)
```

When the goal <- empty(x,y) is executed, the list of commands x will carry out the set operations, producing y as a list of responses. For example, if x is

```
has(2).insert(2).insert(3).has(2).has(3).delete(3).has(3).x'
```

y will be

```
FALSE.TRUE.TRUE.FALSE.y'
```

Each element of the set is represented by a process which is executing an item call. They communicate by arguments of which one process is an eager consumer of the previous process's output. Both commands and responses are passed along the sequence of processes via these shared arguments. Suppose that the set contains 2, 3 and 5, then the situation is:

```
item(2,x,a) & item(3,a?,b) & item(5,b?,c) & empty(c?,y)
```

If x is instantiated to insert(4).x' then a will be bound to insert(4).a' , then b to insert(4).b' :

```
item(2,x',a') & item(3,a'?,b') & item(5,insert(4).b',c) &
empty(c?,y)
```

The item(5,...) process will now cause a process to be created for the new item:

```
item(2,x',a') & item(3,a'?,b') & item(4,b'?,d) & item(5,d?,c) &
empty(c?,y)
```

If x' is now instantiated to delete(3).x" then this command will be passed on by binding a' to delete(3).a" :

```
item(2,x",a") & item(3,delete(3).a",b') & item(4,b'?,d) &
item(5,d?,c) & empty(c?,y)
```

The process executing item(3,...) will now finish, binding b' to a" and hence deleting 3 from the set:

```
item(2,x",a") & item(4,a"?,d) & item(5,d?,c) & empty(c?,y)
```

The only difference in behaviour between examples 4 and 5 is that in example 4, a "has" enquiry proceeds along the list only until the required item is found. In example 5, the "has" command is passed along the list as far as the item concerned, then a "TRUE" response is passed along the rest of the list. Ideally, as soon as a response is generated, there should be no further execution. Hoare's solution sends the "TRUE" or "FALSE" response direct to the calling process as soon as it is generated.

To obtain an approximation to this more desirable behaviour in our formalism, we need the output list of responses to be available to all of the item calls. Then each process has the option of instantiating either this response list or the input list of the next process as before. To permit this, we need to use control alternatives having different logic so that a process can skip any bindings made to the response list by other processes. We end up with example 6, which will (currently) run on IC-PROLOG.

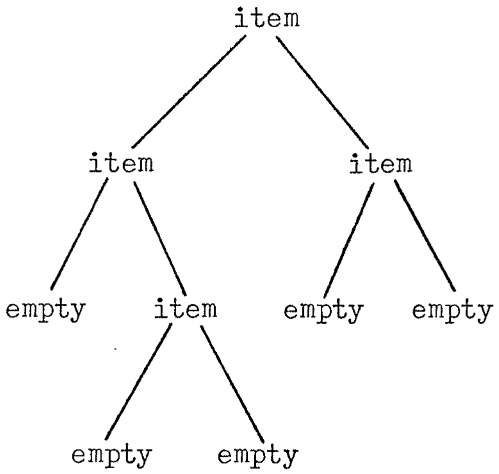
Example 6

```
[empty(insert(m).x,u↑.p) <- item(m,x,w,u.p) & empty(w?,u.p),
 empty(insert(m).x,u?.p) <- empty(insert(m).x,p)]

[empty(delete(m).x,u↑.p) <- empty(x,u.p),
 empty(delete(m).x,u?.p) <- empty(delete(m).x,p)]

[empty(has(m).x,u↑.p) <- u = FALSE & empty(x,p),
 empty(has(m).x,u?.p) <- empty(has(m).x,p)]

[item(n,insert(m).x,y,u↑.p) <-
     m = n
        THEN item(n,x,y,u.p)
        ELSE(n < m
           THEN(y = insert(m).z &
                 item(n,x,z,u.p))
           ELSE(item(m,x,w,u.p) &
                 item(n,w?,y,u.p))),
 item(n,insert(m).x,y,u?.p) <- item(n,insert(m).x,y,p)]
```

```
[item(n,delete(m).x,y,u↑.p) <-
    m = n
       THEN y = x
       ELSE(n < m
          THEN(y = delete(m).z &
               item(n,x,z,u.p))
          ELSE item(n,x,y,u.p)),
 item(n,delete(m).x,y,u?.p) <- item(n,delete(m).x,y,p)]

[item(n,has(m).x,y,u↑.p) <-
    m = n
       THEN(u = TRUE &
            item(n,x,y,p))
       ELSE(n < m
          THEN(y = has(m).z &
               item(n,x,z,u.p))
          ELSE(u = FALSE &
               item(n,x,y,p))),
 item(n,has(m).x,y,u?.p) <- item(n,has(m).x,y,p)]
```

Above, each procedure has an alternative for use when the response list is instantiated (u?.p), which skips the instantiated element u. When this argument is an unbound variable, the usual procedure is used.

Obviously, we are not restricted to lists; other data structures can equally be represented by processes. For example, we can represent a set by an ordered binary tree of processes:



providing each item with one input list and two output lists, one for each subtree, plus the response list:

```
item(n,x,l,r,p)
```

An item receives a command on its input list x and deals with it if possible. The process will either instantiate the response list p or pass on the command to its left or right subtree - according to the value - by instantiating l or r respectively. Again, empty has one input list x:

```
empty(x,p)
```

CHAPTER 2

TRANSFORMATION OF LOGIC PROGRAMS

Program transformation is essentially the process of converting an inefficient program into one which is more efficient but generally less clear. This is done in a systematic manner which preserves the supposed correctness of the original. Program synthesis is similar except that the original is a specification and not considered runnable.

After considering the origins of program synthesis and transformation, we shall investigate the transformation of Horn clause programs in some detail. We show how control annotations may be used to partially automate the transformation and suggest that this approaches the compilation of annotated logic programs.

Our objective shall be: given an algorithm A, expressed as a logic component L and a control component C (i.e. A = L + C), create a new logic component L', such that A = L' + (default control strategy).

2.1 Early work in program transformation and synthesis

We shall briefly consider two systems: that of Burstall and Darlington [1] and that of Manna and Waldinger [17,18]. Both of these derive recursive programs by the use of a recursion introduction or "folding" rule.

Burstall and Darlington's system

In this system, both the **specification language** and the **target language** are recursion equations. There are several rules for transforming a program in this language into a more efficient one. These are:

Definition.
Define a new recursion equation for an expression L, of the form L <= R.

Instantiation.
Create a substitution instance of an existing recursion equation.

Unfolding.

Replace an occurrence of an expression in the rhs of a recursion equation by the rhs of another equation with the first expression as its lhs. i.e. if L <= R and L' <= R' already exist, and L' occurs in R: replace L' in R by R' giving R", the new equation is L <= R" .

Folding.

Replace an occurrence of an expression in the rhs of a recursion equation by the lhs of another equation with the first expression as its rhs. i.e. if L <= R and L' <= R' already exist, and R' occurs in R: replace R' in R by L' giving R", the new equation is L <= R" .

Abstraction.

Replace parts of an expression, in the rhs of an equation, by variables. Define these variables in a **where** clause.
e.g. A <= b + b becomes A <= x + x **where** x = b .

Laws.

Apply rewriting rules to the rhs of an equation to express information about commutativity, associativity, etc.

The strategy used to transform programs is firstly to make definitions and instantiations. Then each instantiated lhs is repeatedly unfolded. Finally, laws and abstraction are applied and the equations repeatedly folded. Examples in [1] and [7] illustrate the use of these rules in "intertwining" programs. Often, programs comprising several recursions are transformed into equivalent programs with a single recursion, preferably linear. We will see examples of such transformations in the Horn clause formalism later in this chapter.

A semi-automatic system was implemented by Burstall and Darlington [1]. This accepted the original recursion equations including those specially defined, a set of instantiated left hand sides, and a set of rewriting rules. The system would search for all possible transformations of the specified lhs's and print the resulting equations.

More recently, Feather [9,10] has implemented a more sophisticated transformation system (ZAP), based on the above rules, which is designed to transform large programs. To this end, the system is claimed to embody a workable blend of automation and user guidance. The user provides the intuitive guidance at a high level while the system performs the more mundane tasks and also supplies defaults to aid the user.

Feather's system uses a "metaprogram" to control the transformation of a "protoprogram", the latter comprising the original recursion equations. The metaprogram consists of a sequence of special commands which considerably reduce the search required to find a transformation. The user supplies each instantiated lhs together with a pattern for the desired transformed equation. Commands are provided for specifying which functions are to be used for unfolding and which may occur in the transformed equations. A number of "default generators" are provided: these make use of user-supplied type information for such purposes as automatically generating the instantiated lhs's. To demonstrate the effectiveness of the system, it has been used to transform various large programs, such as a text formatter [11].

Manna and Waldinger's system

Manna and Waldinger [17,18] have developed a method for synthesizing recursive programs from specifications. They make use of a special purpose specificati on language, intended to facilitate the natural description of the problem, while their target language is a variant of Lisp. The techniques have been implemented in their SYNSYS system.

This system differs from that of Burstall and Darlington mainly in its specification language, which may be extended at will and adapted to suit a particular application. Transformation rules are supplied for each construct in the specification language, to transform it eventually into a "primitive program". The price paid for this rich, extendable specification language is the need for a large number of transformation rules; c.f. the six transformation rules of Burstall and Darlington.

Another feature of the SYNSYS system is that it is fully automatic and does not benefit from any guidance on the part of the user as do the other systems considered.

2.2 Standard form logic program derivation

Burstall and Darlington's transformation techniques soon inspired attempts [2,3,4,6,13] to synthesize Horn clause programs from specifications in the standard form of logic. The specification comprises a set of axioms defining the relation to be computed; logical deduction is used to derive a set of computationally useful Horn clauses. Recursions are introduced into the derived clauses by a variant of the folding rule.

Hogger [13] treats program construction as a goal-oriented derivation, beginning with a single "call" for which a procedure is sought. The derivation consists of a sequence of applications of rules, each of which is either a "goal simplification" or a "goal substitution" rule. The former rule causes the replacement of the current goal by one which it logically implies. A goal substitution rule incorporates new information into the current goal from the specification axioms by activating a "call". The final goal constitutes the body of the derived procedure, whose head is the current substitution instance of the initial goal. The resulting derivation resembles a conventional top-down logic program execution in many respects. However, a call need not be atomic, and the replacement of a call by a "body" is determined by one of several goal substitution rules. Hogger gives a number of inference rules, both for goal substitution and simplification, which have proved to be useful in the derivation of Horn clause programs.

Clark [2,3] also regards program derivation as the top-down symbolic execution of a standard form specification, though in a less formal manner; however, he makes greater use of terms in the derivation process.

The important properties of this kind of program derivation are the high level nature of the specification language and the fact that all derivation rules have a logical justification.

2.3 Horn clause program transformation

We now consider transformation in which both specification and target languages are Horn clauses. This is a special case of logic program derivation, as outlined in the previous section. The specification language is, of course, less powerful but the transformation rules are accordingly simpler. There is a close similarity between (Horn clause - Horn clause) transformation and (recursion equation - recursion equation) transformation; however, Horn clauses provide a slightly richer specification language than pure recursion equations.

Transformation rules

The four principle rules are described below:

Symbolic execution.

This corresponds directly to the unfolding rule for recursion equations. It is identical to the execution mechanism for Horn clause programs, except that "primitive" predicates are not normally executed. One call of the goal clause is selected and unified with the head of a procedure, which has no variables in common with the goal clause. The call is replaced by the procedure body and the unifying substitution applied. To ensure that variables are distinct, we superscript the variables of a procedure by the number of the level at which the procedure is invoked.

Laws.

As in recursion equations, we make use of lemmas to express information about certain predicates. Such a lemma is an equivalence of the form

$$\exists x_1 \ldots \exists x_m [C(x_1,\ldots,x_m,y_1,\ldots,y_p)] <->$$
$$\exists x_1' \ldots \exists x_n' [C'(x_1',\ldots,x_n',y_1,\ldots,y_p)]$$

where $C(x_1,\ldots,y_p)$ and $C'(x_1',\ldots,y_p)$ represent conjunctions and the x_is, x_i's and y_is denote variables. We use the equivalence to rearrange the current goal clause as follows. Suppose the initial goal atom was G_0, the body of the current goal clause is G_n and the current substitution is θ: the derived clause so far is $G_0\theta <- G_n$. If the conjunction G_n can be split into two parts G_{n1} and G_{n2}, and G_{n1} matches with $C(x_1,\ldots,x_m,y_1,\ldots,y_p)$ such that the x_is match variables in G_{n1} which do not appear either in G_{n2} or in $G_0\theta$, then we can replace G_{n1} in G_n by $C'(x_1',\ldots,x_n',y_1,\ldots,y_p)$, where $x_1',\ldots,x_n'$ are new variables, to give a new current goal clause with body G_n'.
e.g. to express the associativity of TIMES, use the equivalence

$$\exists x[\text{TIMES}(r,s,x) \ \& \ \text{TIMES}(x,t,u)] <->$$
$$\exists x'[\text{TIMES}(s,t,x') \ \& \ \text{TIMES}(r,x',u)]$$

The application of laws is easier in the recursion equation formalism since this embodies only functions; there is no need for existentially quantified variables for intermediate results. The above lemma would be expressed as

$$\text{TIMES}(\text{TIMES}(r,s),t) = \text{TIMES}(r,\text{TIMES}(s,t))$$

Functionality.

We can obtain the same effect as the abstraction rule of Burstall and Darlington by using information about the functionality of predicates to delete redundant atoms from the current goal clause. If we know that the predicate p is a function of its first m arguments, we have the equivalence

$$p(t_1,\ldots,t_m,r_1,\ldots,r_n) \;\&\; p(t_1,\ldots,t_m,s_1,\ldots,s_n) <->$$
$$p(t_1,\ldots,t_m,r_1,\ldots,r_n) \;\&\; r_1 = s_1 \;\&\; \ldots \;\&\; r_n = s_n$$

Using this equivalence, if we find two atoms $p(t_1,\ldots,t_m,r_1,\ldots,r_n)$ and $p(t_1,\ldots,t_m,s_1,\ldots s_n)$ in the current goal clause, we can delete the second atom and unify the pairs of terms $<r_1,s_1>$, ..., $<r_n,s_n>$.

Folding.

This is analogous to Burstall and Darlington's rule for recursion equations. Suppose that one of the procedures for predicate r is

$$r(t_1,\ldots,t_n) <- C(x_1,\ldots,x_m,y_1,\ldots,y_p)$$

where $y_1,\ldots,y_p$ are all of the variables which occur in the terms $t_1,\ldots,t_n$, and $x_1,\ldots,x_m$ are the variables which do not; $C(x_1,\ldots,y_p)$ represents a conjunction. If this is the only procedure for r or if all procedures for r treat disjoint tuples of head arguments, then we have the equivalence

$$r(t_1,\ldots,t_n) <-> \exists x_1 \ldots \exists x_m[C(x_1,\ldots,x_m,y_1,\ldots,y_p)]$$

We search the current goal clause in the same manner as described for "laws", searching for a part which unifies with $C(x_1,\ldots,x_m,y_1,\ldots,y_p)$ such that $x_1,\ldots,x_m$ match variables which do not appear elsewhere in the current goal clause or in the current substitution instance of the initial goal atom. If such a part is found, we replace it by the atom $r(t_1,\ldots,t_n)$ and apply the unifying substitution.

This rule is more rigorously defined in [3].

Transformation strategy

In general, we adopt the same strategy for applying the rules as that suggested for recursion equations [1]:

1) The user specifies the program procedures including any specially defined procedures.

2) The user specifies a number of suitably instantiated goal atoms.

3) The goal atoms are each symbolically executed repeatedly, using an arbitrary computation rule.

4) The symbolic execution is frozen.

5) Laws are optionally applied.

6) Functionality simplification is optionally performed.

7) Folding is performed, possibly repeatedly, until a recursion is obtained.

8) The try order of calls in the derived procedures is rearranged, and the procedures ordered for computational use.

The intelligence of the user may be required in each of these steps.

Some examples

We consider first an example adapted from [7] in which a doubly recursive program is transformed into a linear recursive program. The procedure to be transformed is for flatten, where flatten(t,l) holds when l is a list of the frontier of t, an unlabelled binary tree. Example 7 gives the simple program for flatten.

Example 7

```
        flatten(tip(u),u.NIL) <-
        flatten(tree(s,t),z) <- flatten(s,x) & flatten(t,y) & append(x,y,z)

        append(NIL,y,y) <-
        append(u.x,y,u.z) <- append(x,y,z)
```

We follow Darlington and Waldinger and consider three cases for the structure of a tree: 1) tip(u), 2) tree(tip(u),t), 3) tree(tree(r,s),t) . For the first case, a procedure already exists. For each of the other two cases, we provide an instantiated goal atom and symbolically execute each. At the end of the second derivation, we apply the law of associativity of append and then fold twice with flatten. The law is expressed as

$\exists$x[append(r,s,x) & append(x,t,u)] <->
$\exists$x'[append(s,t,x') & append(r,x',u)]

The derivation proceeds thus:

○ <- $\underline{flatten(tree(tip(u^0),t^0),z^0)}$

○ <- $\underline{flatten(tip(u^0),x^1)}$ & $flatten(t^0,y^1)$ & $append(x^1,y^1,z^0)$

$x^1/u^0.NIL$

○ <- $flatten(t^0,y^1)$ & $\underline{append(u^0.NIL,y^1,z^0)}$

$z^0/u^0.z^3$

○ <- $flatten(t^0,y^1)$ & $\underline{append(NIL,y^1,z^3)}$

y^1/z^3

○ <- $flatten(t^0,z^3)$

○ <- $\underline{flatten(tree(tree(r^0,s^0),t^0),z^0)}$

○ <- $\underline{flatten(tree(r^0,s^0),x^1)}$ & $flatten(t^0,y^1)$ &
$append(x^1,y^1,z^0)$

○ <- $flatten(r^0,x^2)$ & $flatten(s^0,y^2)$ & $\underline{append(x^2,y^2,x^1)}$ &
$flatten(t^0,y^1)$ & $\underline{append(x^1,y^1,z^0)}$

Laws

○ <- $\underline{append(y^2,y^1,x^3)}$ & $append(x^2,x^3,z^0)$ &
$flatten(r^0,x^2)$ & $\underline{flatten(s^0,y^2)}$ & $\underline{flatten(t^0,y^1)}$

Fold flatten

○ <- $\underline{flatten(tree(s^0,t^0),x^3)}$ & $\underline{append(x^2,x^3,z^0)}$ &
$\underline{flatten(r^0,x^2)}$

Fold flatten

○ <- $flatten(tree(r^0,tree(s^0,t^0)),z^0)$

We end up with the three procedures

```
flatten(tip(u),u.NIL) <-
flatten(tree(tip(u),t),u.z) <- flatten(t,z)
flatten(tree(tree(r,s),t),z) <- flatten(tree(r,tree(s,t)),z)
```

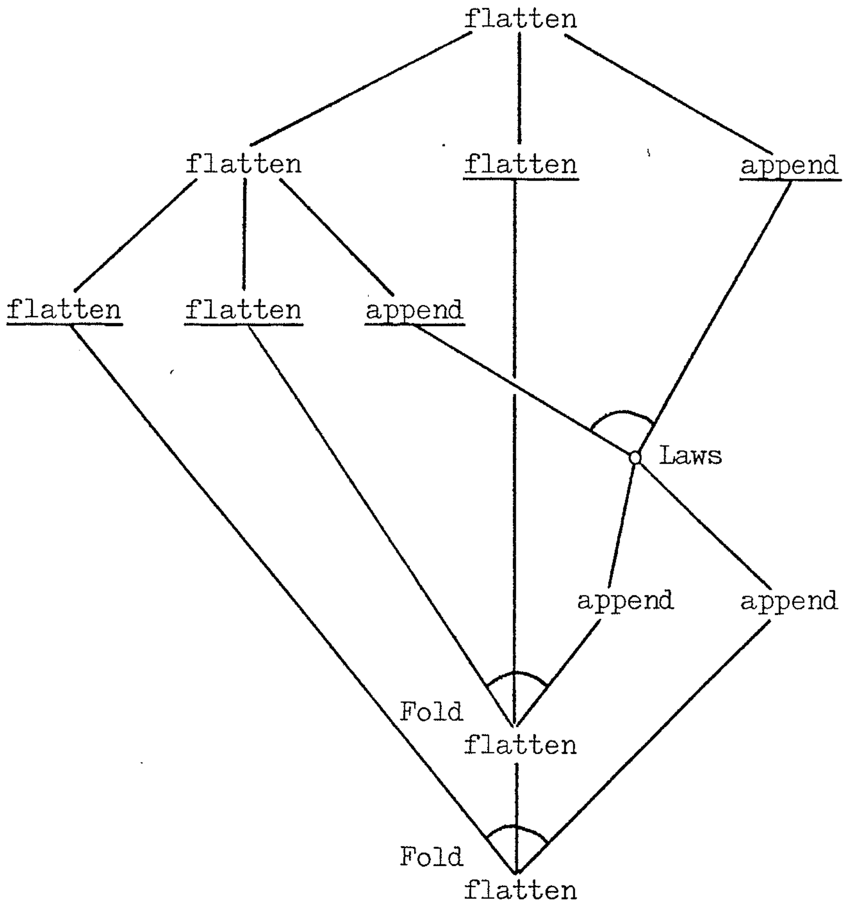
after renaming the superscripted variables to improve appearance.

We are justified in performing the two folds with flatten, since the original procedures provide the equivalences

```
flatten(tip(u),z) <-> z = u.NIL
flatten(tree(s,t),z) <-> ∃x∃y[flatten(s,x) & flatten(t,y) &
                               append(x,y,z)]
```

Our fold steps employ the second of these.

The second transformation above can be shown diagrammatically as follows. The tips of the fully constructed proof tree are underlined:

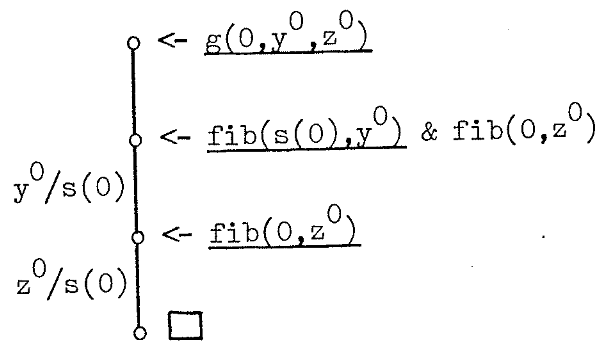


Let us now consider another example, to generate the Fibonacci series, adapted from [1]. This time we need to supply an auxiliary definition, of the relation g.

Example 8

```
fib(0,s(0)) <-
fib(s(0),s(0)) <-
fib(s(s(x)),y) <- fib(s(x),u) & fib(x,v) & PLUS(u,v,y)

g(x,y,z) <- fib(s(x),y) & fib(x,z)
```

Like Burstall and Darlington, we deal with two cases and derive a procedure for each:



$\leftarrow \underline{g(0,y^0,z^0)}$

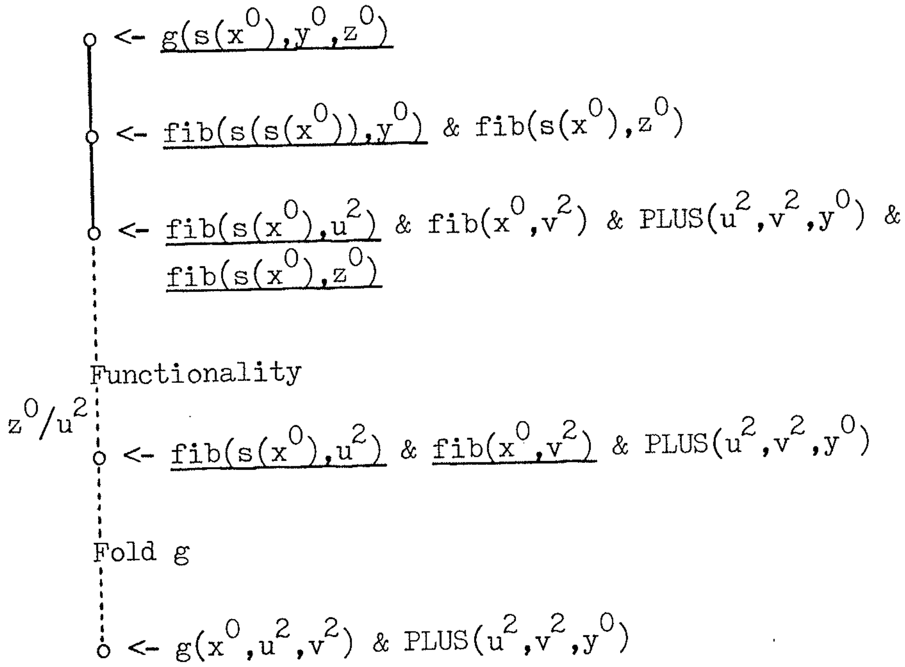
$\leftarrow \underline{fib(s(0),y^0)}$ & $fib(0,z^0)$

$y^0/s(0)$

$\leftarrow \underline{fib(0,z^0)}$

$z^0/s(0)$

□



$\leftarrow \underline{g(s(x^0),y^0,z^0)}$

$\leftarrow \underline{fib(s(s(x^0)),y^0)}$ & $fib(s(x^0),z^0)$

$\leftarrow \underline{fib(s(x^0),u^2)}$ & $fib(x^0,v^2)$ & $PLUS(u^2,v^2,y^0)$ & $\underline{fib(s(x^0),z^0)}$

Functionality

z^0/u^2

$\leftarrow \underline{fib(s(x^0),u^2)}$ & $\underline{fib(x^0,v^2)}$ & $PLUS(u^2,v^2,y^0)$

Fold g

$\leftarrow g(x^0,u^2,v^2)$ & $PLUS(u^2,v^2,y^0)$

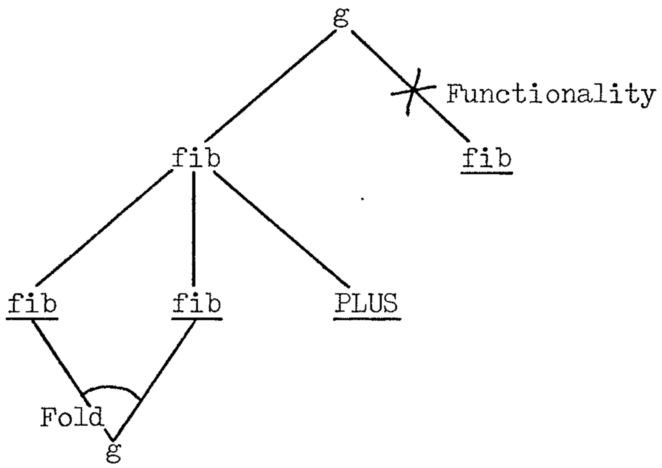
The resulting procedures, after renaming superscripted variables, are

```
g(0,s(0),s(0)) <-
g(s(x),y,u) <- g(x,u,v) & PLUS(u,v,y)
```

To compute fib using g, we need the procedures

```
fib(0,s(0)) <-
fib(s(0),s(0)) <-
fib(s(s(x)),y) <- g(x,u,v) & PLUS(u,v,y)
```

In the third step of the second derivation above, we have made use of the fact that fib is a function of its first argument, in order to delete a redundant atom. The effect upon the proof tree is thus:



A common objective of all of our transformations is to construct each proof tree in such a manner as to find a pattern which can be folded with the goal predicate. In general, to do this we may need to execute each call to a different depth. We may also need to rearrange the proof tree, as we have seen, by "laws" and "functionality" in order to find a suitable pattern.

2.4 Automatic branching

In simple cases, we can relieve the user of the need to provide instantiated goal atoms. Instead, we begin each derivation with a goal atom containing only variables and construct the search tree from this. Each branch of the search tree leads either to a failure node or to a derived procedure. An assertion is derived for each success node, while a recursive procedure is obtained by freezing the construction of each infinite branch of the search tree.

This automatic generation of "cases" is reminiscent of the default generators of Feather's ZAP transformation system [9,10].

To illustrate the method, we again use an example adapted from Burstall and Darlington [1]. Example 9 is a simple program for computing the sum of the scalar products of two pairs of vectors, each represented by a list of integers. f(w,x,y,z,n) holds when

$$n = w \cdot x + y \cdot z$$

Example 9

```
f(w,x,y,z,n) <- samelength(w,x,y,z) & dot(w,x,l) & dot(y,z,m) &
                PLUS(l,m,n)

samelength(NIL,NIL,NIL,NIL) <-
samelength(s.w,t.x,u.y,v.z) <- samelength(w,x,y,z)

dot(NIL,NIL,0) <-
dot(u.y,v.z,n) <- TIMES(u,v,l) & dot(y,z,m) & PLUS(l,m,n)
```

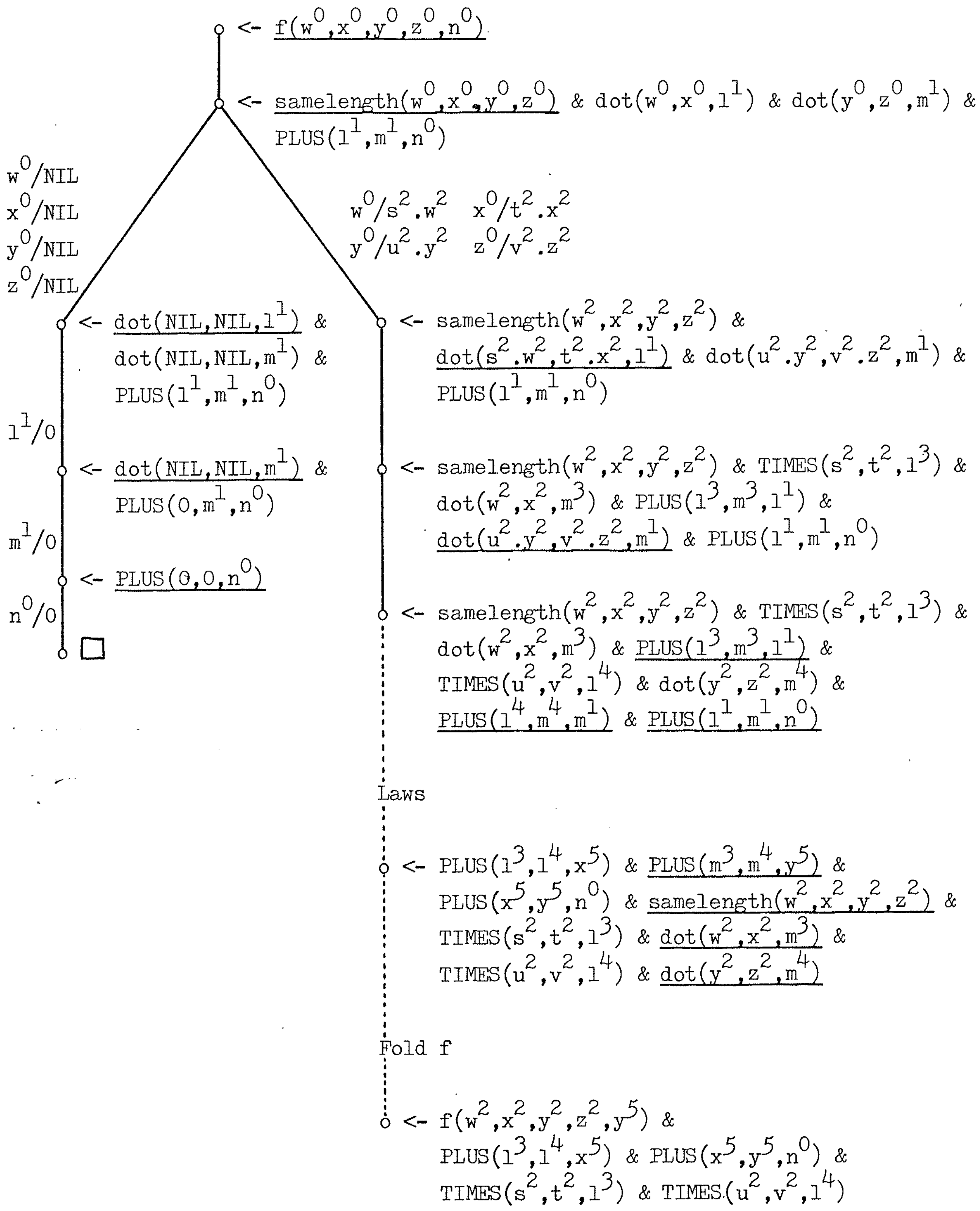
We will need to use a property of addition, namely that

$$(r + s) + (t + u) = (r + t) + (s + u)$$

This is expressed as

```
∃x∃y[PLUS(r,s,x) & PLUS(t,u,y) & PLUS(x,y,v)] <->
∃x'∃y'[PLUS(r,t,x') & PLUS(s,u,y') & PLUS(x',y',v)]
```

The following search tree shows the derivation of new procedures for f:



$\leftarrow \underline{f(w^0,x^0,y^0,z^0,n^0)}$
$\leftarrow \underline{samelength(w^0,x^0,y^0,z^0)}$ & $dot(w^0,x^0,l^1)$ & $dot(y^0,z^0,m^1)$ & $PLUS(l^1,m^1,n^0)$
w^0/NIL
x^0/NIL
y^0/NIL
z^0/NIL
$w^0/s^2.w^2$ $x^0/t^2.x^2$
$y^0/u^2.y^2$ $z^0/v^2.z^2$
$\leftarrow \underline{dot(NIL,NIL,l^1)}$ & $dot(NIL,NIL,m^1)$ & $PLUS(l^1,m^1,n^0)$
$l^1/0$
$\leftarrow \underline{dot(NIL,NIL,m^1)}$ & $PLUS(0,m^1,n^0)$
$m^1/0$
$\leftarrow \underline{PLUS(0,0,n^0)}$
$n^0/0$
□
$\leftarrow samelength(w^2,x^2,y^2,z^2)$ & $\underline{dot(s^2.w^2,t^2.x^2,l^1)}$ & $dot(u^2.y^2,v^2.z^2,m^1)$ & $PLUS(l^1,m^1,n^0)$
$\leftarrow samelength(w^2,x^2,y^2,z^2)$ & $TIMES(s^2,t^2,l^3)$ & $dot(w^2,x^2,m^3)$ & $PLUS(l^3,m^3,l^1)$ & $\underline{dot(u^2.y^2,v^2.z^2,m^1)}$ & $PLUS(l^1,m^1,n^0)$
$\leftarrow samelength(w^2,x^2,y^2,z^2)$ & $TIMES(s^2,t^2,l^3)$ & $dot(w^2,x^2,m^3)$ & $\underline{PLUS(l^3,m^3,l^1)}$ & $TIMES(u^2,v^2,l^4)$ & $dot(y^2,z^2,m^4)$ & $\underline{PLUS(l^4,m^4,m^1)}$ & $\underline{PLUS(l^1,m^1,n^0)}$
Laws
$\leftarrow PLUS(l^3,l^4,x^5)$ & $\underline{PLUS(m^3,m^4,y^5)}$ & $PLUS(x^5,y^5,n^0)$ & $\underline{samelength(w^2,x^2,y^2,z^2)}$ & $TIMES(s^2,t^2,l^3)$ & $\underline{dot(w^2,x^2,m^3)}$ & $TIMES(u^2,v^2,l^4)$ & $\underline{dot(y^2,z^2,m^4)}$
Fold f
$\leftarrow f(w^2,x^2,y^2,z^2,y^5)$ & $PLUS(l^3,l^4,x^5)$ & $PLUS(x^5,y^5,n^0)$ & $TIMES(s^2,t^2,l^3)$ & $TIMES(u^2,v^2,l^4)$

The first branch of the search tree leads to a base procedure while the second branch is frozen and folded to give a recursive procedure. The derived procedures are

```
f(NIL,NIL,NIL,NIL,0) <-
f(s.w,t.x,u.y,v.z,n) <- TIMES(s,t,l) & TIMES(u,v,m) & PLUS(l,m,p) &
                        f(w,x,y,z,q) & PLUS(p,q,n)
```

2.5 Annotations to control symbolic execution

Example 9 illustrated a common effect of transformation: that formerly separate parts of a computation become "intertwined" in the transformed procedures. But this is exactly the effect obtained when the original procedures are executed using a coroutining or parallel computation rule.

The intertwined nature of the transformed procedures comes about because we alternate between branches of the proof tree during the symbolic execution phase. If we include control annotations in the original program and use these to control the symbolic execution, then we will have automated one further part of the transformation. The user now only needs to provide guidance in the following areas:

a) Specifying auxiliary relations.

b) Deciding when to freeze symbolic execution.

c) Supplying information about laws and functionality.

d) Guiding the search for a fold.

e) Rearranging and possibly restructuring the transformed procedures for computational use.

In section 2.6 we shall see several examples of the use of coroutining annotations in transformation, while in section 2.7 we shall consider parallel annotations.

2.6 Coroutining in transformation

None of the remaining examples in this chapter requires the application of laws or functionality; each consists simply of symbolic execution followed by folding. We begin with an annotated program and "compile" this into a sequential program which has the same "intertwined" behaviour.

We begin with a simple example. Example 10 is a program to compute the sum of squares of the integers in a list.

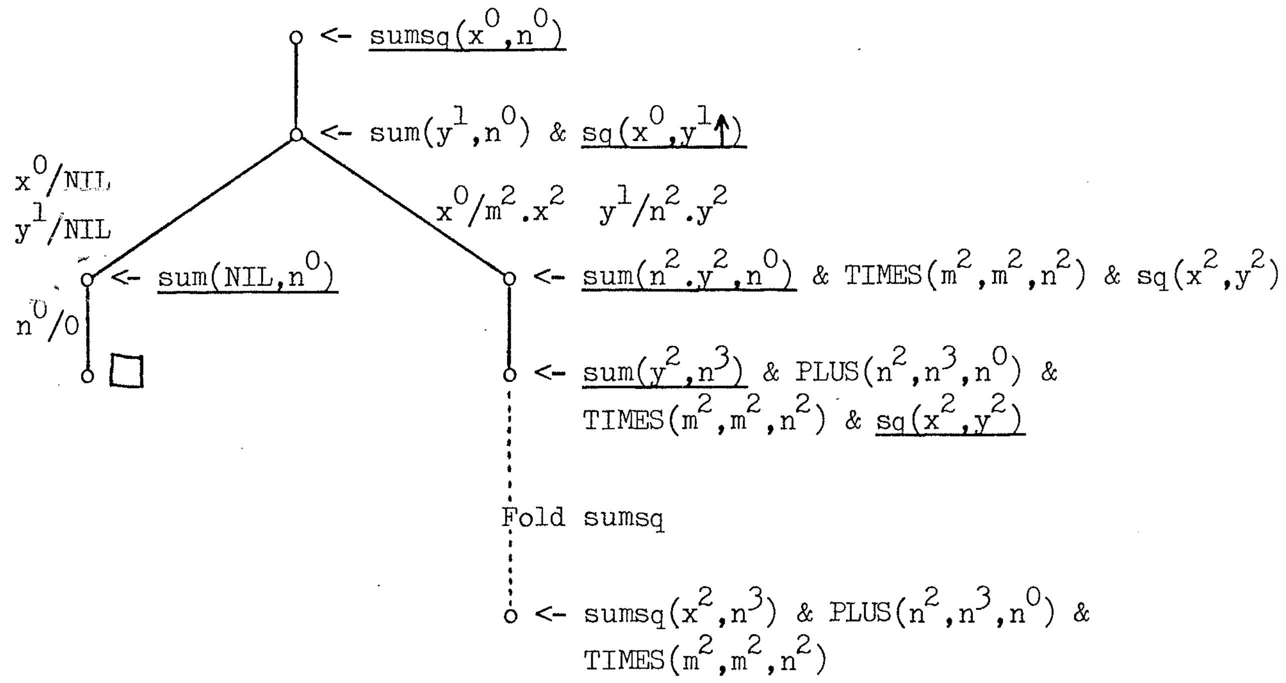
Example 10

```
sumsq(x,n) <- sum(y,n) & sq(x,y↑)

sum(NIL,0) <-
sum(m.x,p) <- sum(x,n) & PLUS(m,n,p)

sq(NIL,NIL) <-
sq(m.x,n.y) <- TIMES(m,m,n) : sq(x,y)
```

Here, sumsq(x,n) holds when

$$n = \sum_{k \text{ in } x} k^2$$

In the search tree for this transformation, the clause bar is treated as a normal "&". Note also that the "primitive" calls - in this case those for PLUS and TIMES - are not executed.



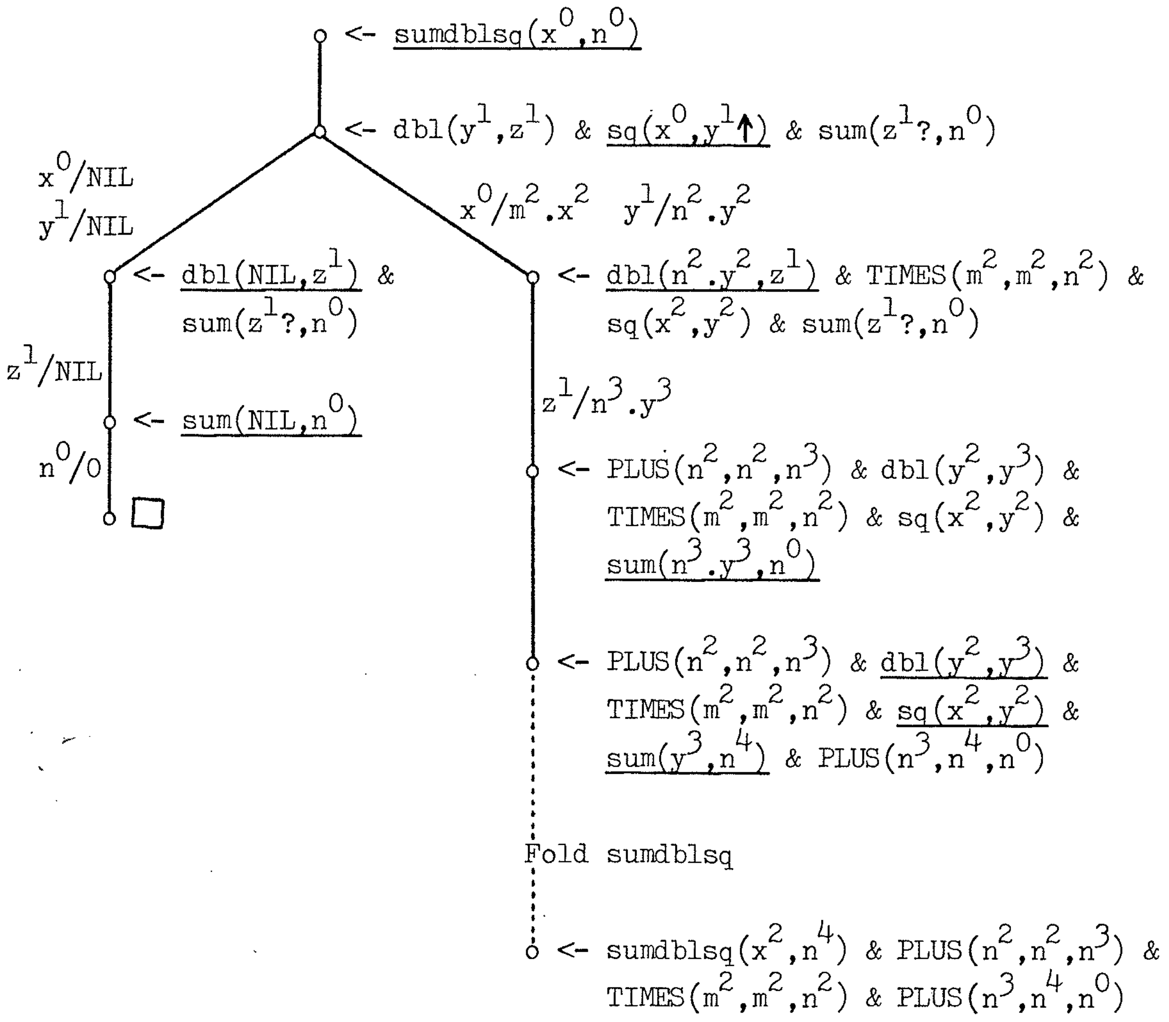
<- $\underline{sumsq(x^0,n^0)}$
<- $sum(y^1,n^0)$ & $\underline{sq(x^0,y^1\uparrow)}$
x^0/NIL
y^1/NIL
$x^0/m^2.x^2$ $y^1/n^2.y^2$
<- $\underline{sum(NIL,n^0)}$
n^0/0
□
<- $\underline{sum(n^2.y^2,n^0)}$ & $TIMES(m^2,m^2,n^2)$ & $sq(x^2,y^2)$
<- $\underline{sum(y^2,n^3)}$ & $PLUS(n^2,n^3,n^0)$ & $TIMES(m^2,m^2,n^2)$ & $\underline{sq(x^2,y^2)}$
Fold sumsq
<- $sumsq(x^2,n^3)$ & $PLUS(n^2,n^3,n^0)$ & $TIMES(m^2,m^2,n^2)$

The resulting procedures are

```
sumsq(NIL,0) <-
sumsq(m.x,n) <- TIMES(m,m,p) & sumsq(x,q) & PLUS(p,q,n)
```

It is a general principle of transformation that the clause bar should be ignored, since it obstructs the required alternation between branches of the proof tree. This makes no difference in the example above but is significant in more complex transformations. For example, Clark's program [3] for the eight queens problem contains a clause bar which is ignored in his transformation.

We now transform the program of example 1.



$\leftarrow \underline{sumdblsq(x^0,n^0)}$
$\leftarrow dbl(y^1,z^1)\ \&\ \underline{sq(x^0,y^1\uparrow)}\ \&\ sum(z^1?,n^0)$
x^0/NIL
y^1/NIL
$x^0/m^2.x^2\quad y^1/n^2.y^2$
$\leftarrow \underline{dbl(NIL,z^1)}\ \&\ sum(z^1?,n^0)$
$\leftarrow \underline{dbl(n^2.y^2,z^1)}\ \&\ TIMES(m^2,m^2,n^2)\ \&\ sq(x^2,y^2)\ \&\ sum(z^1?,n^0)$
z^1/NIL
$\leftarrow \underline{sum(NIL,n^0)}$
$n^0/0$
□
$z^1/n^3.y^3$
$\leftarrow PLUS(n^2,n^2,n^3)\ \&\ dbl(y^2,y^3)\ \&\ TIMES(m^2,m^2,n^2)\ \&\ sq(x^2,y^2)\ \&\ \underline{sum(n^3.y^3,n^0)}$
$\leftarrow PLUS(n^2,n^2,n^3)\ \&\ \underline{dbl(y^2,y^3)}\ \&\ TIMES(m^2,m^2,n^2)\ \&\ \underline{sq(x^2,y^2)}\ \&\ \underline{sum(y^3,n^4)}\ \&\ PLUS(n^3,n^4,n^0)$
Fold sumdblsq
$\leftarrow sumdblsq(x^2,n^4)\ \&\ PLUS(n^2,n^2,n^3)\ \&\ TIMES(m^2,m^2,n^2)\ \&\ PLUS(n^3,n^4,n^0)$

The transformed procedures are .

```
sumdblsq(NIL,0) <-
sumdblsq(m.x,n) <- TIMES(m,m,p) & PLUS(p,p,q) & sumdblsq(x,r) &
                   PLUS(q,r,n)
```

A more interesting example is the simple sort program of example 11.

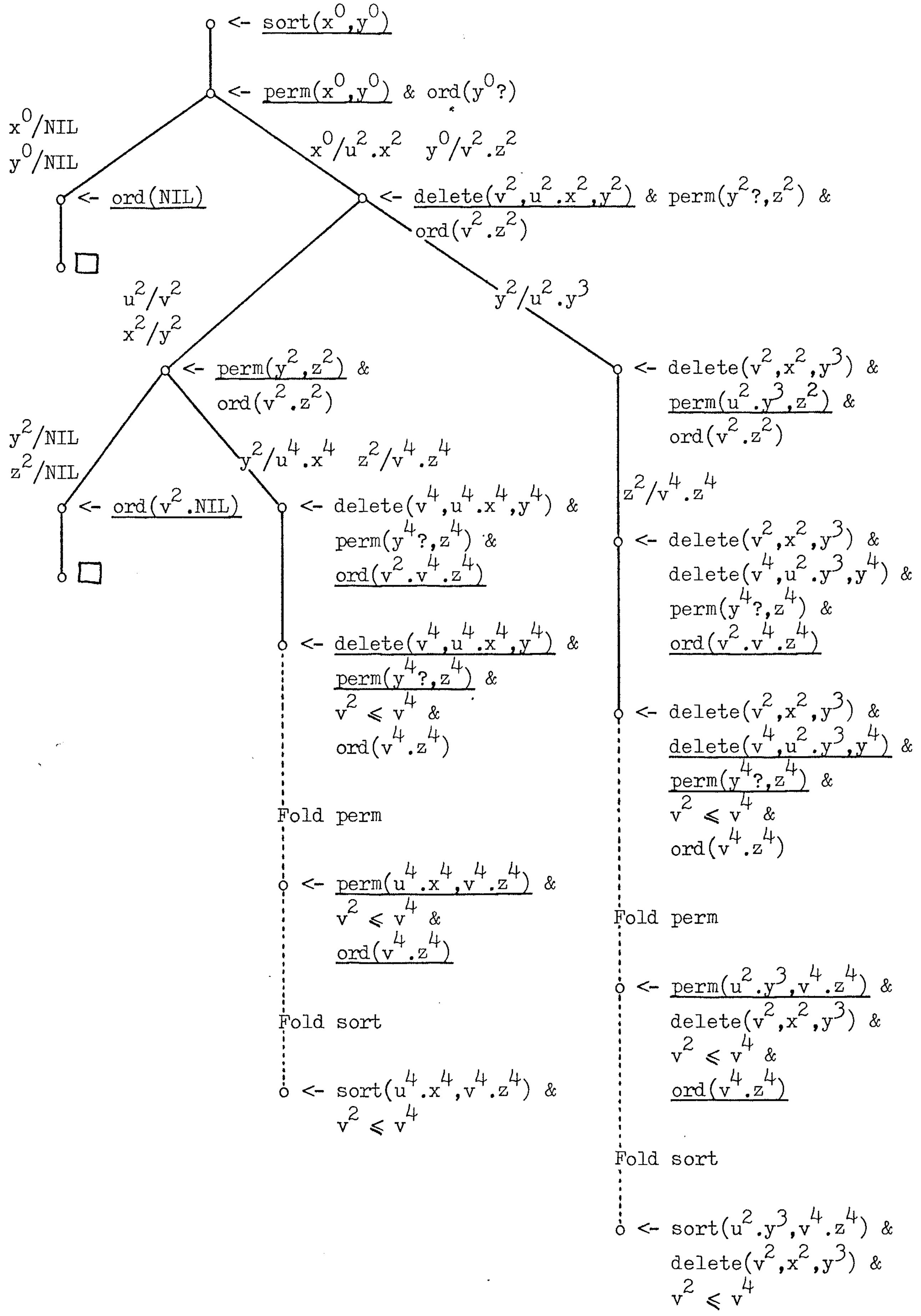
Example 11

```
sort(x,y) <- perm(x,y) & ord(y?)

perm(NIL,NIL) <-
perm(u.x,v.z) <- delete(v,u.x,y) & perm(y?,z)

delete(u,u.x,x) <-
delete(v,u.x,u.y) <- delete(v,x,y)

ord(NIL) <-
ord(u.NIL) <-
ord(u.v.x) <- u ≤ v & ord(v.x)
```

Here, sort(x,y) holds when list y is list x sorted; perm(x,z) holds when lists x and z are permutations; delete(v,x,y) holds when list y is x with v removed; ord(x) holds when list x is ordered.

The coroutining within the sort procedure has the effect that each permutation is rejected as soon as it is found to be unordered; the whole permutation need not be generated. The coroutining within the perm procedure is not necessary in normal execution. However, this must be included in transformation since otherwise the delete call would run indefinitely without making the jump that is needed. It is for similar reasons that we ignore the clause bar in transformation.

The transformation proceeds thus:


$\leftarrow\ \underline{\mathrm{sort}(x^0,y^0)}$
$\leftarrow\ \underline{\mathrm{perm}(x^0,y^0)}\ \&\ \mathrm{ord}(y^0?)$
x^0/NIL
y^0/NIL
$x^0/u^2.x^2\quad y^0/v^2.z^2$
$\leftarrow\ \underline{\mathrm{ord}(\mathrm{NIL})}$
□
$\leftarrow\ \underline{\mathrm{delete}(v^2,u^2.x^2,y^2)}\ \&\ \mathrm{perm}(y^2?,z^2)\ \&\ \mathrm{ord}(v^2.z^2)$
u^2/v^2
x^2/y^2
$y^2/u^2.y^3$
$\leftarrow\ \underline{\mathrm{perm}(y^2,z^2)}\ \&\ \mathrm{ord}(v^2.z^2)$
$\leftarrow\ \mathrm{delete}(v^2,x^2,y^3)\ \&\ \underline{\mathrm{perm}(u^2.y^3,z^2)}\ \&\ \mathrm{ord}(v^2.z^2)$
y^2/NIL
z^2/NIL
$y^2/u^4.x^4\quad z^2/v^4.z^4$
$z^2/v^4.z^4$
$\leftarrow\ \underline{\mathrm{ord}(v^2.\mathrm{NIL})}$
□
$\leftarrow\ \mathrm{delete}(v^4,u^4.x^4,y^4)\ \&\ \mathrm{perm}(y^4?,z^4)\ \&\ \underline{\mathrm{ord}(v^2.v^4.z^4)}$
$\leftarrow\ \mathrm{delete}(v^2,x^2,y^3)\ \&\ \mathrm{delete}(v^4,u^2.y^3,y^4)\ \&\ \mathrm{perm}(y^4?,z^4)\ \&\ \underline{\mathrm{ord}(v^2.v^4.z^4)}$
$\leftarrow\ \underline{\mathrm{delete}(v^4,u^4.x^4,y^4)}\ \&\ \underline{\mathrm{perm}(y^4?,z^4)}\ \&\ v^2 \leqslant v^4\ \&\ \mathrm{ord}(v^4.z^4)$
$\leftarrow\ \mathrm{delete}(v^2,x^2,y^3)\ \&\ \underline{\mathrm{delete}(v^4,u^2.y^3,y^4)}\ \&\ \underline{\mathrm{perm}(y^4?,z^4)}\ \&\ v^2 \leqslant v^4\ \&\ \mathrm{ord}(v^4.z^4)$
Fold perm
$\leftarrow\ \underline{\mathrm{perm}(u^4.x^4,v^4.z^4)}\ \&\ v^2 \leqslant v^4\ \&\ \underline{\mathrm{ord}(v^4.z^4)}$
Fold perm
$\leftarrow\ \underline{\mathrm{perm}(u^2.y^3,v^4.z^4)}\ \&\ \mathrm{delete}(v^2,x^2,y^3)\ \&\ v^2 \leqslant v^4\ \&\ \underline{\mathrm{ord}(v^4.z^4)}$
Fold sort
$\leftarrow\ \mathrm{sort}(u^4.x^4,v^4.z^4)\ \&\ v^2 \leqslant v^4$
Fold sort
$\leftarrow\ \mathrm{sort}(u^2.y^3,v^4.z^4)\ \&\ \mathrm{delete}(v^2,x^2,y^3)\ \&\ v^2 \leqslant v^4$

Transformed procedures:

```
sort(NIL,NIL) <-
sort(v.NIL,v.NIL) <-
sort(w.u.x,w.v.z) <- sort(u.x,v.z) & w ⩽ v
sort(u.x,w.v.z) <- delete(w,x,y) & sort(u.y,v.z) & w ⩽ v
```

In the above search tree, two branches each lead to a success node while two infinite branches are each cut to give a recursive procedure. Notice that on each of these branches we must continue executing until the ord call is activated, by which time the perm call has been executed too far. Hence the need for a fold with perm prior to the final fold with sort.

An interesting feature of this transformed procedure set is its nondeterminism. This reflects the behaviour of the original coroutined sort procedure, where backtracking occurs upon generating the first unordered member of a permutation. The first of the recursive procedures for sort corresponds to the case where the first element of the first list is its minimum. The second corresponds to the case where the minimum is elsewhere in the list.

Another point to note is that folding requires the use of the occur check. To fold each unfolded conjunction with sort, we must check that the second argument of the perm call unifies with the argument of the ord call. The occur check is needed so that the unification of $<z^4, v^4.z^4>$ fails.

The next example we shall consider illustrates the need for auxiliary relations. Sometimes a goal clause cannot be folded because it contains a conjunction which, although similar to some procedure body, does not unify with it. If the conjunction and the procedure body are both instances of some more general conjunction, then we should generalize the original procedure and repeat the transformation using this.

Example 12

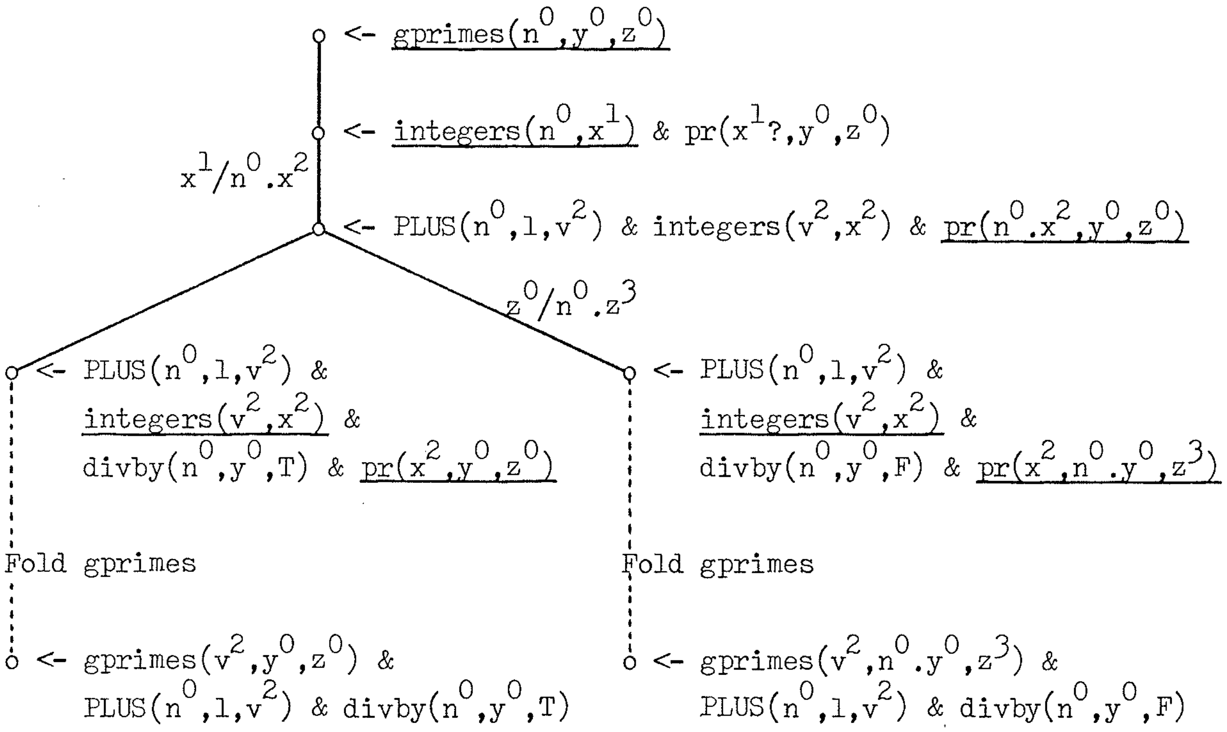
```
primes(z) <- integers(2,x) & pr(x?,NIL,z)

integers(u,u.x) <- PLUS(u,1,v) & integers(v,x)

pr(u.x,y,z) <- divby(u,y,T) & pr(x,y,z)
pr(u.x,y,u.z) <- divby(u,y,F) & pr(x,u.y,z)

divby(u,NIL,F) <-
divby(u,v.y,T) <- TIMES(v,w,u)
divby(u,v.y,r) <- ¬ TIMES(v,w,u) & divby(u,y,r)
```

This is a program for generating an infinite list z of prime numbers by the goal <- primes(z) . It works by coroutining between the generation of an infinite list of integers and checking the list for "primeness". integers(u,x) holds when x is a list of all integers in ascending order, starting at u; pr(x,y,z) holds when z is a list of the integers from list x which are deemed to be prime by checking against the list y of primes accumulated so far; divby(u,y,r) means r is T (true) iff u is divisible by any of the integers in list y. This algorithm will be discussed further in section 2.8.

In order to transform this program, we need to generalize the primes procedure by defining primes as an instance of gprimes:

```
primes(z) <- gprimes(2,NIL,z)
gprimes(n,y,z) <- integers(n,x) & pr(x?,y,z)
```

We can now go ahead and transform gprimes:



$\leftarrow \underline{gprimes(n^0,y^0,z^0)}$
$\leftarrow \underline{integers(n^0,x^1)}$ & $pr(x^1?,y^0,z^0)$
$x^1/n^0.x^2$
$\leftarrow PLUS(n^0,1,v^2)$ & $integers(v^2,x^2)$ & $\underline{pr(n^0.x^2,y^0,z^0)}$
$z^0/n^0.z^3$
$\leftarrow PLUS(n^0,1,v^2)$ & $\underline{integers(v^2,x^2)}$ & $divby(n^0,y^0,T)$ & $\underline{pr(x^2,y^0,z^0)}$
Fold gprimes
$\leftarrow gprimes(v^2,y^0,z^0)$ & $PLUS(n^0,1,v^2)$ & $divby(n^0,y^0,T)$
$\leftarrow PLUS(n^0,1,v^2)$ & $\underline{integers(v^2,x^2)}$ & $divby(n^0,y^0,F)$ & $\underline{pr(x^2,n^0.y^0,z^3)}$
Fold gprimes
$\leftarrow gprimes(v^2,n^0.y^0,z^3)$ & $PLUS(n^0,1,v^2)$ & $divby(n^0,y^0,F)$

The transformed procedures are

```
gprimes(n,y,z) <- PLUS(n,1,v) & divby(n,y,T) & gprimes(v,y,z)
gprimes(n,y,n.z) <- PLUS(n,1,v) & divby(n,y,F) & gprimes(v,n.y,z)
```

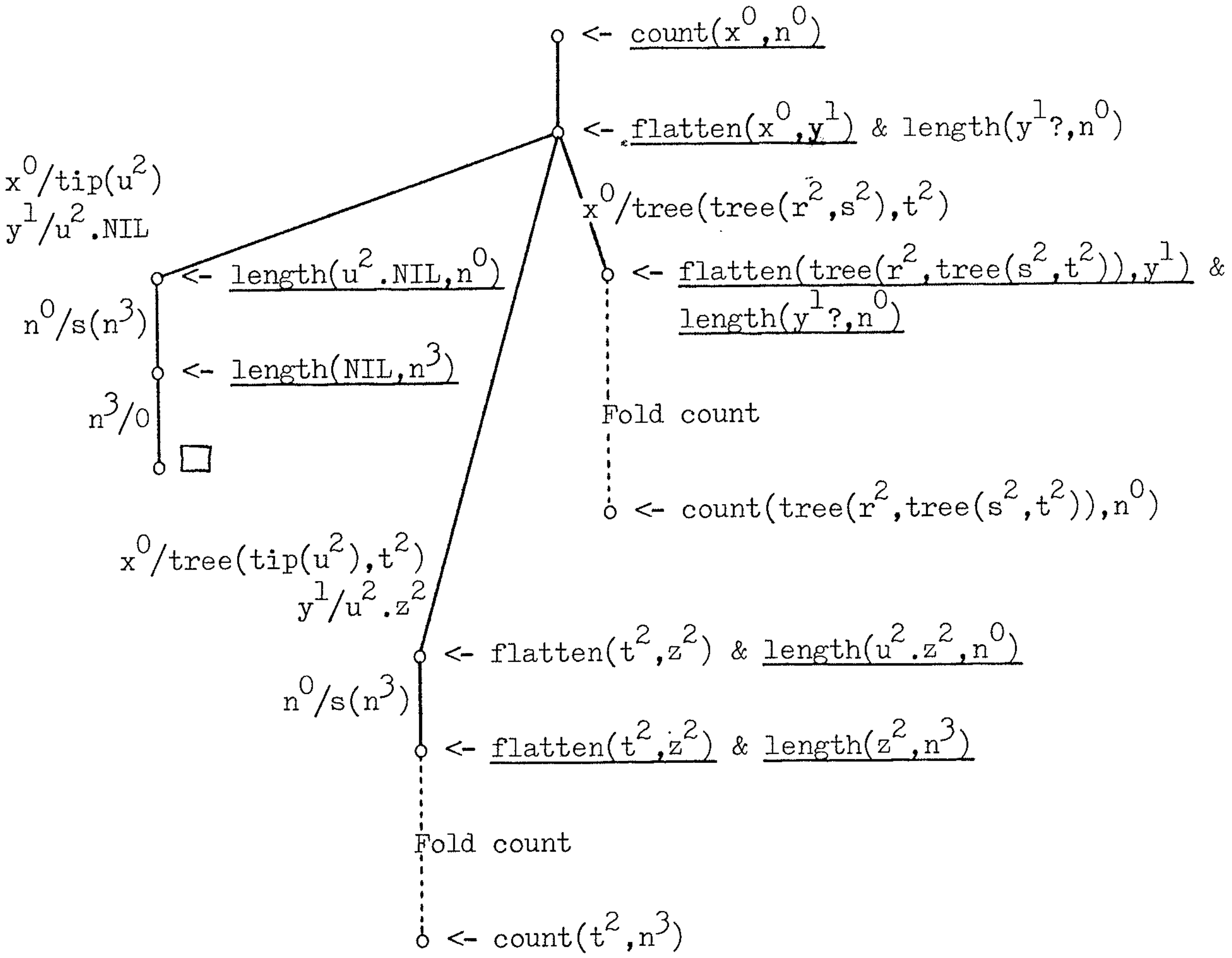
Finally, for effective computational use, these should be restructured to a conditional:

```
gprimes(n,y,x) <- PLUS(n,1,v) & divby(n,y,r) &
                  r = T
                     THEN gprimes(v,y,x)
                     ELSE(x = n.z &
                          gprimes(v,n.y,z))
```

We conclude this section with a tree manipulation program, adapted from [7]. Example 13 counts the number of tips in a binary tree by finding the length of the list representing the flattened tree. We use the flatten procedures derived in example 7. count(x,n) holds when tree x contains n tips.

Example 13

```
count(x,n) <- flatten(x,y) & length(y?,n)

flatten(tip(u),u.NIL) <-
flatten(tree(tip(u),t),u.z) <- flatten(t,z)
flatten(tree(tree(r,s),t),z) <- flatten(tree(r,tree(s,t)),z)

length(NIL,0) <-
length(u.x,s(n)) <- length(x,n)
```


$\leftarrow \underline{count(x^0,n^0)}$
$\leftarrow \underline{flatten(x^0,y^1)}$ & $length(y^1?,n^0)$
$x^0/tip(u^2)$
$y^1/u^2.NIL$
$\leftarrow \underline{length(u^2.NIL,n^0)}$
$n^0/s(n^3)$
$\leftarrow \underline{length(NIL,n^3)}$
$n^3/0$
□
$x^0/tree(tree(r^2,s^2),t^2)$
$\leftarrow \underline{flatten(tree(r^2,tree(s^2,t^2)),y^1)}$ & $\underline{length(y^1?,n^0)}$
Fold count
$\leftarrow count(tree(r^2,tree(s^2,t^2)),n^0)$
$x^0/tree(tip(u^2),t^2)$
$y^1/u^2.z^2$
$\leftarrow flatten(t^2,z^2)$ & $\underline{length(u^2.z^2,n^0)}$
$n^0/s(n^3)$
$\leftarrow \underline{flatten(t^2,z^2)}$ & $\underline{length(z^2,n^3)}$
Fold count
$\leftarrow count(t^2,n^3)$

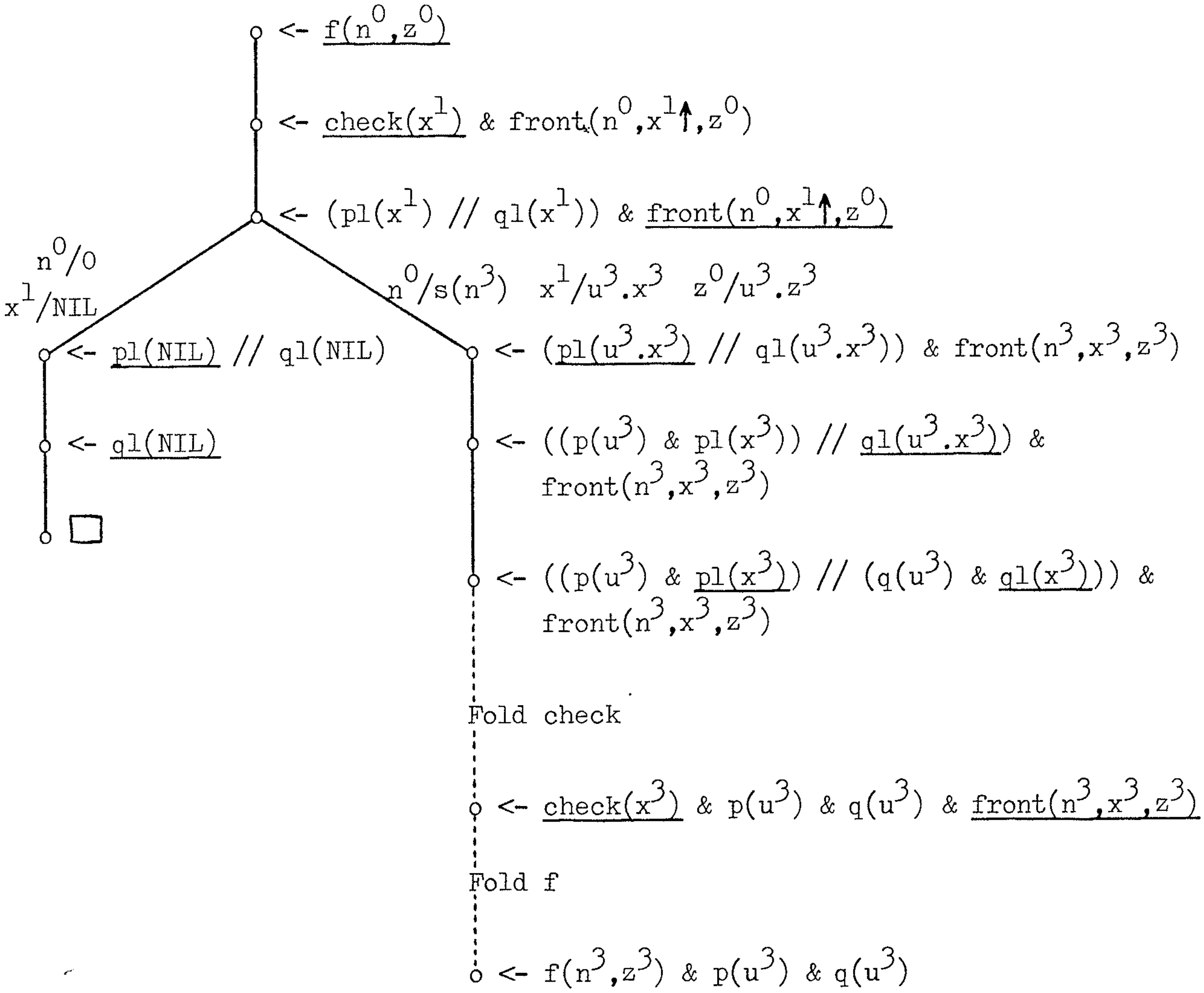
The transformed procedures are

```
count(tip(u),s(0)) <-
count(tree(tip(u),t),s(n)) <- count(t,n)
count(tree(tree(r,s),t),n) <- count(tree(r,tree(s,t)),n)
```

2.7 Parallelism in transformation

We can "compile" programs utilizing parallelism in the same manner as those already considered, namely by controlled symbolic execution and folding. We use the mechanism for pseudo-parallel execution that is described in section 1.4.

We begin by compiling the program of example 3 to an equivalent sequential program.


<- f(n^0,z^0)
<- check(x^1) & front(n^0,$x^1\uparrow$,z^0)
<- (pl(x^1) // ql(x^1)) & front(n^0,$x^1\uparrow$,z^0)
n^0/0
x^1/NIL
n^0/s(n^3) x^1/u^3.x^3 z^0/u^3.z^3
<- pl(NIL) // ql(NIL)
<- ql(NIL)
□
<- (pl(u^3.x^3) // ql(u^3.x^3)) & front(n^3,x^3,z^3)
<- ((p(u^3) & pl(x^3)) // ql(u^3.x^3)) & front(n^3,x^3,z^3)
<- ((p(u^3) & pl(x^3)) // (q(u^3) & ql(x^3))) & front(n^3,x^3,z^3)
Fold check
<- check(x^3) & p(u^3) & q(u^3) & front(n^3,x^3,z^3)
Fold f
<- f(n^3,z^3) & p(u^3) & q(u^3)

The transformed procedures are

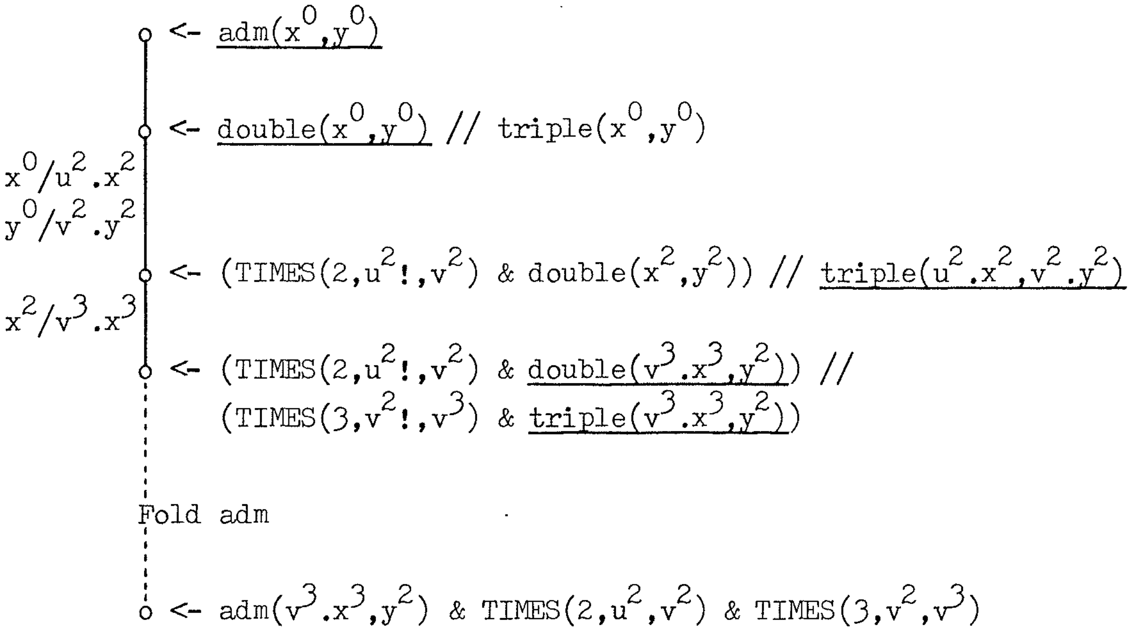
```
f(0,z) <-
f(s(n),u.z) <- p(u) & q(u) & f(n,z)
```

Example 14 shows the program for Kowalski's "admissible pairs" problem [16], suitably annotated to give the desired behaviour. This program generates a pair of infinite lists possessing the "admissible" relationship.

Example 14

```
adm(x,y) <- double(x,y) // triple(x,y)

double(u.x,v.y) <- TIMES(2,u!,v) & double(x,y)

triple(u.v.x,w.y) <- TIMES(3,w!,v) & triple(v.x,y)
```

The "!" annotations ensure that the process executing double suspends until its incoming list is instantiated; likewise with the process executing triple. We can "compile" this program using the normal parallel mechanism.



<- adm(x⁰,y⁰)
<- double(x⁰,y⁰) // triple(x⁰,y⁰)
x⁰/u².x²
y⁰/v².y²
<- (TIMES(2,u²!,v²) & double(x²,y²)) // triple(u².x²,v².y²)
x²/v³.x³
<- (TIMES(2,u²!,v²) & double(v³.x³,y²)) //
(TIMES(3,v²!,v³) & triple(v³.x³,y²))
Fold adm
<- adm(v³.x³,y²) & TIMES(2,u²,v²) & TIMES(3,v²,v³)

The transformed procedure is

```
adm(u.v.x,w.y) <- TIMES(2,u,w) & TIMES(3,w,v) & adm(v.x,y)
```

2.8 Processes as data structures

There is a class of programs, annotated for coroutining or parallelism, which cannot be "compiled" using the techniques we have considered. These are those programs which use the methodology of "processes as data structures" that we discussed in section 1.6. For example, consider the problem of generating an infinite list of prime numbers. Example 15 shows the natural way of programming this in IC-PROLOG.

Example 15

```
primes(z) <- integers(2,x) & sift(x?,z)

integers(u,u.x) <- PLUS(u,1,v) : integers(v,x)

sift(u.x,u.z) <- sieve(u,x,y) & sift(y?,z)

sieve(u,v.x,y) <- TIMES(u,w,v) : sieve(u,x,y)
sieve(u,v.x,v.y) <- ¬ TIMES(u,w,v) : sieve(u,x,y)
```

This is the algorithm of Eratosthenes' sieve; an infinite list z of primes is generated by the goal <- primes(z) . Neglecting the coroutining annotations, it can be seen that a list of all the integers is generated and then sifted. The sifting consists of removing from the list all multiples of the first number, which is a prime, and then sifting the resulting list in the same manner, recursively. The coroutining means that we need not wait for the infinite list to be generated before obtaining results.

The coroutined program has the same behaviour as that given by Hoare [12] in his Communicating Sequential Processes notation. There is essentially one process for each prime generated so far; this process is engaged in removing multiples of its prime from its input list and passing the resulting list to its successor. Any number which gets through the entire list of processes is a prime, and a new process is created for that number.

Suppose that the primes 2, 3 and 5 have been generated so far: the situation is

```
integers(6,x) & sieve(2,x?,a) & sieve(3,a?,b) & sieve(5,b?,c) &
sift(c?,z)
```

At the next step, the integers process generates the number 6. This only passes as far as the first sieve process which discovers that it is divisible by 2:

```
integers(7,x') & sieve(2,x'?,a) & sieve(3,a?,b) & sieve(5,b?,c) &
sift(c?,z)
```

Now the integers process generates 7, which passes through all of the sieves and creates a new sieve process:

```
integers(8,x") & sieve(2,x"?,a') & sieve(3,a'?,b') &
sieve(5,b'?,c') & sieve(7,c'?,d) & sift(d?,z')
```

The output list z has been instantiated to 7.z' , thus generating 7 as the next prime.

If we symbolically execute the above program, we fail to obtain a pattern of atoms that can be folded. This is a consequence of the "inherent parallelism" of the program: the list of "primes generated so far" is represented by a list of sieve processes. We must reformulate the program so that this list is represented by a term in the conventional manner. The reformulated program has appeared earlier, as example 12, in which each integer found to be prime is added to the head of an explicit list.

In both formulations of the program, each candidate integer is checked for divisibility by each of the primes so far generated. The programs differ, however, in the order of checking: example 15 checks in ascending order, example 12 in descending order - a less efficient algorithm. In order for a program of the form of example 12 to perform the check in ascending order, we would need to add each newly discovered prime to the tail of the prime list. The second procedure for pr would become

```
pr(u.x,y,u.z) <- divby(u,y,F) & insert(u,y,w) & pr(x,w,z)
```

where

```
insert(u,NIL,u.NIL) <-
insert(u,v.y,v.z) <- insert(u,y,z)
```

This is still a different algorithm from that of example 15, since more work is required to insert the new prime, depending upon the length of the list. It appears that coroutining is essential to obtain the optimum algorithm.

Clark [3] describes a program for the eight queens problem, which he reformulates in a similar manner in order to eliminate parallelism. His original program contains the procedure

```
safe(u.x) <- notake(u,x,1) // safe(x)
```

to check that a list of queen positions is "safe". As the computation proceeds, parallel processes are created, each representing a queen that has been placed; each process is engaged in checking its queen against all subsequently placed queens. To obtain a sequential program, the list of already-placed queens is made explicit as a term and the following procedures used:

```
safe(x) <- safepair(NIL,x)

safepair(y,u.x) <- notake(u,y,1) & safepair(u.y,x)
safepair(y,NIL) <-
```

Despite the different representation, the behaviour of the two programs is almost identical, although it is slightly overspecified in the sequential, lower-level program.

CHAPTER 3

IMPLEMENTATION OF A COMPILATION SYSTEM

We have seen in chapter 2 that annotated logic programs can be "compiled" by controlled symbolic execution followed by folding. In this chapter we shall describe a system (SCALP) which has been implemented to compile programs with coroutining annotations by this method.

Vasey [20] has written, in Waterloo PROLOG, an interpreter for sequential logic programs. Our system is based on this, but includes two novel features: dataflow coroutining and folding. Later in the chapter we suggest how the system may be extended further to incorporate pseudo-parallelism in the symbolic execution phase.

3.1 Use of the system

To perform a transformation, the user creates a file containing a number of clauses of Waterloo PROLOG; this is loaded, together with the SCALP program, into the PROLOG interpreter. Four items of information are given in this file, described below:

a) Set of clause assertions.
These represent the clauses of the program to be transformed. Each clause assertion represents all of the Horn clauses for one predicate, including any annotations; the restrictions quoted in section 1.3 concerning the use of annotations must be observed. The exact form of the representation is described in the next section.

b) Set of primitive assertions.
These specify which predicates are "primitive", i.e. not to be executed during the transformation. These usually coincide with the "built in" predicates of PROLOG. e.g.

```
primitive(PLUS)
primitive(TIMES)
```

c) Foldorder assertion.
This names a list of predicates, specifying the order in which folding is to

be attempted. It is the means by which the user controls the folding phase. The list always begins with the top level predicate, followed by successively lower level predicates. For example, in the program of example 10, we wish to fold the goal clause

```
<- sum(y,n) & sq(x,y) & TIMES(m,m,p) & PLUS(p,n,q)
```

To obtain the desired recursive procedure, we ought to fold with sumsq rather than with sum or sq; the last two would merely undo the work of the symbolic execution. Hence, we would provide the assertion

```
foldorder(sumsq.sum.sq.NIL)
```

Sometimes, a fold must be performed on a lower level predicate before the final fold is possible, as in example 11. In such cases, the remainder of the foldorder list is utilized.

d) Call to transform.

A call to the transform procedure giving as its argument the atom to be transformed (the goal atom) in the format described in the next section. If this call appears in a procedure:

```
q <- transform(...)
```

then the transformation may be run by the goal <-q .

The system symbolically executes the specified goal atom using the program clauses supplied. At each step of the execution, trace information is displayed: e.g. procedure call, disjunction, jump, return, etc. and the level number is included. The execution procedure itself is described in the next section. Immediately prior to each extension of the proof tree, i.e. when a call is about to be replaced by a body, the system prompts the user for a command: either "continue" or "stop". A "continue" command allows the execution to proceed; a "stop" command halts the execution of that branch of the search tree.

Upon a "stop" command, the current derived procedure is displayed: the head of this is the current substitution instance of the goal atom; its body is the current goal clause. The system then tries to fold the body of this procedure to obtain a recursion on the top level predicate as specified in the foldorder list. For each predicate the various procedures are tried in turn. If a fold is not possible with the top level predicate, lower level predicates are tried. When a fold is eventually made, the system again tries to fold with the top level predicate, and so on until this is successful.

When the folding is complete, the folded procedure is displayed as one of the solutions. The system then backtracks and pursues the next branch in the search tree to seek further solutions. When there are no further branches, the transformation is complete and a message is displayed to this effect.

The solutions produced during transformation are the transformed procedures. It only remains for the user to rearrange the try order and rename variables.

3.2 Implementation of system

In this section we shall continue to use the syntax of IC-PROLOG, although the system itself is written in Waterloo PROLOG. In the syntax of the latter, all code is in upper case and variables are distinguished from constants by being prefixed by "*".

Representation of logic program components

A constant sym is represented by the term k(sym) . e.g. NIL becomes k(NIL) ; 4 becomes k(4) .

A variable n, used on level 1, i.e. n^1 is represented by the term v(n',1) , where n' is n written as a constant. e.g. x^5 becomes v(X,5) .

A compound term func($term_1$,...,$term_n$) is represented by the term f(func',$term_1'$. ... .$term_n'$.NIL) , where func' is func written as a constant and each $term_i'$ is the representation of $term_i$. e.g. cons(x^5,NIL) becomes f(CONS,v(X,5).k(NIL).NIL) .

An atom pred($term_1$,...,$term_n$) is represented by the term atom(pred',$term_1'$. ... .$term_n'$.NIL) , where pred' is pred written as a constant and each $term_i'$ is the representation of $term_i$. e.g. fact(4,z^2) becomes atom(FACT,k(4).v(Z,2).NIL) .

An equality $term_1$ = $term_2$ is represented by the term $term_1'$ @ $term_2'$, where each $term_i'$ is the representation of $term_i$.

A conjunction $atom_1$& ... & $atom_n$ is represented by the term $atom_1'$ # ... # $atom_n'$ # T , where each $atom_i'$ is the representation of $atom_i$.

Representation of logic programs

The original Horn clause program must first be written in a modified form such that there is only one clause for each predicate. The body of each such clause may be either a conjunction or a disjunction of conjunctions, each of which may begin with a number of equalities to bind head variables to terms for use in that conjunction. Note that a procedure head may only contain variables and moreover these must not appear annotated in the body. For example,

```
append(NIL,y,y) <-
append(cons(u,x),y,cons(u,z)) <- append(x,y,z)
```

becomes

```
append(xx,y,zz) <-
        (xx = NIL & y = zz) |
        (xx = cons(u,x) & zz = cons(u,z) & append(x,y,z))
```

If a head variable is annotated in the body, e.g.

```
front(n,x,z) <- append(x,y,z) & length(x?,n)
```

then a different variable must be used in the head and equated with the annotated one:

```
front(n,xx,z) <- xx = x & append(x,y,z) & length(x?,n)
```

The reason for this form of procedure is to allow a call to be replaced by a body without the possibility of either a failure - the head variables can always unify with terms in the call - or a coroutining jump. These may occur later while matching terms in the equalities.

For use by the SCALP system, the program is represented by a set of clause assertions, one for each predicate:

```
clause(predicate,level,head-variables,body)
```

predicate is the predicate part of the procedure head, written as a constant.

level is a variable, to be bound to an integer when the procedure is invoked.

head-variables is a list of the representations of the variables constituting the arguments of the procedure head.

body is the representation of the procedure body: either a conjunction or a disjunction of conjunctions. A body which is a disjunction

$\text{conj}_1 \mid \ldots \mid \text{conj}_n$ is represented as $(\text{conj}_1' + \ldots + \text{conj}_n')$ # T , where each conj_i' is the representation of conj_i. A conjunction conj containing coroutining annotations is represented as cor(conj') # T .

There are many examples of this representation to be found in the appendix.

Data structures used

The principle data structure is a list of **nodes**, representing the search tree. The first of these is the **current node:**

current-node.other-nodes

The current node represents the current branch of the search tree; the others are retained for backtracking purposes. In a deterministic computation only one node would be required throughout. Each node contains a level number and a list of **processes**, the first of which is the **current process:**

node(level,current-process.other-processes)

The **level** of each node is either an <integer> or a term old(<integer>) , and the nodes are consecutively numbered starting at 0, e.g. node(3,...).node(old(2),...).node(1,...).node(0,...).NIL . The level of the current node is the **current level.**

Each process may be either **direct:**

pr(id,dir(type,status,done-items,items))

or **indirect:**

pr(id,ind(parent-id))

There always exists one (direct) process whose id is MAIN. If there is no coroutining this will be the only process; otherwise the id of every other process will be a representation of a variable, e.g. v(X,1) . The parent-id of an indirect process is the id of another (direct) process. The type of a direct process is either CONSUMER or PRODUCER. The status of a process is either PASSIVE or active(return-id) , where return-id is the id of another process.

done-items and items are each a representation of a conjunction. done-items contains only atoms with primitive predicates. Each conjunct of items (unless items is empty) is termed an **item** - not necessarily an atom - the first of which is the **current item:**

```
current-item # other-items
```

The binding environment is represented by a set of assertions:

```
bind(level,variable,term)
```

indicating the level at which a variable is bound to a certain term.

Basic symbolic execution mechanism

We describe here the mechanism for symbolic execution in the absence of coroutining. This is a slight modification of that used in Vasey's system.

Initially there is just one node, whose level is 0, containing one process, whose id is MAIN and whose type and status are irrelevant; done-items is empty; items contains just the goal atom (that which is to be executed). Thus the initial node list is

```
node(0,pr(MAIN,dir(X,X,T,goal-atom # T)).NIL).NIL
```

Assuming there is no coroutining, only one process exists in any node and the conjunction of items in this process is executed left - right depth first.

We describe the mechanism as a repeated application of rules to the data structure

```
node(current-level,
     pr(id,dir(type,status,done-items,current-item # other-items))
     .other-processes).other-nodes
```

Rule a) Current item is an atom with a primitive predicate. Move the current item to the done-items conjunction. The node list becomes

```
node(current-level,
     pr(id,dir(type,status,current-item # done-items,other-items))
     .other-processes).other-nodes
```

Rule b) Current item is an atom with a predicate for which a procedure exists. Change current-level to old(current-level) in the current node. Add a new node whose level is current-level + 1 and which is a copy of the current node except that the current item is replaced by the body of the corresponding procedure. At the same time, the procedure's head variables become bound to the arguments of the current item. The node list becomes

```
node(current-level + 1,
      pr(id,dir(type,status,done-items,new-items))
     .other-processes)
.node(old(current-level),
      pr(id,dir(type,status,done-items,current-item # other-items))
     .other-processes).other-nodes
```

where new-items denotes the body conjoined to other-items. The reason for retaining the old node is to provide for possible partial backtracking in connection with coroutining - this will be discussed later. In the absence of coroutining, "old" nodes will always be ignored.

Rule c) Current item is a disjunction, say altl + alt2 . Replace the current item by alt2 in the current node. Add a new node which is a copy of the current node except that its level is incremented by 1 and that the current item is replaced by altl:

```
node(current-level + 1,
      pr(id,dir(type,status,done-items,new-itemsl))
     .other-processes)
.node(current-level,
      pr(id,dir(type,status,done-items,new-items2))
     .other-processes).other-nodes
```

where new-itemsl is altl conjoined to other-items and new-items2 is alt2 conjoined to other-items.

Rule d) Current item is an equality, say terml @ term2 . Try to match terml and term2. If the match succeeds, any necessary bindings are induced and the current item is simply removed. If however the match fails, we must backtrack to the last branch point. This is done by deleting the current node together with all subsequent nodes with an "old" level. Any bindings made by the deleted nodes' levels are undone.

The remaining rule applies when the items conjunction is empty, i.e. the node list is

```
node(current-level,
      pr(id,dir(type,status,done-items,T))
     .other-processes).other-nodes
```

Rule e) Items is empty.
i.e. have finished the MAIN process. This means that we have a solution - an assertion - which consists of the goal atom with the current bindings applied. Print this solution and then backtrack to try other branches.

Coroutining symbolic execution mechanism

In section 1.3, we discussed how a coroutined execution could be regarded in terms of processes. We shall extend the above rules in order to implement this mechanism.

The MAIN process is permanently in existence, constructing the whole proof tree. When a coroutining interaction is in effect, there are a number of other processes, each developing a subtree of the proof tree. Each process other than MAIN is uniquely identified by a variable: that which is annotated in the atom at the root of the process's subtree. Such processes first come into being when a procedure is invoked whose body contains coroutining annotations. Our next rule caters for this situation.

Rule f) Current item is cor(conj) .
conj is a conjunction bearing annotations. Suppose that conj is

$$A_1(t_{11},\ldots,t_{1m_1}) \;\&\; \ldots \;\&\; A_j(t_{j1},\ldots,x?,\ldots,t_{jm_j}) \;\&\; \ldots \;\&$$
$$A_k(t_{k1},\ldots,y\uparrow,\ldots,t_{km_k}) \;\&\; \ldots \;\&\; A_n(t_{n1},\ldots,t_{nm_n})$$

We preprocess conj to remove the annotations and place each annotated atom in a new process. Each annotated atom is replaced in conj by the term p(var) , where var is the annotated variable. For each annotated atom a new direct process is created, whose id is the annotated variable. The type of the new process is either CONSUMER or PRODUCER according to whether the annotation was "?" or "↑", while its status is PASSIVE. The done-items of the new process is empty and the items conjunction contains just the annotated atom with its annotation removed. In the case of the above example, the preprocessed conjunction is

$$\mathrm{atom}(A_1,t_{11}.\ \ldots\ .t_{1m_1}.\mathrm{NIL}) \;\#\; \ldots \;\#\; p(x) \;\#\; \ldots \;\#\; p(y) \;\#\; \ldots \;\#$$
$$\mathrm{atom}(A_n,t_{n1}.\ \ldots\ .t_{nm_n}.\mathrm{NIL}) \;\#\; T$$

and the two new processes are

$$\text{pr}(x,\text{dir}(\text{CONSUMER},\text{PASSIVE},\text{T},\text{atom}(A_j,t_{j1}.\ \ldots\ .t_{jm_j}.\text{NIL})\ \#\ \text{T}))$$

and

$$\text{pr}(y,\text{dir}(\text{PRODUCER},\text{PASSIVE},\text{T},\text{atom}(A_k,t_{k1}.\ \ldots\ .t_{km_k}.\text{NIL})\ \#\ \text{T}))$$

The preprocessed conjunction cor-conj then replaces the current item and the new processes cor-procs are added to the list of processes in the current node. The node list becomes

```
node(current-level,
     pr(id,dir(type,status,done-items,new-items))
    .new-processes).other-nodes
```

where new-items is cor-conj conjoined to other-items, and new-processes is cor-procs appended to other-processes.

During execution, indirect processes may be created: whenever a variable for which a process exists is bound to a non-variable term, its process is inherited by all variables (if any) in that term. Suppose y, which has a direct process, is bound to the term $f(x_1,\ldots,x_n)$. Then n indirect processes are created (one for each variable in the term):

$$\text{pr}(x_1,\text{ind}(y))$$
$$\vdots$$
$$\text{pr}(x_n,\text{ind}(y))$$

We can now explain how the coroutining interaction is effected. A jump or return is triggered whenever an annotated variable (i.e. one for which a process exists) is bound to a non-variable, either directly or indirectly. When any variable is bound to a non-variable, we see whether there exists a process (either direct or indirect) for the variable being bound. We then conceptually set a flag indicating the direct process concerned, if necessary after indirection via an indirect process. The actual jump or return is performed by our next rule.

Rule g) If the control flag is set, identifying process dpr (regardless of the current item) then continue execution until all equalities (if any) are dealt with, then do the jump or return as follows.

Firstly, if the process dpr is an active consumer or a passive producer - deduced from its type and status - then the last execution step is undone. This is achieved by deleting the current node so that the last "old" node becomes current.

Next, if the type of dpr is PASSIVE, a jump must be performed to dpr. This is done by changing the status of dpr to active(cid) , where cid is the id of the current process, and making dpr the current process (by placing it at the head of the list of processes).

Alternatively, if the type of dpr is active(rid) a return must be performed to the process whose id is rid. This is done by changing the status of dpr to PASSIVE, and making the process identified by rid the current one.

The next rule deals with the case of a subtree being entered in the left - right depth first order, while being constructed by a separate process.

Rule h) Current item is p(pid) .
The process whose id is pid is removed from the process list and the construction of its subtree is continued by the current process. This is done by adding the done-items and items of the terminated process to those of the current process.

Finally, our previous rule e) must be revised as follows.

Rule e') Items is empty.
i.e. have finished a process. If the id of the current process is MAIN, we have a solution: print it and backtrack as before. Otherwise, return from the current process by the method given in rule g).

Folding

As mentioned in section 3.1, the user is prompted for input prior to every application of rule a) or b). If the command "stop" is given, the system flattens the contents of all processes of the current node into a single conjunction and attempts to fold this, repeatedly if necessary. We have already stated the order in which the procedures are tried in the search for a fold. For each procedure, a fold is sought as follows.

Firstly, a new instance of the procedure body is taken, using a level number greater than any previously used; this ensures that the variables in the body are distinct from those in the conjunction. The atoms in the procedure body are matched against the first atoms of all permutations of the current conjunction until a match is found. Then the equalities at the beginning of the procedure body are evaluated (this will always succeed) and the matched part of the conjunction is replaced by the procedure head.

For example, consider the fully unfolded conjunction of example 11:

$$\mathrm{delete}(v^2,x^2,y^3)\ \&\ \mathrm{delete}(v^4,u^2.y^3,y^4)\ \&\ \mathrm{perm}(y^4,z^4)\ \&\ v^2 \leqslant v^4\ \& \\ \mathrm{ord}(v^4.z^4)$$

To fold this with perm, we use the procedure

$$\mathrm{perm}(xx^6,zz^6) \text{ <- } xx^6 = u^6.x^6\ \&\ zz^6 = v^6.z^6\ \&\ \mathrm{delete}(v^6,u^6.x^6,y^6)\ \& \\ \mathrm{perm}(y^6,z^6)$$

The atoms

$$\mathrm{delete}(v^6,u^6.x^6,y^6)\ \&\ \mathrm{perm}(y^6,z^6)$$

match with the first portion of the permuted conjunction

$$\mathrm{delete}(v^4,u^2.y^3,y^4)\ \&\ \mathrm{perm}(y^4,z^4)\ \&\ \mathrm{delete}(v^2,x^2,y^3)\ \&\ v^2 \leqslant v^4\ \& \\ \mathrm{ord}(v^4.z^4)$$

inducing the bindings v^6/v^4, u^6/u^2, x^6/y^3, y^6/y^4, z^6/z^4. When the equalities are evaluated, we get the bindings $xx^6/u^2.y^3$ and $zz^6/v^4.z^4$. Replacing the first two atoms of the conjunction by the procedure head, we have

$$\mathrm{perm}(u^2.y^3,v^4.z^4)\ \&\ \mathrm{delete}(v^2,x^2,y^3)\ \&\ v^2 \leqslant v^4\ \&\ \mathrm{ord}(v^4.z^4)$$

as required.

The need for the occur check in folding has been noted in section 2.6. Our example above illustrates this. If the occur check were not used, it would have been possible to fold the original conjunction with the sort procedure:

$$\mathrm{perm}(y^4,z^4)\ \&\ \mathrm{ord}(v^4.z^4)\ \&\ \ldots$$

would be replaced by

$$\mathrm{sort}(y^4,z^4)\ \&\ \ldots$$

inducing the self-referential binding $z^4/v^4.z^4$.

Also during folding, while matching the conjunction against a procedure body, no variable of the conjunction may be bound, for the following reason. Let $G_0,G_1,\ldots$ denote the bodies of the successive goal clauses in the transformation; then for each i, G_i is implied by G_{i+1}. Specifically, if we are folding G_n to G_{n+1} then G_{n+1} must imply G_n. However, if this fold were to bind any variables of G_n, then G_{n+1} would only imply some substitution instance of G_n, not G_n itself. In the present example, it would have been possible - were this check not performed - to fold the permuted conjunction

$$\text{delete}(v^2,x^2,y^3) \;\&\; \text{perm}(y^4,z^4) \;\&\; \text{delete}(v^4,u^2.y^3,y^4) \;\&\; v^2 \leqslant v^4 \;\&\; \text{ord}(v^4.z^4)$$

with the perm procedure, giving

$$\text{perm}(u^6.x^6,v^2.z^4) \;\&\; \ldots$$

and binding $x^2/u^6.x^6$ and y^3/y^4 .

The above fold would not have been logically valid for another reason: the rule stated in section 2.3. This rule forbids the fold because the existentially quantified y^6 in the perm procedure matches with y^3 in the first atom, which also appears in the third atom. This rule is not actually checked by the system; neither is the requirement that all procedures for a predicate treat disjoint tuples of arguments.

3.3 Possible enhancements to system

Our existing system does not provide for the application of laws or functionality, as described in section 2.3. A simple way to provide these would be to allow the user to enter a new conjunction - to replace the fully unfolded one - immediately prior to folding. The user would then be responsible for the correctness of the rearranged conjunction.

A more automatic method for applying laws would be for the user to supply a set of equivalences in the form that we have considered. The system would automatically search the unfolded conjunction, in a similar manner to that employed in folding, for opportunities to apply the laws. A problem with this method is the overhead that would be incurred by the search. Also, there would often be a large number of possible rearrangements, most of which would not result in the desired fold.

It should be easier to implement a method for automatically applying functionality simplification. The user would enumerate the circumstances in which each relation is a function in a similar manner to the IC-PROLOG directive $FUNCTION: $FUNCTION(p,$a_1,\ldots,a_n$) states that p is a function if its a_1th and ... and a_nth arguments are all given. The system would search the unfolded conjunction for pairs of atoms having the same predicate. It would then note all argument positions in which the atoms have identical arguments and check whether these are subsumed by those in any $FUNCTION specification for the predicate.

3.4 Adding parallelism to system

In section 1.4 we discussed a control strategy embodying both coroutining and parallelism. We now suggest a possible implementation of this scheme within the framework of the SCALP system. The difference lies in the symbolic execution phase; folding is performed in the same way. We shall need a modified data structure and execution mechanism.

Data structures used

Since backtracking works as before, we still have a list of nodes, of which the first is the current node. The contents of each node are different, however:

node(level,current-process-set,suspension-record-set)

The level of a node has the same role as previously.

The **current process set** fulfils the purpose described in section 1.4, i.e. it contains the processes being executed concurrently. It is a list of processes, the first of which is the **current process**:

current-process.other-processes

The **suspension record set** is a list of **suspension records**, each of which may be either **direct**:

susp(var,dir(type,suspension-type,suspended-processes))

or **indirect**:

susp(var,ind(parent-var))

The suspension record set in a node will be empty if there is no coroutining; otherwise it will contain one direct suspension record for each annotated variable. The annotated variable occupies the var field of the corresponding suspension record. Indirect suspension records have the same purpose as indirect processes in the original implementation: one is created for each variable in a term bound to a variable for which a suspension record exists.

The type of a direct suspension record is either CONSUMER or PRODUCER, as is the suspension-type. suspended-processes is a list of all processes which are coroutine suspended on the variable in the var field of the suspension record. All processes suspended thus will be of the same type, which is indicated by the suspension-type of the record.

Each process is now of the form

```
pr(id,origin,ancestor-vars,done-items,items)
```

The id of a process is no longer a variable but is either MAIN or some unique identifier. The origin of a process is either

```
parallel(parent-id)
```

where parent-id is the id of another process, or

```
coroutined(var)
```

where var is the representation of a variable. ancestor-vars is a list of representations of variables; done-items and items are the same as previously.

Parallel symbolic execution mechanism

Initially there is one node, whose level is 0. Its current process set contains just one process, whose id is MAIN and whose origin is irrelevant; ancestor-vars and done-items are empty; items contains just the goal atom. The suspension record set is empty. The initial node list is therefore

```
node(0,pr(MAIN,X,NIL,T,goal-atom # T).NIL,NIL).NIL
```

If there is no coroutining, the suspension record set will remain empty. In the absence of parallelism, there will never be more than one process in the current process set.

We shall again explain the mechanism as a number of steps, each applying one of several rules to the data structure

```
node(current-level,
     pr(id,origin,ancestor-vars,done-items,
        current-item # other-items)
    .other-processes,
     suspension-record-set).other-nodes
```

according to the current item. Note that between steps of the execution, the current process is cycled to the tail end of the current process set. This has the effect of timeslicing between the processes.

Rule a") Current item is an atom with a primitive predicate. This rule is analogous to the old rule a), i.e. move the current item to done-items.

Rule b") Current item is an atom with a predicate for which a procedure exists. Firstly, check that any arguments of the atom which are suffixed by the "!" annotation are bound to a non-variable term. If so, apply the equivalent of rule b), otherwise do nothing (i.e. the current process is temporarily suspended).

Rule c") Current item is a disjunction. Analogous to the old rule c).

Rule d") Current item is an equality. Analogous to the old rule d).

Rule f") Current item is cor(conj) . Here, as in the old rule f), coroutining is introduced. Suppose again that conj is the annotated conjunction

$$A_1(t_{11},\ldots,t_{1m_1}) \ \& \ \ldots \ \& \ A_j(t_{j1},\ldots,x?,\ldots,t_{jm_j}) \ \& \ \ldots \ \&$$
$$A_k(t_{k1},\ldots,y\uparrow,\ldots,t_{km_k}) \ \& \ \ldots \ \& \ A_n(t_{n1},\ldots,t_{nm_n})$$

We preprocess conj in a similar manner except that this time each new process created has a new id and each annotated atom is replaced in conj by the term p(id) . The items conjunction of the new process contains just the annotated atom with its annotation removed. The origin of the new process is coroutined(v) where v is the annotated variable; the ancestor-vars list is that of the current process with v added. Finally, the new process is placed in a new suspension record whose var is v, and whose type and suspension-type depend upon whether the annotation is "?" or "↑".

In our example, the preprocessed conjunction is

$$\text{atom}(A_1,t_{11}. \ \ldots \ .t_{1m_1}.\text{NIL}) \ \# \ \ldots \ \# \ \text{p(ID1)} \ \# \ \ldots \ \# \ \text{p(ID2)} \ \#$$
$$\ldots \ \# \ \text{atom}(A_n,t_{n1}. \ \ldots \ .t_{nm_n}.\text{NIL}) \ \# \ \text{T}$$

and the new suspension records are

susp(x,dir(CONSUMER,CONSUMER,
pr(ID1,coroutined(x),x.ancestor-vars,T,
atom(A_j,t_{j1}. ... .t_{jm_j}.NIL) # T).NIL))

and

susp(y,dir(PRODUCER,PRODUCER,
pr(ID2,coroutined(y),y.ancestor-vars,T,
atom(A_k,t_{k1}. ... .t_{km_k}.NIL) # T).NIL))

The preprocessed conjunction pro-conj then replaces the current item and the new suspension records pro-susps are added to the suspension record set in the current node. The node list becomes

node(current-level,
pr(id,origin,ancestor-vars,done-items,new-items)
.other-processes,
new-suspension-record-set).other-nodes

where new-items is pro-conj conjoined to other-items and new-suspension-record-set is pro-susps appended to suspension-record-set.

Rule h") Current item is p(pid) .
Analogous to the old rule h).

Rule i") Current item is (conj_1 // ... // conj_n) .
Each conj_i is a conjunction. This rule introduces parallelism as follows. The current item is replaced in the current process by the term par(id_1. ... .id_n.NIL) where the id_is are new identifiers. Also, n new processes are added to the current process set:

pr(id_1,parallel(id),ancestor-vars,T,conj_1)
⋮
pr(id_n,parallel(id),ancestor-vars,T,conj_n)

In this way, a new parallel process is created for each conjunction; the parent-id of each is the id of the original process.

Rule j") Current item is par(id_1. ... .id_n.NIL) .
This means that the current process is temporarily suspended, waiting for subsidiary processes to finish. Therefore do nothing.

Rule g") A variable, for which there exists a suspension record, is bound to a non-variable term. This is the trigger for a coroutining jump or return. (Indirect suspension records are treated similarly to the previous indirect processes.)

Suppose that the variable that has just been bound is v and the suspension record for v is sr. As we saw in section 1.4, we must first discover whether the current process is a consumer or a producer of v. This depends upon whether v was annotated by "?" or "↑" and whether the current process is descended from the call in which v is annotated. The former question is answered by checking the type of sr; the latter by searching for v in ancestor-vars of the current process: if v is present then this process is descended from the annotated call. The action to be taken depends upon whether producers or consumers are currently suspended on v, as indicated by the suspension-type of sr.

Suppose that the current process is a consumer of v, i.e. type of sr is CONSUMER and v is in ancestor-vars or type of sr is PRODUCER and v is not in ancestor-vars. Firstly, the last execution step is undone, then the current process is saved in the suspended-processes list of sr. If the suspension-type of sr is PRODUCER then the producers which were suspended there must be added to the current process set and the suspension-type changed to CONSUMER.

If instead the current process is a producer of v, i.e. type of sr is PRODUCER and v is in ancestor-vars or type of sr is CONSUMER and v is not in ancestor-vars: The current process is saved in the suspended-processes list of sr. If the suspension-type of sr is CONSUMER then the consumers which were suspended there must be added to the current process set and the suspension-type changed to PRODUCER.

Our final rule deals with the event of finishing a process.

Rule e") Items is empty.
i.e. have finished a process. If the id of the current process is MAIN, we have a solution: print it and backtrack as before. Otherwise:-
If the origin of the current process is coroutined(var), then resume any processes suspended on var, delete the suspension record for var and remove the terminated process.
If the origin of the current process is parallel(parent-id), then search the current process set for the process whose id is parent-id. The current item - par(l) - of that process is replaced by par(l') where l' is the

list l omitting the id of the current process; if l' is now empty then the par(l') item is removed. Finally, remove the terminated process.

CONCLUSION

Control annotations provide a convenient means of improving the behaviour of a logic program without detracting from its clarity; given a sufficiently complex scheme of annotations we could write logic programs consisting solely of a specification, heavily annotated. However, as Hogger [14] points out, if the annotations were to become excessively intricate then programs would be difficult to write and to understand.

This last problem would be less serious if annotated programs could be compiled, shifting the complexity from the annotations into the logic. The original program - with simple logic - would serve best for declarative purposes while the compiled version - with few, if any, annotations - would be more amenable to analysis from an operational standpoint.

We have considered the simple scheme of annotations of IC-PROLOG and have seen that programs annotated in this way can often (though not always) be compiled, given a certain amount of user intervention. It is probable that other annotations that could be devised may lend themselves to compilation in a similar way. In attempting to compile annotated programs, we are using the annotations not for their intended purpose of controlling execution but to control transformation. Regarded in this way, the annotations resemble the "metaprogram" of Feather; however, the latter contains all of the information required to control the transformation, since it is designed for this purpose.

The principal advantage of compilation is of course the increased speed. In testing, on IC-PROLOG, each example appearing in this report, we have noted a typical 50 percent reduction in execution time for the compiled version over the original annotated program.

REFERENCES

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

2. Clark, K.L.
The synthesis and verification of logic programs.
Research report, CCD, Imperial College, 1977.

3. Clark, K.L.
Predicate logic as a computational formalism.
Research monograph, Imperial College, 1979.

4. Clark, K.L. and Darlington, J.
Algorithm classification through synthesis.
Computer Journal 23(1), 1980.

5. Clark, K.L. and McCabe, F.G.
IC-PROLOG reference manual.
Research report, CCD, Imperial College, 1980 (in preparation).

6. Clark, K.L. and Sickel, S.
Predicate logic: a calculus for deriving programs.
Proc. IJCAI, 1977.

7. Darlington, J. and Waldinger, R.
Case studies in program transformation and synthesis.
SRI International, 1978.

8. Dijkstra, E.W.
A discipline of programming.
Prentice-Hall, 1976.

9. Feather, M.S.
"ZAP" program transformation system primer and users' manual.
Research report 54, Dept. AI, Edinburgh University, 1978.

10. Feather, M.S.
A system for program transformation.
Transformation workshop, Harvard University, 1979.

11. Feather, M.S.
Design of a text formatter.
Transformation workshop, Harvard University, 1979.

12. Hoare, C.A.R.
Communicating sequential processes.
CACM 21(8), 1978.

13. Hogger, C.J.
Derivation of logic programs.
PhD thesis, Imperial College, 1978.

14. Hogger, C.J.
Logic representation of a concurrent algorithm.
Research report, Imperial College, 1980.

15. Kowalski, R.A.
Algorithm = logic + control.
CACM 22(7), 1979.

16. Kowalski, R.A.
Logic for problem solving.
North Holland, 1979.

17. Manna, Z. and Waldinger, R.
Knowledge and reasoning in program synthesis.
AI Journal 6(2), 1975.

18. Manna, Z. and Waldinger, R.
The automatic synthesis of systems of recursive programs.
Proc. IJCAI, 1977.

19. Schwarz, J.
Using annotations to make recursion equations behave.
Research report 43, Dept. AI, Edinburgh University, 1977.

20. Vasey, P.E.
A logic-in-logic interpreter.
MSc thesis, CCD, Imperial College, 1979.

21. Warren, D.
Implementing PROLOG - compiling predicate logic programs.
Research reports 39, 40, Dept. AI, Edinburgh University, 1977.

22. Warren, D., Pereira, L. and Pereira, F.
PROLOG - the language and its implementation compared with Lisp.
Proc. SIGART/SIGPLAN Language Conference, Rochester University, 1977.