OK

Efficient Logic Programs: A Research Proposal

by

John S. Conery
Paul H. Morris
Dennis F. Kibler

Technical Report 166

March 1981

Department of Information and Computer Science
University of California
Irvine, Ca. 92717

Portions of the research described in this proposal were supported by the UCI Dataflow Architecture Project
NSF Grant MCS76-12460

Table of Contents

List of Figures

ABSTRACT

The goal of the proposed research is to develop methods for efficient implementation of logic programs. There are two areas we wish to investigate, both of which are continuations of research conducted by members of the UCI dataflow architecture group. One aspect of the proposed research involves development of a non-von Neumann architecture for parallel execution of logic programs; preliminary work in this area is reported by Conery [9]. The second area involves transformation of high level logic specifications into efficient Prolog and/or procedural language programs, and is based on work by Morris [20].

1. CONTEXT

In 1965, J. A. Robinson [24] published the resolution algorithm, a procedure for automatic theorem proving that works for theorems expressed as clauses of first order predicate logic. Since then, other researchers (c.f. [13, 21]) have argued that a resolution proof can be viewed as a computation. Specifically, given a satisfiable set of clauses S, and a clause C which is to be proven, a resolution proof shows that the set $\{ S \cup \sim C \}$ leads to a contradiction. Moreover, the proof is constructive; that is, if C contains one or more variables, the proof process will construct the value(s) for those variables that make the set unsatisfiable. Thus, the proof that $\{ S \cup \sim C \}$ is unsatisfiable also computes values for the variables of C. In this context, the set S is known as a logic program.

Prolog [23] is a language that is based on the resolution procedure. It is quickly gaining credibility as a viable programming language. Warren, Pereira, and Pereira [27], Kowalski [17], and others cite numerous examples of large and useful programs written in Prolog.

One major advantage of using logic as a programming language is that it is a very high level language. Kowalski [17] advocates the use of Prolog as a specification language. He argues that there are many benefits obtained from being able to directly execute the logical specification of a program (not the least of which is that if the Prolog is efficient enough, there is no need

to design a procedural language version of the program). Davis [10] describes a system which creates LISP or Pascal programs from Prolog specifications.

One factor that will determine the ultimate success of logic programming is the difficulty of learning to program using the clause form of logic. Kowalski [15] describes some of the difficulties of trying to express certain relationships in clause form as opposed to the more familiar first order predicate calculus. This style of programming, which can be called relational, is quite different from two other common styles of programming, namely procedural (as typified by programs written in Pascal, FORTRAN, or PL/I) and functional (FFP [3], LISP, SASL [26]).

Another key to the acceptance of logic programming is the efficiency of logic programs. Warren, Pereira, and Pereira [27] found that programs written in Prolog and LISP required about the same amount of time to execute. Kowalski [17] cites an article which claims that compiled Prolog is as efficient as "well structured" Pascal.

Improving the efficiency of logic programs is the major goal of our proposed research. Among the various strategies are

- Improved resolution algorithms. There has been quite a bit of research in this area (c.f. [25, 14]), with the emphasis on improved control strategies over the choice of resolvents. We do not propose to do any work in this area.

- Parallel execution of logic programs. Section 2 of this paper is a discussion of current work in this area, and our proposal for further research.

- Optimization of logic programs. In section 3 of this paper we present an alternative functional notation for logic programs and a scheme for transforming expressions in this notation into efficient programs.

The next two sections are introductions to topics which are related to our proposal. Readers who are familiar with these topics are invited to skip the corresponding sections. The first is a description of the Prolog language. Our proposals for multiprocessor architectures in section 2 are related to dataflow architectures, so the second is an introduction to the dataflow system upon which our proposals are based -- the UCI Dataflow Architecture.

1.1. Prolog

A Prolog program consists of a set of clauses. There are two kinds of clauses -- implications and assertions. Implications are of the form

```
w :- x, y, z.
```

where w, x, y, and z are predicates. w is the head of the clause, and those predicates to the right of the :- symbol make up the body of the clause. Assertions are clauses that have an empty set of predicates for the body:

```
w.
```

In most cases, predicates have arguments, which are enclosed in

parentheses after the predicate name. These arguments are called terms, and can be atoms, variables, or structured objects. The names of atoms begin with lower case letters, and names of variables (which take values of either atoms or structures) begin with upper case letters. The syntax for structured objects is the same as that for terms. The "scope" of a variable is a single clause; if X appears in two different clauses, it will not necessarily refer to the same object. Variables in Prolog are completely different from variables in most other languages, where they are names of memory locations that hold values.

A Prolog program is activated by giving it a goal statement

```
:- x, y.
```

The system tries to solve the goals in order, from left to right. To solve a goal, the system looks for a clause whose head unifies with, or matches, the goal. Two predicates can be unified if there is a substitution for variables that makes the predicates identical (see [22] for a definition of a unification algorithm). For example, the two predicates

```
p(X,a),p(b,Y)
```

can be unified by the substitution {X/b,Y/a}, meaning substitute "b" for X and "a" for Y, to give the single predicate p(b,a). When the system unifies a goal and the head of some clause, the unifying substitution is applied to the entire goal list and the body of the clause, and this new body now replaces the original

Prolog program:

```
      clauses                                  comments

(1) father(curt,elaine).          /* curt is the father of      */
                                  /*   elaine (an assertion)    */
(2) father(dan,pat).
(3) father(pat,john).
(4) mother(elaine,john).
(5) grandfather(X,Z) :-           /* X is the grandfather of Z  */
      father(X,Y),father(Y,Z).    /*   if there exists a Y such */
                                  /*   that X is the father of Y*/
                                  /*   and Y is the father of Z */
(6) grandfather(X,Z) :-
      father(X,Y),mother(Y,Z).
```

Execution 1: find G such that "dan" is the grandfather of G.

step	goals to be solved	matching clause	unifying substitution
[1]	:- grandfather(dan,G)	5	{X/dan}
[2]	:- father(dan,Y), father(Y,G)	2	{Y/pat}
[3]	:- father(pat,G)	3	{G/john}
[4]	{empty}		

answer: G = john

Execution 2: find G such that "john" is the grandson of G.

[1]	:- grandfather(G,john)	5	{Z/john}
[2]	:- father(G,Y), father(Y,john)	1	{G/curt, Y/elaine}
[3]	:- father(elaine,john)	none; backtrack to [2]	
[4]	:- father(G,Y), father(Y,john)	2	{G/dan, Y/pat}
[5]	:- father(pat,john)	3	{ }
[6]	{ empty }		

Note: if this answer, G = "dan", is rejected, the system will backtrack and find another substitution for step [2]. These will all fail, and eventually step [1] will be redone using clause (6), and the second answer, G = "curt", will be produced.

Figure 1-1: Example of a Prolog Program

goal at the front of the goal list. Note that under this interpretation, predicates in the body of a clause can be considered subgoals, i.e. one can read the statement

```
w :- x, y, z.
```

as "in order to solve the goal w, solve the goals x, y, and z in that order." Assertions (goals with no subgoals) always succeed.

If the current goal does not unify with the head of any clause, it fails, and the system backtracks by undoing the latest unifying substitution, and trying another match for the most recently matched goal.

When the goal list is empty, the system stops and prints the values obtained for any variables that were in the original goal.

Figure 1-1 contains an example of a Prolog program, and shows the series of steps carried out when the program is given various goals to solve.

References [23], [17], [8] and [13] offer a variety of other descriptions of logic programming and the Prolog language.

1.2. Dataflow

The UCI Dataflow Architecture project has three distinct levels. At the most abstract level is an applicative language, Id (for Irvine dataflow). Id programs are translated into a graph-like base language, and at the lowest level a multi-microprocessor architecture interprets base language

programs. The following (extremely brief) introduction contains a simple Id program and the corresponding base language program.

Id programs are essentially multivalued expressions. The following program computes the two roots of a quadratic equation $Ax^2 + Bx + C$:

```
r1, r2 <- ( x <- sqrt(b↑2 - 4*a*c);
            y <- 2*a
            return (-b+x)/y, (-b-x)/y  )
```

The base language translation, which is also known as a dataflow schema, for this program is shown in figure 1-2. There is one "box" for every function in the expression. The variables of the program correspond to labels on the arcs connecting the boxes. Constants are represented by constant functions, i.e. the constant C is represented by a function f such that f(x) = C for any x.

This base language program is interpreted as follows. Values are represented as tokens, which flow along the arcs connecting boxes. The tokens flow into the input ports of function boxes. If a token comes to a fork, where the arc splits into two or more arcs, it is replicated, and copies of the token travel down each of the successor arcs. When tokens are present at all of the input ports of a function box, the box absorbs the tokens, computes a value, and emits the result as a new token at the output port of the box. This application of functions when and only when all input values are available is known as data driven

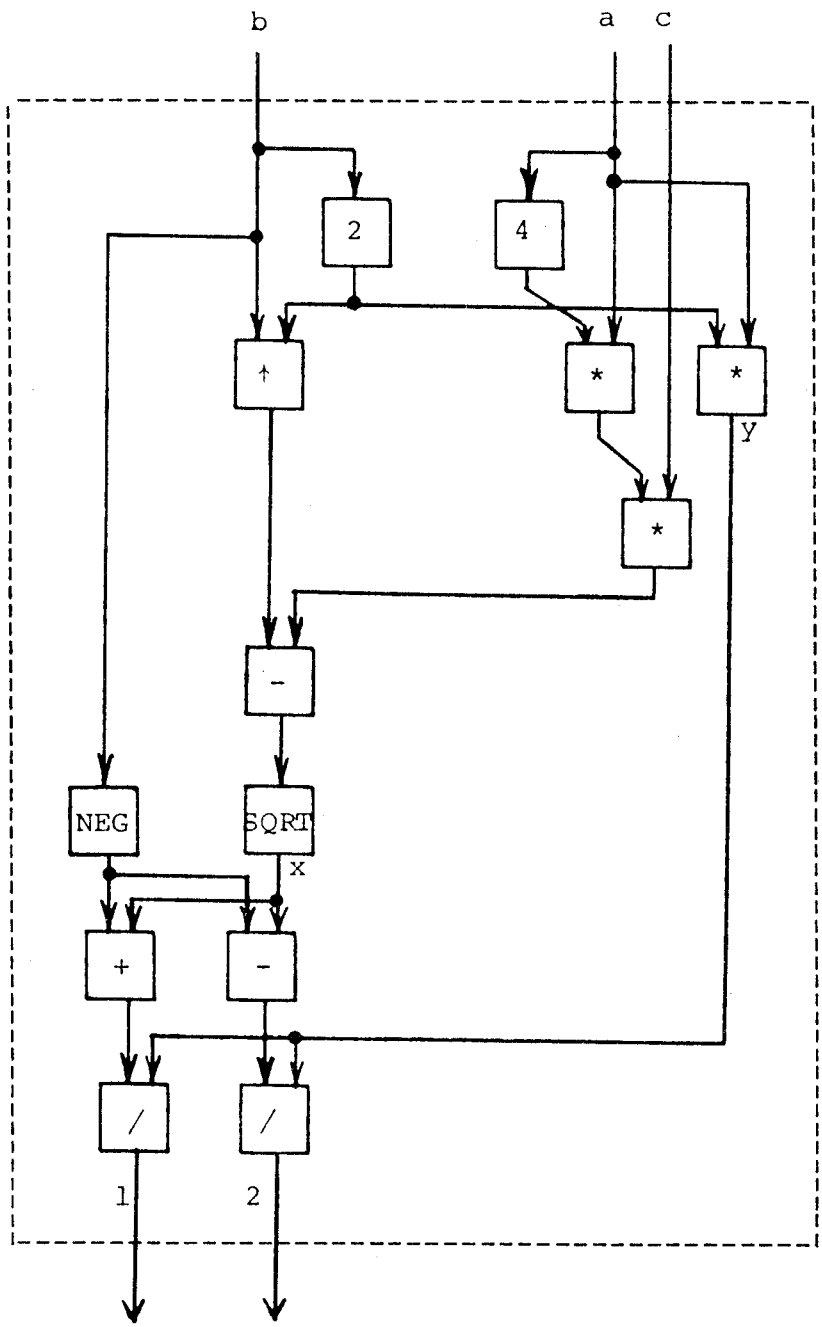


Figure 1-2: Id Schema

Base language form of a program that computes the roots of a quadratic equation. Reprinted, with permission, from Arvind, Gostelow, and Plouffe [2].

computation.

It is important to note that Id variables are the names of values, not memory locations that hold values. There is no concept of a memory cell in Id, and thus no way for a function to generate side effects, or in any manner interfere with other functions (other than simply producing and consuming values). This makes it possible to achieve a high degree of parallelism in the architecture.

We have necessarily omitted a great deal of the Id language and corresponding base machine constructs. Refer to Arvind, Gostelow, and Plouffe [2] for a more complete description of Irvine dataflow.

2. PARALLEL EXECUTION OF LOGIC PROGRAMS

Nilsson [22] describes the resolution algorithm as a production system, and cites work by Zisman [28] as a possible control regime for parallel application of production rules.

In the following sections we will present Nilsson's production system model of resolution and Zisman's control strategy. We do this not because we propose to follow this line of research, but because this discussion exposes a major difficulty. Following that we will present two completely different approaches to parallel execution of logic programs.

2.1. The Production System Model of Resolution

A production system consists of three basic components: a set of rules (each of the form P -> A), a global memory, and a control strategy. The P in each rule is a predicate, and the A is an action. When the system is in operation, the predicates of the various rules are evaluated. If a predicate is true, the corresponding action can be carried out. Among the various kinds of actions are modifications of the memory, addition/deletion of rules, and so on. The particular control strategy is independant of the specification of the rules, and determines such things as the order in which rules are scanned when looking for a true predicate, and the overall structure of the memory (e.g. memory may be organized as a stack, with only the most recently inserted item being accessible).

In a backward chaining production system model of resolution theorem proving, the clauses of the logic program are the rules of the production system, i.e. a clause x :- y, z corresponds to a rule x -> y, z. A goal to be solved is the only thing contained in the memory when the production system is started. The system operates by scanning every rule for one whose predicate matches (can be unified with) a goal currently in the memory. If a match is found, the unifying substitution is applied to the entire contents of memory and the action part of the selected rule; then the action part of the rule is inserted into memory.

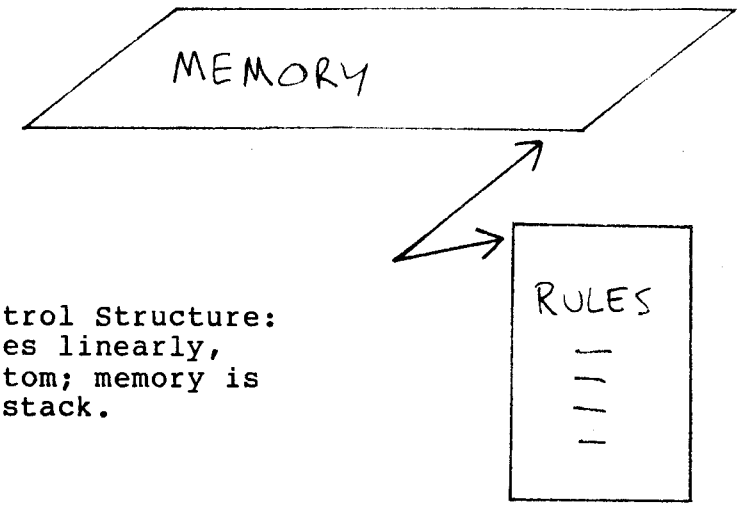
The control strategy for this production system has two components: the order in which rules are scanned when searching for a match, and the process which decides where in memory new actions are inserted. The control structure for Prolog, as was mentioned in section 1.1, organizes memory as a stack, where new actions (subgoals) are pushed onto the stack, and the new top of stack becomes the goal that is to be matched next. Rules are scanned linearly from first to last.

2.2. Parallel Application of Rules

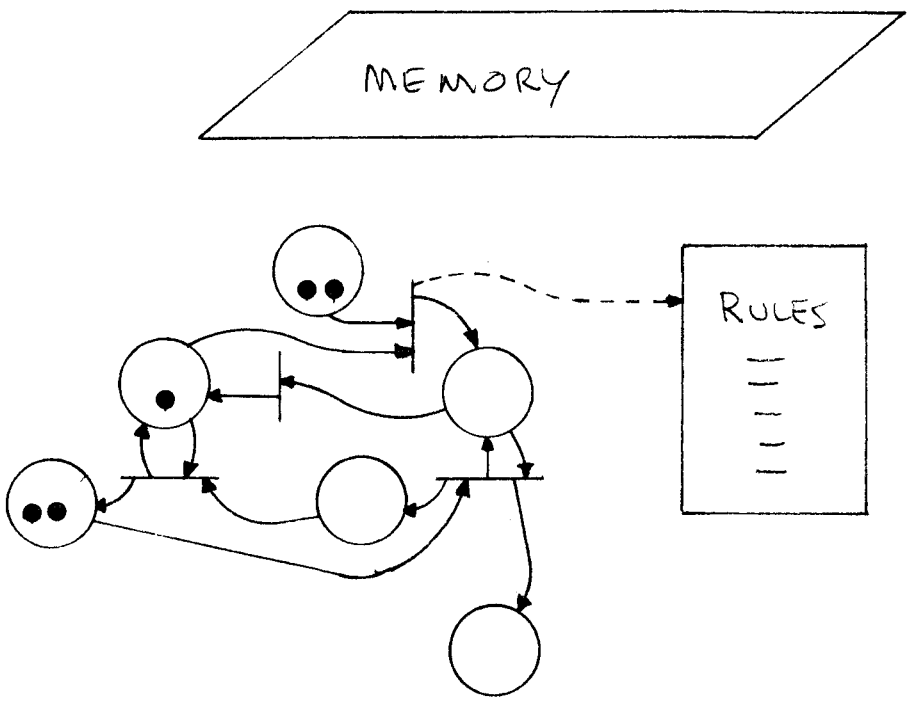
Zisman [28] describes the use of a production system in modeling the actions of a Petri net. Nilsson [22] suggests that this scheme can be adapted for use as a parallel control structure for production systems, and thus, by extension, for logic programs. Figure 2-1 shows this pictorially, where the linear scan of rules has been replaced with a correspondence between the transitions of a Petri net and the rules of the Prolog program.[1] The idea is that any transition that is enabled to fire "points at" a rule that can be applied. When any two or more transitions are enabled, they can all fire at once, and

---

[1] This replacement of one control structure by another is consistent with Kowalski's observation [16] that algorithms have two major components: the logic component L and the control component C (symbolically, A = L + C). In the context of logic programming, the logic component is the set of axioms that make up the program. If one wants parallel execution of logic programs, it is only necessary to exchange Prolog's "depth-first" search strategy with some method of applying two or more rules simultaneously.



Prolog Control Structure:
scan rules linearly,
top to bottom; memory is
a stack.



Petri Net Control Structure: transitions point to rules; enabled transitions correspond to rules that can be applied; organization of memory not specified.

Figure 2-1: Control Structures for Production Systems

therefore the rules pointed to can all be applied at the same time.

The major drawback to using this scheme for parallel execution of logic programs stems from the fact that the control structure of any resolution proof necessarily involves the unification procedure. In order for a rule to be applied, its predicate must unify with some goal already present in the memory. Furthermore, the unification affects not only variables in the matched goal, but all other occurances of those same variables, no matter where they occur in memory.

When two rules are applied simultaneously, the corresponding unifications may generate conflicting values for variables. For example, suppose the current goals in memory are

```
f(X,Y), g(Y), h(Y,Z)
```

and that among the rules are

```
f(X,0).
h(3,4).
```

Using Prolog's control structure, the system would be looking for a rule whose predicate unifies with f(X,Y). That rule is f(X,0), the unifying substitution is {Y/0}, and the new goals would be

```
g(0), h(0,Z)
```

If h(Y,Z) happened to occur before f(X,Y) in the goal list, the

rule to be applied would be h(3,4), and the new goals would be

f(X,3), g(3)

Using a hypothetical control structure that would enable both f(X,Y) and h(Y,Z) to be solved simultaneously, the substitution {Y/0} would conflict with {Y/3,Z/4} since one produces the goal g(0) and the other produces g(3).

It is quite clear that the source of this problem is that values are communicated among subgoals via a common, shared memory. It is precisely this problem -- coordinating access to common memory among processes executing in parallel -- that is avoided in data driven systems. What we propose, therefore, are data driven models of computation for logic programs.

This analogy is shown pictorially in figure 2-2, which shows the current status of research in languages and computer architecture as we see it. This figure shows two dimensions for describing a programming language: language style (relational, functional, procedural) and underlying architecture (von Neumann, non-von Neumann).

Square 1 represents the vast majority of language and architecture research in computer science to date: procedural languages for von Neumann machines. Languages in this square include older languages such as ALGOL, as well as newer languages such as Ada. There have been various attempts at using traditional languages on multiprocessor machines (e.g. FORTRAN

Underlying Architecture \ Language Style	procedural	functional	relational
von Neumann	1 ALGOL, FORTRAN, Pascal, Ada	2 SASL, LISP	3 Prolog
non von Neumann	4 FORTRAN for ILLIAC IV	5 Id, FFP	6 ?

Figure 2-2: Research in Languages and Architectures

for the ILLIAC IV); these are part of square 4.

Squares 2 and 5 represent, respectively, the functional languages mentioned before (LISP, SASL) and architectures designed specifically for a given functional language (e.g. Id [2], Mago's machine for FFP [18]).

Square 3 represents Prolog, a relational language with a control structure that makes it suitable for use on a single processor machine.

The final square represents the general area of our proposal for parallel execution of logic programs -- development of non-von Neumann multiprocessor architectures for executing logic programs.

2.3. Proposed Research

We have two independent proposals for parallel architectures for logic programs. One is a machine that would expand Prolog programs in a breadth-first search on independent processors (we call this the Parallel Prolog Machine, or PPM); the other is a relational dataflow machine. Each will be discussed in detail below.

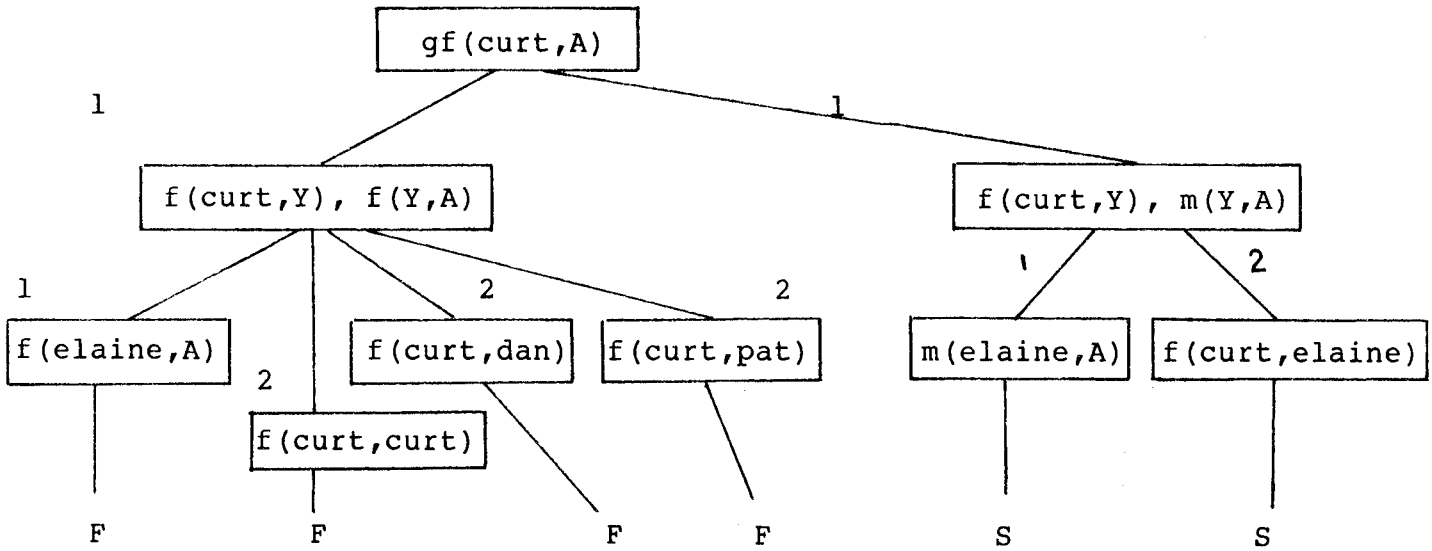
At this time we do not intend to actually build a piece of hardware. This proposal covers only the first phases of architecture design, namely specification of a "paper machine", followed by simulation studies.

2.3.1. A Breadth-First Prolog Machine

Figure 2-3 shows a tree in which each node is a possible state of a Prolog computation (by state we mean the current list of unsolved subgoals). This example uses the same computation that was performed in figure 1-1.

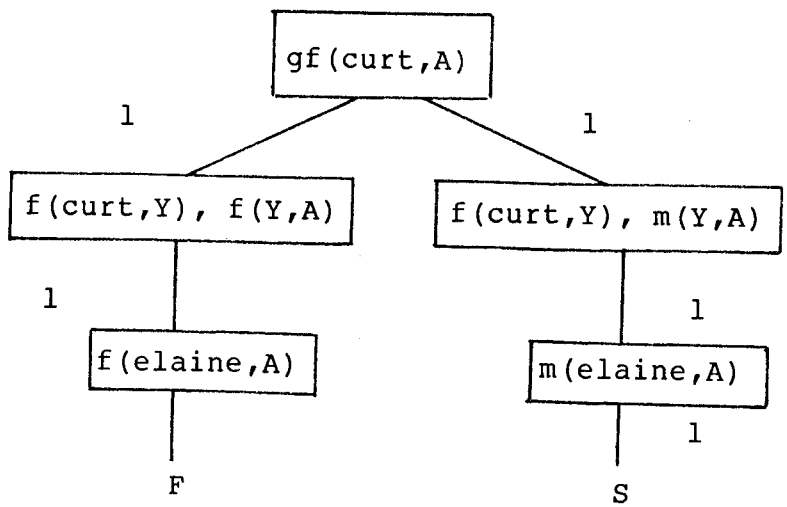
The tree is constructed by starting with the main goal as the root. A descendent of a node is created by unifying some goal in the current list with the head of some clause in the program. In other words, generation of a descendent corresponds exactly to one unification and replacement step in the standard Prolog system, where the new goal list is the descendant.

It may be possible to perform more than one unification. For example, in the first step of computation 1 of figure 1-1, the



The numbers labelling arcs refer to the goal within the parent that is used to create the descendant

Figure 2-3: Maximum Breadth-First Goal Tree



A less ambitious strategy for expanding nodes uses only the first goal in the current list (i.e. every label is 1).

Figure 2-4: Breadth-First Goal Tree

goal

grandfather(dan,G)

could have been unified with either clause (5) or (6). In such situations, all unifications are applied, and each one creates a descendant in the tree. In the proposed architecture, sibling goal lists would be processed in parallel by independant processors.

There are two possible ways to stop the growth of the tree along any path from the root:

1. If there is no substitution that can unify the head of any clause with any goal in the node currently being expanded, then the current node is a failure node and its only descendant is marked F.

2. If a substitution and replacement step results in the empty set of clauses, the node is a success node and its only descendant is marked S.

The sequence of substitutions on a path from the root node to a success node provides the solution to the computation. If the root clause contains a variable X, the first substitution for X that is encountered on the path from the root to the success node is the computed value for X.[2] Notice that where DECsystem-10 Prolog produces multiple answers for a single goal through

---

[2]More accurately, since the first substitution might only "partially bind" X by assigning it a structured term that itself contains variables, the value of X is given by the composition of substitutions labeling the path from the root to S.

backtracking (c.f. computation 2 of figure 1-1), the PPM produces the same answers in parallel, along different paths and on different processors.

The tree of figure 2-3 exhibits the maximum amount of parallelism possible. This can lead to a large number of unsuccessful paths in the tree, with a corresponding waste of processing. A less ambitious rule for expanding nodes is a compromise: only the leftmost goal in any list is eligible for unification, but it is unified with as many heads of clauses as possible. The tree constructed using this rule is shown in figure 2-4.

Another strategy for controlling growth is related to the control annotations of IC-Prolog (Clark and McCabe [8]). The idea here is that the most efficient ordering of predicates in the body of a clause depends on the pattern of variable instantiations in the head of the clause. Clause (5) of figure 1-1 is expressed as two clauses in IC-Prolog:

```
grandfather(X?,Z↑) :- father(X,Y), father(Y,Z).
grandfather(X↑,Z?) :- father(Y,Z), father(X,Y).
```

Here X? means that X has to be bound to a non-variable before the head can be unified with a goal, and X↑ means that the clause will produce a value for X. Thus the call

```
grandfather(dan,G)
```

matches only the first of these clauses, and

grandfather(G,john)

matches only the second. Note that the body of the second clause has the same goals as the first clause, but in the opposite order. This order is the most efficient for this pattern of variable instantiation in the head of the clause.[3]

We feel that this ordering of subgoals in the body of a clause can be done dynamically, at run-time, without requiring the programmer to explicitly use annotations. This analysis could be used in the PPM to limit the growth of the tree.

Our proposed research program for the breadth-first machine is as follows:

1. Survey existing Prolog programs, analyzing them for potential sources of parallelism.
2. Develop techniques for dynamically generating efficient parallel search spaces (where efficiency is measured by the number of processors working on successful paths in the goal tree).
3. Design the PPM, at the same level of abstraction as the Irvine dataflow base machine. This will involve specifying mechanisms for assigning goal lists to processors, implementing algorithms developed in step 2, and designing methods for gathering results at the root of the tree.
4. Develop a simulator for the PPM, written in SIMULA and running on the DECsystem-10 at UCI.

---

[3] Computation 2 of figure 1-1 is an example of this situation. This call required six steps, as opposed to the four steps used in computation 1. If the order of the subgoals were reversed, this computation would also be completed in four steps.

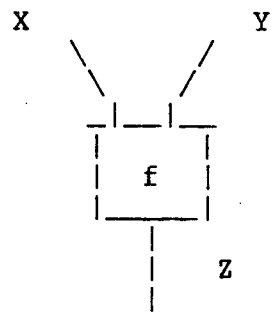
5. Design and simulate a physical architecture for the PPM. We anticipate that this architecture will not actually be a tree-shaped organization of processors, but that we should be able to use much of the work already done at this level for the UCI Dataflow Architecture Project (see Gostelow and Thomas [11]).

2.3.2. Relational Dataflow

The idea for a relational dataflow system stems from attempts to create dataflow schemata for Prolog programs instead of Id programs. What developed was a system where the high level language is first order predicate calculus, and the base language is a modified form of the Irvine dataflow base language. Reference [9] is a preliminary report on some ideas for a relational dataflow system. The remainder of this section is a summary of the key ideas from [9], and our proposals for continued research on this architecture.

High level language programs are well formed formulae (wffs) of first order predicate calculus. The graph form that these wffs are translated into is very similar to the Id base language: function and predicate names become boxes, and variables are labels on lines connecting boxes. Another similarity is that values are communicated via tokens, and that when a box has enough tokens, it "fires" and creates new tokens for results.

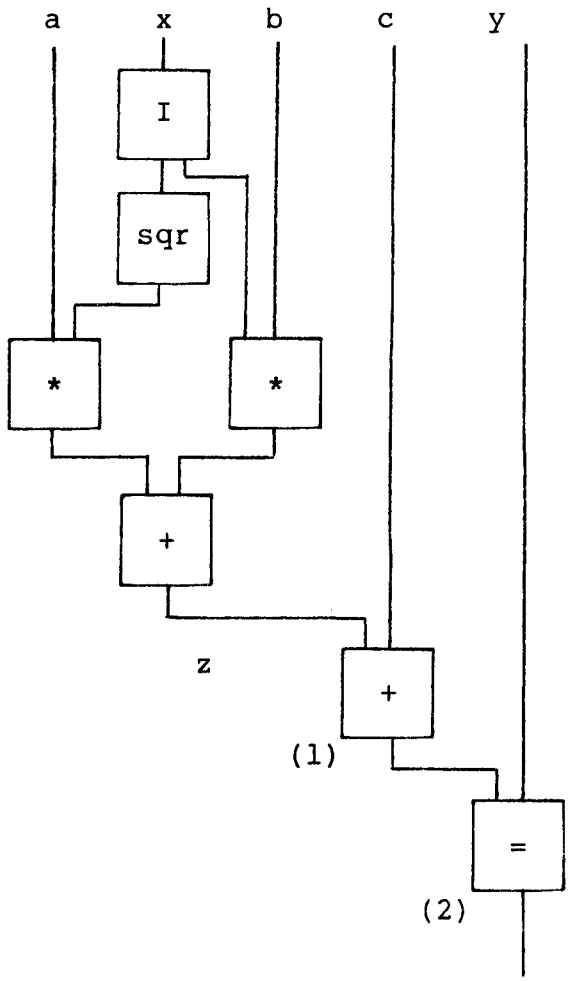
Boxes in the relational dataflow system represent relations, and not functions. The difference is illustrated by the folowing example.



In Id, tokens always flow along lines X and Y; when both are present, f can be applied, and the value of f(X,Y) sent out along line Z. In the relational dataflow system, it is possible for tokens to move "backwards"; it is possible for tokens to arrive at the box on lines Z and Y. If this happens, the box absorbs the tokens, and computes a value for X, such that f(X,Y) = Z. In other words, the box works to preserve the relationship Z = f(X,Y).

Figure 2-5 shows a simple wff, its corresponding graph, and an explanation of a computation of values for one of the variables of the wff.

Some functions in relational dataflow are non-strict functions, meaning that values of their outputs can be computed before all inputs are present. An example is conjunction, Z = Y /\ X. If Y is known to be false, then the box can absorb the token carrying Y and emit a false for Z. This also works for tokens moving backwards. If a true token arrives at Z, true tokens can be generated for both X and Y (since only T /\ T = T).



Relational dataflow schema for the expression $ax^2 + bx + c = y$. If values for x, a, b, and c are known, a system can compute c as follows: tokens for x, a, and b are used to compute, in the normal manner, a value (call it z) for the top left port of the box labeled (1). A token carrying the value of y arrives at the top of box (2), and a token with a True value goes to the bottom of this box. In order to preserve the "=" relation, this box copies the value of y and sends it to the bottom port of box (1). Box (1) now computes y - z for the value of c.

This schema cannot be used to compute a value for x, given values for a, b, c and y (i.e. it is not equivalent to a schema for the quadratic equation); see [9].

Figure 2-5: Relational Dataflow Program

The relational dataflow system is similar to the PPM in that it is easy to waste processing power on irrelevant computations. One way to control this is to use demand tokens, which are sent from a function F to another function G when F requires some information that can be produced by G. This means that G will never compute values unless it is known that these values will be used. Keller, et al [12] describe an implementation of demand-driven computation, and [9] describes how demand tokens could be used in the relational dataflow system.

Our research program for the relational dataflow system is:

1. Design a high level logic language, programs of which would be translated to run on the base machine. Required extensions to first order predicate calculus include data structures and user defined functions. There should be some means of specifying assertions, as there is in Prolog (so that the system could interpret programs such as the "grandfather" example).

2. Define the base machine. Further work in this area includes a complete definition of the rules for using demand tokens, and providing a means for implementing some of the nondeterminism of Prolog (especially the ability to produce more than one answer from one goal).

3. Design and implement simulators for both the base machine and a lower level physical machine (this step is similar to items four and five for the tree machine).

4. The tree machine has a very sound and well understood theoretical basis, namely resolution theorem proving. However, the relational dataflow machine is "breaking the rules" of Id and other dataflow systems by having tokens move in all directions. Part of our research effort will go toward defining how/why/what a relational dataflow system is capable of computing.

3. OPTIMIZATION OF LOGIC PROGRAMS

3.1. Objective

The goal of this research segment is to attain a certain level of automatic programming, using logic programs as an intermediate language for manipulation. Specification will be in a high-level declarative language resembling mathematical notation. Conversion to efficient code will proceed in four stages:

1. Straight-forward translation to a (probably inefficient) logic program.
2. General transformations to remove some well-defined types of inefficiency, using the dataflow network[4] representation of logic programs [20].
3. Further optimization based on runtime information.
4. Straight-forward translation to a conventional language.

3.2. Initial Design and Methods

3.2.1. Specification Language

Logic programs, while well suited to manipulation because of their simple uniform structure, do not constitute an ideal specification language. The main shortcoming is the restriction of pattern match to simple patterns involving the primitive

---

[4] This version of dataflow is oriented towards manipulation, rather than direct implementation, and differs somewhat from both Irvine dataflow and the relational dataflow system considered earlier.

constructor functions. Specification is made easier by the "bootstrapping" approach of allowing patterns to be based on defined functions. This consideration leads to a language where one may define multivalued[5] functions, as well as relations, by means of what we call modulated recursion equations (MREs). These resemble the clauses of logic programs except for the use of

1. equality and functional notation.
2. patterns involving defined functions in the clause head.

Example of 1: SORT may be defined by

```
sort(X)=Y / perm(X)=Y, ordered(Y).
```

The "/" may be read as "modulo" or "provided that." Compare this with the clause

```
sort(X,Y) :- perm(X,Y), ordered(Y).
```

Example of 2: Suppose now that the usual APPEND function on lists has been defined. We wish to define a multivalued MEMBER function that returns any member of a list. The definition is given by the following MRE:

```
member(append(Y,cons(X,Z)))=X.
```

---

[5]Here we use "multivalued" in the sense of having multiple alternative values. This is the same usage as in the mathematical theory of functions of a complex variable.

Here CONS is the usual primitive list operation, as in LISP. Note that "/" does not appear here since no modulation is needed. In this example, the pattern append(Y,cons(X,Z)) may match the input list in multiple ways, giving alternate values for X. Another example:

```
fact(0)=1.
fact(N)=times(N,fact(subl(N))) / greater(N,0).
```

defines the factorial function.

Modulated recursion equations are closely related to the recursion equations of [7]. The language also has much in common with SASL [26], and with the specification language of Manna and Waldinger [19].

It is not difficult to see that there is a simple systematic way of transforming MREs to equivalent logic program clauses. For example, the definition of MEMBER translates to

```
member(X,W) :- append(Y,XZ,W), cons(X,Z,XZ).
```

Notice, however, that the resulting logic program is inefficient because the list Y is created unnecessarily. This kind of inefficiency is typical in the translation of the most naturally defined MREs. We will discuss methods below for removing these inefficiencies.

The surface form of the specification language can be made even more natural by a judicious use of devices like infix notation

and ellipsis. Thus, the MEMBER definition could be rendered as

```
member([...,X,...])=X.
```

3.2.2. Transformation Methods

Certain inefficiencies can be readily detected and removed by exploiting the dataflow representation of logic programs. The MEMBER predicate above can be optimized by this method. However, we will use a more interesting example to illustrate the technique. We define a multivalued function EMBED by

```
embed(X,[...,...]) = [...,X,...].
i.e.  embed(X,append(Y,Z)) = append(Y,cons(X,Z)).
```

(Thus embed(A,[B,C]) has [A,B,C], [B,A,C] and [B,C,A] as alternative values. Notice that MEMBER could now be defined as member(embed(X,Y)) = X.)

A straightforward translation gives the logic program clause

```
embed(X,U,V) :- append(Y,Z,U), cons(X,Z,XZ), append(Y,XZ,V).
```

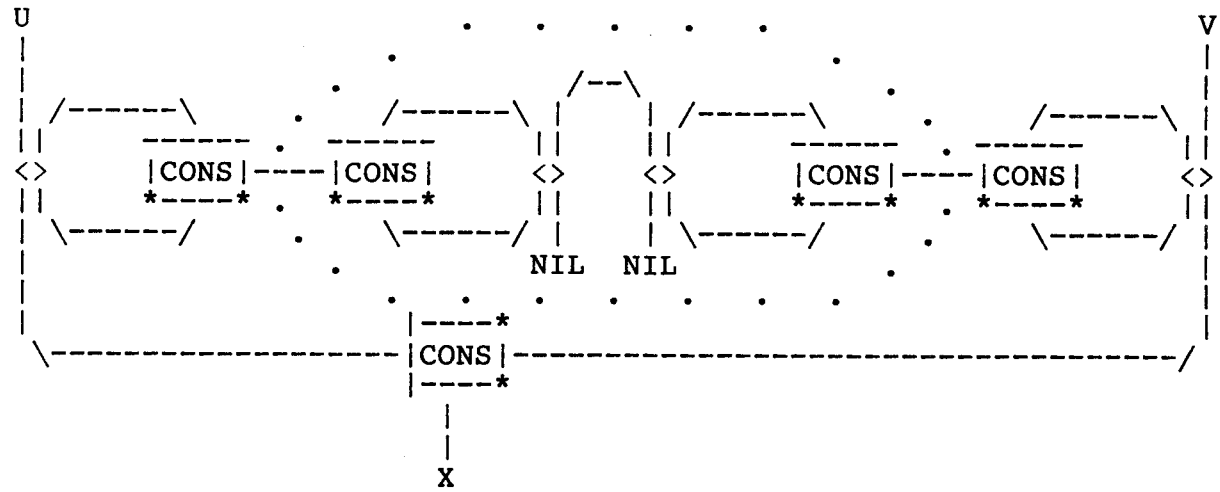
Let APPEND be defined by the logic program

```
append(nil,Q,Q).
append(P,Q,R) :- cons(P1,P2,P), append(P2,Q,R1), cons(P1,R1,R).
```

These definitions correspond to the network[6]

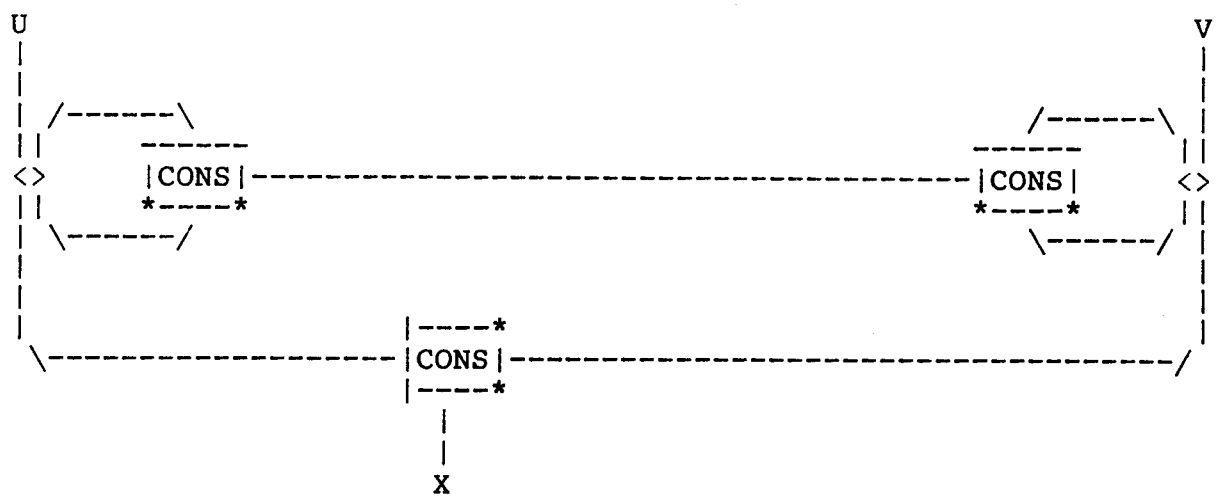
---

[6]The networks have been slightly simplified since [20] by allowing loops to be entered from both the top and the bottom. This avoids the need for elements whose sole purpose is to transport values to the bottom of loops.



In this diagram the CONS boxes are oriented, with the CONS value at the side marked with asterisks, the CDR value at the opposite side, and the CAR value at right angles to these (see figure 3-1).

The region inside the dots reduces to the identity function, giving the simplified network



which corresponds to the efficient logic program

```
embed(X,U,V) :- cons(X,U,V).
embed(X,U,V) :- cons(Y,U1,U),cons(Y,V1,V),embed(X,U1,V1).
```

We hope eventually to apply this technique in all situations where a computation does internal work that is unused for the output. Discovery of these simplifications can be facilitated by constructing an example computation and annotating the lines of the dataflow network with the resulting streams of values. The coincidences noted suggest candidate simplifications.

3.2.3. Using Runtime Information

Many inefficiencies which are obscure in a program are easily seen in a runtime trace. This suggests the following plan:

1. Run the program on example input, saving the computation trace.
2. Convert the trace to a straight-line program, equivalent to the original program on the given input.
3. Apply known techniques for optimizing straight-line programs [1] Convert the optimized straight-line program back to a trace.
4. Infer a general optimized program from the sample trace, drawing on established methods [5, 4].

We illustrate this process with the naive REVERSE-A-LIST arising from the definition

```
reverse(NIL) = NIL.
reverse(cons(X,Y)) = append(reverse(Y),cons(X,NIL)).
```

Here NIL is the empty list.

Applying the naive algorithm to the input [A,B,C] gives rise to

the computation in figure 3-1 (shown as a bipartite graph; we have purposely omitted chronological information). Observe that the unrolled loops appear as ladder-like structures. Noticing the identities indicated by the dotted lines in figure 3-1, we can simply excise entire portions of the network, as in figure 3-2. We may attach the identified edges and bend the network into the familiar ladder shape, leading to the simpler system shown in figure 3-3 which corresponds to the efficient REVERSE algorithm. Notice that input and output are now at opposite ends of the ladder. A logic program for this computation would require an additional variable to pass the output back up.

It would seem that the process of infering the general optimized program could be aided by information retained from the original program. We plan to investigate this possibility.

It is instructive to compare figure 3-1 with the dataflow network for the same program. This is shown in figure 3-4. The correspondence in structure suggests another interpretation of the dataflow network: it may be regarded as a kind of generalized trace (over all inputs). This raises the possibility of operating directly on the dataflow network to produce the optimization previously noted.

3.2.4. Final Compilation

Although the optimized logic program could be run directly using the Prolog interpreter [23], it is worthwhile to consider the final step of translating to conventional code. This can

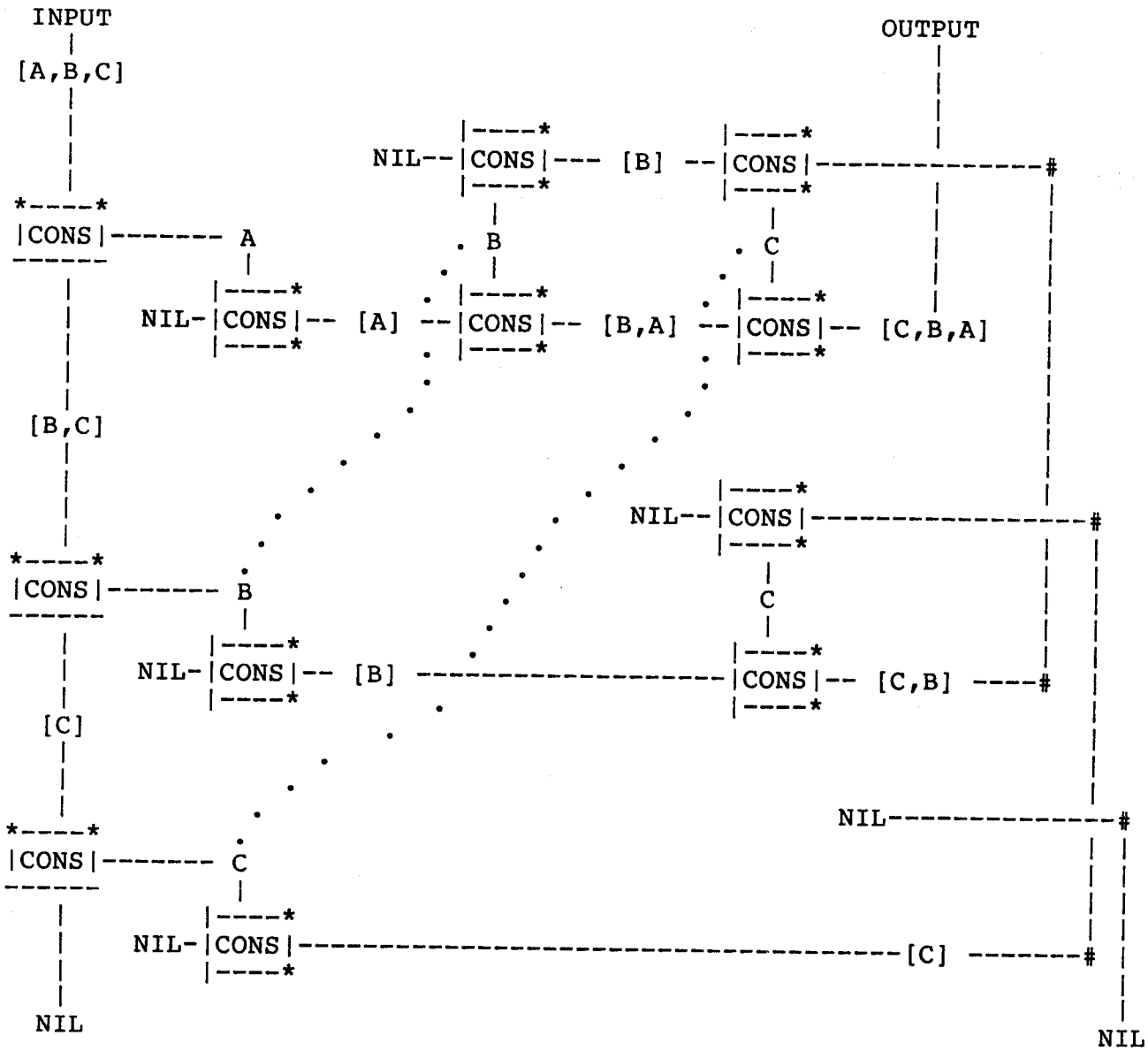


Figure 3-1: Computation Trace for Naive REVERSE-A-LIST

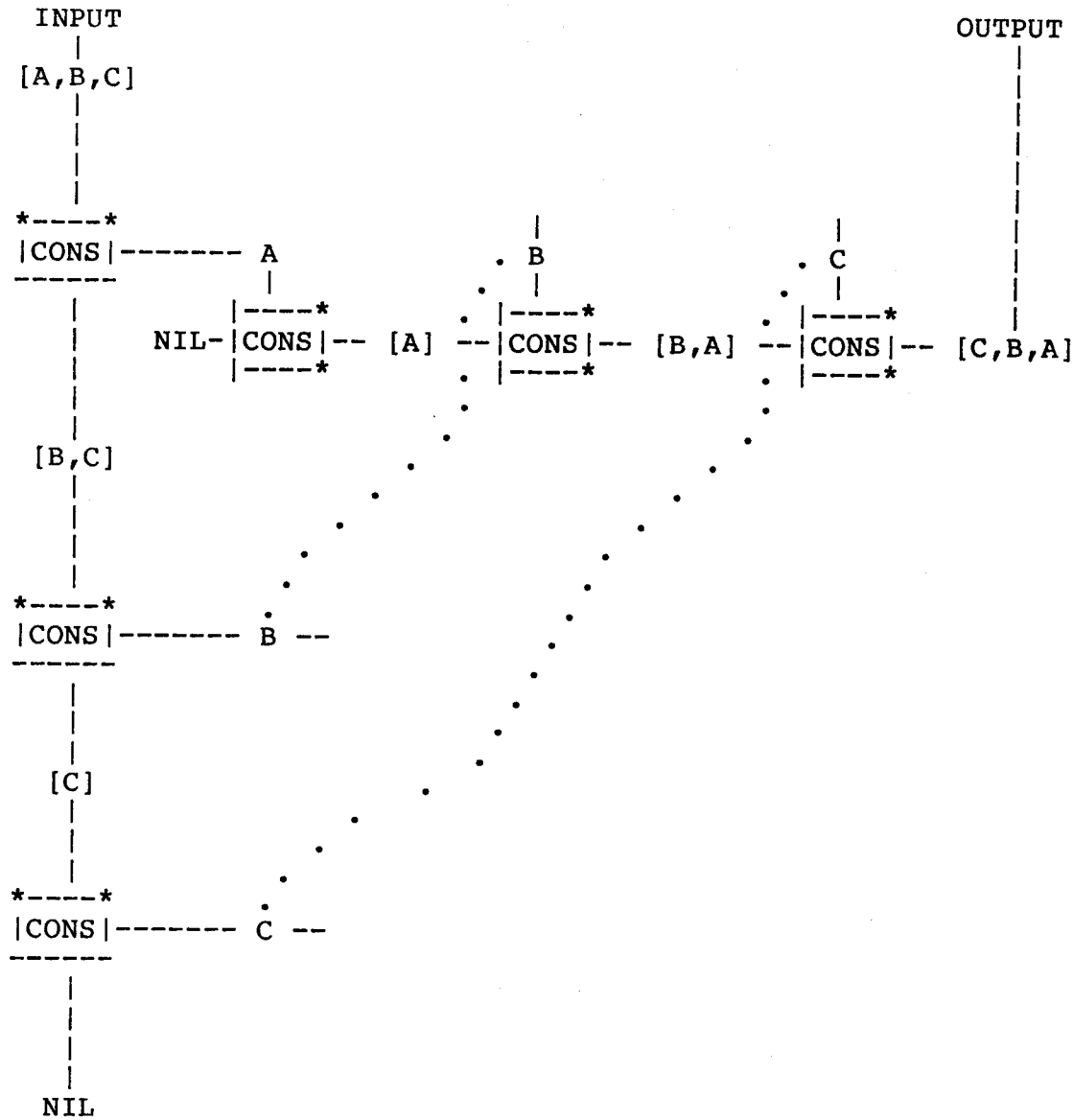


Figure 3-2: Removal of Unnecessary Relationships

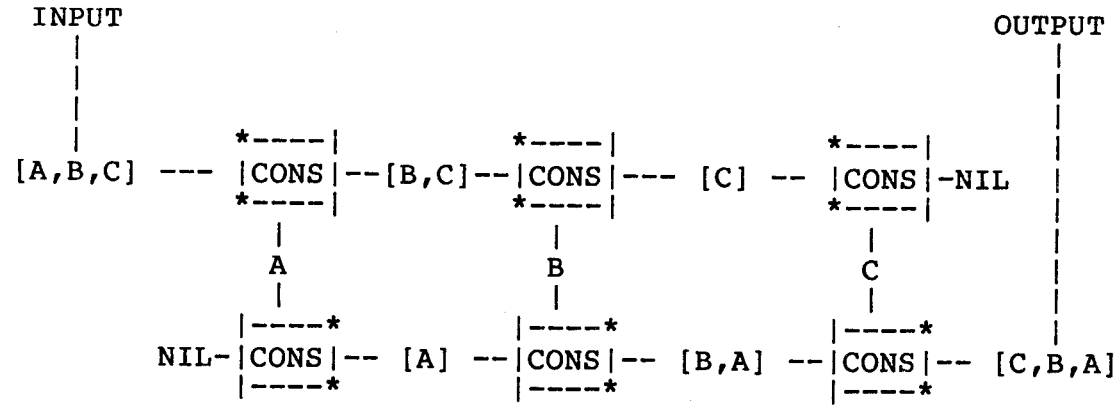


Figure 3-3: Computation Trace for Efficient REVERSE-A-LIST

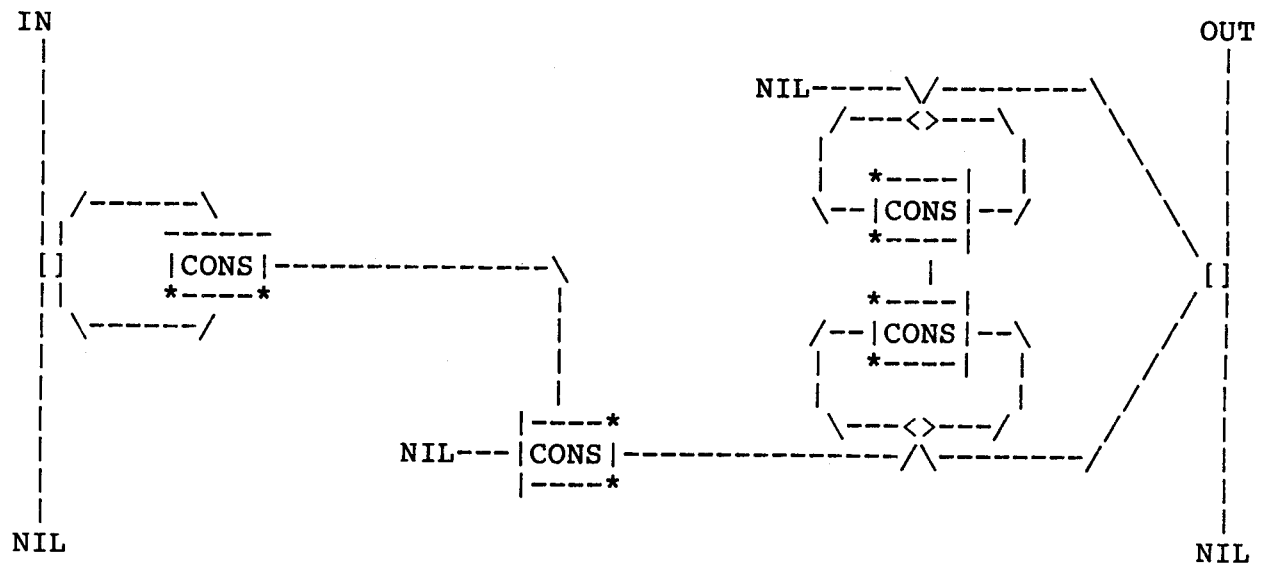


Figure 3-4: Dataflow Network for Naive REVERSE-A-LIST

best be done by identifying a restricted category of logic program that possesses a superficial translation into ordinary code. For example, only determinate logic programs would belong to this category. The problem is then pushed back to reducing logic programs to the desired form. An essential aspect of this process involves correct ordering of the computation steps, i.e. addition of control information. Runtime information should prove useful for this stage also.

3.3. Significance and Relationship to Other Work

With rapidly diminishing hardware expenses, the cost of producing and maintaining software has become the major barrier to increased computer use. In addition, more sophisticated applications have produced more complex programs that are difficult to fully understand, resulting in programmer error and unreliable systems. An obvious solution to these problems is to transfer more of the burden of producing code to the computer itself. An appropriate division of labor might be to make the computer responsible for constructing efficient code, freeing the human programmer to concentrate on logical specification. In other words, we should tell the computer what to do rather than how to do it. This is the rationale for the field of automatic programming (see Biermann [6] for a survey of work in this area).

The work proposed here is most closely related to the transformation approach [7, 19] but also draws upon inductive techniques [4, 5]. It improves previous work in the following

respects:

- It introduces a more concise and flexible model of computation.
- The transformations are guaranteed to produce improvements.
- Application of transformations can be guided by example computations.
- It is not necessary to submit information about non-primitive functions (e.g. the associativity of APPEND in the REVERSE optimization).
- So far, we haven't needed any eurekas [7].

REFERENCES

1. Aho, A. and J. Ullman. Principles of Compiler Design. Addison Wesley, 1977.

2. Arvind, Gostelow, K. P., and Plouffe, W. P. An Asynchronous Programming Language and Computing Machine. Technical Report 114a, Dept. of Information and Computer Science, University of California, Irvine, December, 1978.

3. Backus, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Comm. ACM 21, 8 (August 1978), 613-641.

4. Bauer, M. Programming by Examples. Artificial Intelligence 12, 1 (May 1979), 1-21.

5. Biermann, A. and R. Krishnaswamy. Constructing Programs from Example Computations. IEEE Transactions on Software Engineering 2 (September 1976), 141-153.

6. Biermann, A. Approaches to Automatic Programming. Advances in Computers 15 (1976), 1-63.

7. Burstall, R. and J. Darlington. A Transformation System for Developing Recursive Programs. J. ACM 21, 1 (January 1977), 44-67.

8. Clark, K. L., and G. McCabe. The Control Facilities of IC-Prolog. In D. Michie, Ed., Expert Systems in the Micro Electronic Age, Edinburgh University Press, 1979.

9. Conery, J. S. A Relational Dataflow System. Dataflow Note 48a, Dept. of Information and Computer Science, University of California, Irvine, May, 1980.

10. Davis, R. E. Generating Correct Programs from Logic Specifications. Technical Report 79-05-001, Information Sciences, University of California, Santa Cruz, May, 1979.

11. Gostelow, K. P. and R. Thomas. Performance of a Simulated Dataflow Computer. IEEE Transactions on Computers C-29, 10 (October 1980), 905-919.

12. Keller, R. M., G. Lindstrom, and S. Patil. An Architecture for a Loosely Coupled Parallel Processor. Technical Report UUCS-78-105, University of Utah, 1978.

13. Kowalski, R. A. Predicate Logic as a Programming Language. Proc. IFIPS 74, 1974.

14. Kowalski, R. A. A Proof Procedure Using Connection Graphs. J. ACM 22, 4 (October 1975), 522-595.

15. Kowalski, R. A. Logic for Problem Solving. Elsevier - North Holland, New York, 1979.

16. Kowalski, R. A. Algorithm = Logic + Control. Comm. ACM 22, 8 (July 1979), 424-436.

17. Kowalski, R. A. Logic as a Computer Language. Proc. Infotech State of the Art Conference "Software Development: Management", June, 1980.

18. Mago, G. A. A Network of Microprocessors to Execute Reduction Languages. Dept. of Computer Science, University of North Carolina at Chapel Hill, March, 1979.

19. Manna, Z. and R. Waldinger. A Deductive Approach to Program Synthesis. ACM Transactions on Programming Languages and Systems 2, 1 (January 1980), 90-121.

20. Morris, P. H. A Dataflow Interpreter for Logic Programs. Dataflow Note 50, Dept. of Information and Computer Science, University of California, Irvine, May, 1980.

21. Nilsson, N. J. Problem Solving Methods in Artificial Intelligence. McGraw - Hill, New York, 1971.

22. Nilsson, N. J. Principles of Artificial Intelligence. Tioga Publishing Company, Palo Alto, Ca., 1980.

23. Pereira, L. M., F. C. N. Pereira, and D. H. D. Warren. User's Guide to DECsystem-10 Prolog. Dept. of Artificial Intelligence, Univ. of Edinburgh, September, 1978. version 1.32

24. Robinson, J. A. A Machine Oriented Logic Based on the Resolution Principle. J. ACM 12, 1 (January 1965), 23-41.

25. Sickel, S. Variable Range Restrictions in Resolution Theorem Proving. In Elcock, E. W. and D. Michie, Ed., Machine Intelligence 8, John Wiley and Sons, New York, 1978.

26. Turner, D. A. SASL Language Manual. Univeristy of St. Andrews, St. Andrews, Scotland, 1979.

27. Warren, D. H. D., L. M. Pereira, and F. C. N. Pereira. Prolog - The Language and its Implementation Compared with LISP. ACM SIGPLAN Notices 12, 8 (1977), 109-115.

28. Zisman, M. D. Use of Production Systems for Modeling Asynchronous, Concurrent Processes. In Waterman, D. and F. Hayes-Roth, Ed., Pattern Directed Inference Systems, Academic Press, New York, 1978.