On Transformation of Logic Specifications into Procedural Programs

Janet M. Lin and J. Y. Juang
Electrical Engineering and Computer Science Department
Northwestern University
Evanston, IL 60208-3118
juang@eecs.nwu.edu

Abstract

In this paper a method for transforming a source program expressed as a set of Horn clauses into a target program in an Algol-like procedural language is presented. Compared with deterministic execution of a conventional program, the top-down procedural interpretation of a Horn clause program is inherently nondeterministic. This transformation method is aimed at removing certain type of nondeterminism in a Horn clause program via a variable-dependence analysis and a procedure formation scheme. It is assumed that the input/output mode information for each predicate in the Horn clause is known in advance and that each predicate is used unidirectionally, namely each argument in the argument list of a predicate is used either as an input or an output, but not both. Therefore, the Horn clauses treated by this method represent only a subclass of the general Horn logic clauses. During the transformation process, the method is also capable of discovering inconsistent or missing information in the source program.

1 Introduction

Horn-logic-based programming languages such as Prolog [1] and its various extensions have been gaining more and more acceptance during the past decade. The formal semantic theories and the expressive power provided by Horn logic as well as the availability of efficient interpreters for Horn clause programs allow these logic programming languages to be suitable for both software design specification and implementation.

As far as software system design is concerned, using a logic programming language as a specification language or as a basis for designing new specification languages increases the potential for automatic synthesis of programs and rapid prototyping, which are the two most significant benefits advocated by the operational approach to software design [2]. It also allows system designers to take full advantage of not only the expressive power offered by logic programming, but the execution efficiency offered by procedural languages in implementing the software. However, such advantages all depend on the availability of a transformation system for translating the logic specification into the target program.

There have been a number of transformation systems developed over the past two decades [3]. These systems differ greatly in the input languages they accept as well as their underlying techniques for performing transformation. Among them are some systems intended for transforming specifications expressed in either logic or mathematic notations. They adopted either the theorem-proving techniques, the direct application of rewriting rules to the program specification, or a combination of the two as the transformation method, and the target programs produced are mainly in LISP [4,5,6]. In this paper we present a method which is suitable for accepting a program expressed as a subclass of Horn clauses and transforming it into a target program in an Algol-like procedural language.

The paper is organized as follows. In Section 2 we describe the process for converting a Horn clause into clause canonical form. Section 3 presents a method for analyzing variable-dependence relations among variables in a clause, and Section 4 shows how each target subroutine is formed. Conclusions are drawn in Section 5.

2 Conversion to Clause Canonical Form

2.1 Nondeterminism

Compared with conventional programs, Horn clause programs executed top-down are non-deterministic in two main senses [7]. The first type of nondeterminism involves OR-related clauses which are the clauses with the same head predicate symbol and the same arity. (Note that we will be using the word 'clause' and 'procedure' interchangeably in the remaining discussion to mean the same thing. This is based on the procedural interpretation there exists for Horn clauses [7].) Each of the clauses which are OR-related can match a given procedure call, and the logic program itself does not determine how the alternative clauses will be tried. It is determined by the search strategy adopted by the theorem prover instead.

The second type of nondeterminism is that when there are more than one procedure calls needed to be executed in a clause, the order of execution is not specified by the program but rather by the execution mechanism. This is due to the fact that the literals in the body of a clause represents a set rather than a sequence of procedure calls. However, in order to map a clause to a target procedure in a procedural language, we need to decide on an explicit sequence for such procedure calls. We will show that it is possible to deduce the control-flow sequence of the procedure calls in the body of a clause when the mode information is known. The examples in this paper assume availability of mode information for each predicate. We will use 'i' to represent the input mode and 'o' for the output mode. Therefore, p(1,1,o) means that the predi-
cate 'p' will always be called with its first two arguments instantiated and the third one uninstantiated.

Only after nondeterminism has been successfully removed can each rule cluster be translatable into a subroutine in any conventional language such as C, PASCAL, etc. However, since recursion is an important feature in logic programming, it is preferable that the target language supports recursion. Converting a recursive construct into an iterative construct is nontrivial although it is not impossible.

Note that in the examples presented throughout this paper, the syntax assumed is basically that of Prolog [I], but with some of the syntactic restrictions relaxed. Most notably, more complex mathematical expressions are allowed. This relaxation does not pose any problem during the translation; nevertheless, it should greatly enhance the expressive power of logic clauses in problem solving. Extra logics presents no problem to the translator, either. How-ever, new set of restrictions may have to be introduced to warrant the compilability of resulting specification. So far, we haven't found major restrictions that are needed by the proposed method. Exact scope of the compilability of the proposed method is under study.

2.2 Clause Canonical Form

The first step in the transformation process is to convert each clause in the logic program into Clause Canonical Form.

Definition 1 (Free Variable Terms) A term occurring in the argument list of a literal is a free variable term if (1) it is of the variable type and (2) it is a distinct symbol in the argument list.

For example, X is a free variable term in p(X,Y), but not in p(X,X). A free variable term implies that it is not bound to any other variable in the argument list, thus can be substantiated with any value without affecting the values of other variables in the same argument list.

Definition 2 (Clause Canonical Form) A clause is in canonical form if all terms in the argument list of each literal are free variable terms.

For example, the following clause is in canonical form

\[ \text{gcd}(X,Y,D) : \neg X < Y, Y1 := Y - X, \text{gcd}(X,Y1,D) \]

but not the following one

\[ \text{gcd}(X,X,X). \]

neither is the one below

\[ p(X,f(Z)) : \neg q(X,W), r(X + W, Z). \]

2.3 The Conversion Process

The canonical form of a clause can be obtained by replacing each constant or complex term in each literal by a variable symbol which does not already exist in the clause. One or more literals are then added to the clause body to quantify the newly introduced variables. The newly introduced literals can take on the form of either a condition test or an assignment. When a new variable is introduced to replace a constant or a complex term in a head literal, the form of the literal depends on whether that constant or complex term is used as an input or an output. This information is available from the mode of the predicate assumed known. If it is used as an input, it means that the term is to be matched with the actual parameter of the caller; therefore, an equation will be added to the clause body; otherwise, an assignment will be added because it represents a value requested by the caller. In contrast, the case is simpler with body literals. The new literals added will always be assignments when a new variables is introduced to replace a constant or a complex term.

When variable terms are bounded to each other in a head literal, they can be set free by replacing the variables with distinct symbols and introducing equations to the clause to record the relationship among these variables. Variables which are bound together in the argument list of a body literal can be left alone.

Complex data structures in a logic program can be built with lists. A list is denoted by [X | Y] where X is called the head, which is an element of the list, and Y the tail, which is recursively the rest of the list. List structures can be represented with the record construct in conventional programming languages. As the elements of a list can be simple or compound terms, the record structure representing a list will have one or more fields for actual data, and there will be one additional field for the pointer to the next element in the list. The equation replacement approach described above carries-over to list structures as well.

In the case that the original term is a list structure, a corresponding record structure is declared. The original term is replaced by a pointer that is made to point to the record representing the term. The variable which refers to the tail part of the list will be represented by another pointer when the variable is used inside the clause body. If an individual field of a record is referenced by a variable in the clause, we add a new assignment literal to the clause body to indicate that the variable, in fact, subordinates to a record structure instead of being an independent variable. Note that the assignment statements or condition tests (whether originally existing in or newly added to the clause) are not processed during conversion. This is because they are already in proper form for further analysis and translation. The conversion is illustrated in the following examples:

Example: Converting a clause with complex terms

The following clause consists of functors and arithmetic expressions in its literals.

\[ p(X,f(Z)) : \neg q(X^2, W), r(X + W, Z). \]

Assuming that the mode of predicate p is p([_], _), the clause will be converted into the canonical form shown below according to the above conversion rules.

\[ p(X,V_1) : \neg q(V_2, W), r(V_3, Z), V_1 := f(Z), V_2 := X^2, V_3 := X + W. \]

Notice that \( V_1, V_2, \) and \( V_3 \) are new variables introduced to replace the complex terms \( f(Z), X^2 \) and
Example: Converting a clause with bound variable terms and constants
The following clause consists of literals with bound variable terms and literals with constant terms.

\[ a(X, X, Y) : - b(X, W), c(0, W, Y). \]

The literal \( a(X, X, Y) \) denotes an 'equal' relation between its first two arguments. Therefore, when we differentiate the two by introducing a new symbol, \( V_i \) in this example, to replace the second \( X \), the equal relation has to be restored by an equation \( V_i = X \). The constant '0' in the literal \( c(0, W, Y) \) is replaced with \( V_i \). If the modes of predicates \( a, b, \) and \( c \) are \( a(1, 1, 1), b(1, 1), \) and \( c(1, 1, 1) \) respectively, the canonical form of this clause then becomes

\[ a(X, V_i, Y) : - b(X, W), c(V_i, W, Y), V_i = X, V_2 := 0. \]

Example: Canonical form of a clause with a list of records
The following clause recursively manipulates on the first element of a list of records called "Conditions". Each record consists of three data fields: Type, Test and Rating. These components are referenced in the body of the clause.

\[
\text{verify } (\text{condition}(Tp, Test, Rating)|\text{Conditions}, \text{Profile}) \\
\quad := \text{scale}(Tp, Scale), \\
\quad \text{select-value}(Tp, \text{Profile}, \text{Rating}), \\
\quad \text{compare}(\text{Test}, \text{Scale}, \text{Fact}, \text{Rating}), \\
\quad \text{verify}(\text{Conditions}, \text{Profile}).
\]

The structure of the record in the clause can be declared as follows (shown in Pascal):

\[
\text{TYPE condition } \bowtie \text{ "node" ;} \\
\quad \text{node } \bowtie \text{ RECORD} \\
\quad \quad \text{Tp} : \text{factor} ; \\
\quad \quad \text{Test} : \text{relation} ; \\
\quad \quad \text{Rating} : \text{real} ; \\
\quad \quad \text{next} : \text{condition} ; \\
\quad \text{END ;}
\]

In the resulting canonical form, a pointer \( P_1 \) is introduced to replace the list structure in the head literal, and a second pointer \( P_3 \) is introduced to represent the partial list in the last literal. This results in the following canonical form:

\[
\text{verify } (P_1, \text{Profile}) \\
\quad := \text{scale}(Tp, Scale), \\
\quad \text{select-value}(Tp, \text{Profile}, \text{Fact}), \\
\quad \text{compare}(\text{Test}, \text{Scale}, \text{Fact}, \text{Rating}), \\
\quad \text{verify}(P_2, \text{Profile}), \\
\quad \quad Tp := P_1 \uparrow t_p, \\
\quad \quad \text{Test} := P_2 \uparrow \text{test}, \\
\quad \quad \text{Rating} := P_2 \uparrow \text{rating}, \\
\quad \quad P_2 := P_1 \uparrow \text{next}.
\]

In the case that the list is a null list as in the following clause

\[ \text{length}([], 0) \]

the null list is still represented by a pointer, but the pointer is set to 'nil' with an assignment added to the clause body. Thus we have the following canonical form

\[ \text{length}(P_1, V_i) : - P_1 := \text{nil}, V_1 := 0. \]

3 Sequencing Procedure Calls via Dependence Analysis

This section deals with the nondeterminism within each individual clause, that is, the issue of unknown order in the set of procedure calls which constitute the body of a clause.

3.1 Variable Dependence Relations

After a clause has been converted into the canonical form, the next step is to analyze each clause to determine the execution sequence of these operations. We have found that variable-dependence analysis is sufficient for determining the execution sequence of body literals as long as the input/output pattern of the parameters in each predicate, i.e. the mode of each predicate, is known and is unidirectional. Although variables in a predicate can be used either as input or output in pure logic programming, in general it is hard to write many bidirectional procedures in languages such as Prolog for various reasons given in [8].

As for imposing the requirement for mode information in order to prevent ambiguity in obtaining correct execution sequence, it should not affect expressiveness of logic programming for software development since all procedures (clauses) are usually written according to well-defined interfaces among the procedures. There should be little, if at all, uncertainty left about what input is required by a procedure or what output a procedure is supposed to produce during the development of a software.

It should be pointed out that the literals in a clause which is already in clause canonical form can be of three basic types. The first type is a procedure call. A procedure call can be easily recognized since it is of the form \( p(t_1, t_2, \ldots, t_n) \) where \( p \) is the predicate symbol and the \( t_i \)'s are its arguments. Note that \( p \) can also be system predicates such as read/write, etc. The second type is the assignment statements which are of the form \( X := \text{expression} \), where \( X \) is a variable name. As for the right-hand side of an assignment, it can be an arithmetic expression to be computed or simply a field of a record type introduced during conversion of a list structure into a record as described in the previous section. The third type consists of literals with comparison operators, including the equations introduced during the conversion of a clause into canonical form.
Before we discuss variable-dependence analysis, a dependence relation between variables is defined:

Definition 3 (Variable dependence relations)

Let \( P = \{ P_1, P_2, \ldots, P_k \} \) be a set of literals in a procedure, including the head literal and the body literals. The variable dependence relations \( \Rightarrow \) among literals are a set of ordered pairs drawn from \( P \times P \):  
\[
\Rightarrow = \{ (P_i, P_j) \mid \text{the value of variable } X \text{ in } P_i \text{ depends on the value of variable } X \text{ in } P_j \}
\]

If \( (P_i, P_j) \in \Rightarrow \), we write \( P_i \Rightarrow P_j \), and say that \( P_i \) is a direct predecessor of \( P_j \), and \( P_j \) is a direct successor of \( P_i \). If there is a sequence \( P_i \Rightarrow P_1 \Rightarrow \ldots \Rightarrow P_k \), \( P_k \) is a predecessor of \( P_i \). Two literals are independent if neither is a successor of the other.

According to the definition, a body literal which has a variable in input mode is a successor of another body literal in the same clause which has the same variable in output mode. However, a head literal which has a variable in input mode is a direct predecessor of its body literals which have the same variable in input mode. This is because the body literals can not be executed without the value of the input variable provided by the head literal. On the other hand, a head literal which has a variable in output mode is a direct successor of its body literals which have the same variable in output mode. This is because the value of the output variable to be returned to the caller depends on the value produced by its body literals containing that variable.

Two operations, \( OP_1 \) and \( OP_2 \), can be independent of each other because there does not exist any dependence relation between them. Under such circumstances, \( OP_1 \) can be executed before \( OP_2 \) or vice versa without affecting correctness of output produced by a sequence of operations where \( OP_1 \) and \( OP_2 \) occur. When such cases happen during the transformation, we will sequence \( OP_1 \) and \( OP_2 \) in the original order they are written.

The following example illustrates how variable dependence relations are obtained conceptually.

Example: Operation sequencing

The clause below is already in canonical form. The conversion was shown in the previous chapter.

\[
p(X, V_1) : = \ q(V_2, W), r(V_3, Z), V_1 := f(Z),
V_2 := X^2, V_3 := X + W.
\]

We will use \( e_1 \), \( e_2 \) and \( e_3 \) to stand for the three equations in the clauses. It is assumed that the mode of predicates \( p \), \( q \), and \( r \) are \( \{+, -\} \), \( \{1\} \), \( \{1\} \), and \( \{1, 1\} \) respectively. Since variable \( X \) in the clause head is in input mode, its value will be supplied by the caller when the procedure is called. The variable \( X \) is referenced in two assignment statements, \( e_2 : V_1 := X^2 \) and \( e_3 : V_3 := X + W \). Assignment \( e_2 \) then produces a new result into variable \( V_2 \) which in turn is required when procedure \( q(V_2, W) \) is to be invoked. The result returned from calling \( q(V_2, W) \) is stored in \( W \) which will be used in \( e_3 \) for computing \( V_3 \). Procedure \( r(\cdot) \) cannot start before the value of \( V_3 \) is available. After \( r(\cdot) \) is completed, it will return \( Z \) for computing the output \( V_1 \). As a result, the variable-dependence relations derived for this clause is shown graphically in Figure 1.

\[\begin{align*}
V_1 &= f(Z) \\
V_2 &= X^2 \\
V_3 &= X + W \\
q(V_2, W) \\
r(V_3, Z)
\end{align*}\]

Figure 1: An example of variable-dependence relations in a clause.
**Example: A clause with condition test literals**

The clause in the original form:

\[ f(X, X+1, Y) : - g(X, W), X \geq W, Y := \sqrt{X - W}. \]

The clause in clause canonical form:

\[ f(X, Y, V_Y, V_X) : - g(X, W), X \geq W, Y := \sqrt{X - W}, V_X := X + 1. \]

Assuming predicates \( f \) and \( g \) have the mode of \( f(\downarrow, \downarrow, \top) \) and \( g(\downarrow, \top) \) respectively, the variable dependence graph for this example is constructed as in Figure 3. Note that a super node which represents a condition test has an incoming edge for each of its nodes. This is because the variables involved in a condition test should all be in input mode.

A execution sequence obtained from the graph is shown below.

```plaintext
if \( V_1 = X + 1 \)
then begin
  call \( g(X, W) \);
  if \( g = \text{false} \) then return \( f = \text{false} \);
  if \( X \geq W \) then \( Y := \sqrt{X - W} \);
  return \( p = \text{true} \);
else return \( p = \text{false} \);
```

3.3 Anomalies in Variable Dependence Graphs

The variable dependence graph constructed according to the algorithm described in the previous section may contain a number of anomalous situations which prevent straightforward interpretation of execution sequence of operations in a clause. The possible situations are discussed below.

1. Existence of cycles in the graph
   - It is possible that there is inconsistency in the mode information of predicates provided, resulting in a situation in which two literals are dependent on each other. Consider the following example:

   \[ a(X, Y) : - b(X, W, Z), c(X, Z), d(W, Z, Y). \]

   If predicate \( b \) has the mode \( (\downarrow, \downarrow, \top) \) and predicate \( d \) has the mode \( (\top, \top, \top) \), then \( b() \) will have to be executed before \( d() \) because \( d() \) requires the value of \( Z \) from \( b() \). However, \( d() \) also has to be executed before \( b() \) because it supplies the value of \( W \) to \( b() \). Such cases lead to cycles in variable dependence graph.

   Cycles can also result from predicates which are used multi-directionally, i.e. there is more than one input/output pattern for the predicates. Different execution sequences may be obtained by following one direction or the other of a cycle, thus resulting in a set of different target procedures for a clause. In
logic programming systems, such ambiguity is allowed since it represents a nondeterminism which can be resolved at run time. However, in our case, all kinds of nondeterminism has to be removed so that a deterministic program can be generated. Therefore, it is assumed by our transformation method that all predicates are unidirectional, and we will consider it an error when cycles exist in a dependence graph.

2. Lack of dependence relations between supernodes - A graph may consist of a number of supernodes which do not have dependence relations among them. This implies the operations represented by the supernodes can be executed in any sequence. The exact sequence adopted by the target procedure would depend on the control flow features provided by the target procedural language. With more sophisticated languages, such as Ada or Simula, the operations may be arranged as communicating parallel tasks or as interleaved coroutines; whereas for languages such as Pascal or C, the operations in a clause have to be executed in some fixed sequence.

3. Lack of dependence relations between head and body of a clause - One of the reasons may be insufficient input. There may also be some variables declared as inputs in the body of a clause, while no dependence relation can be derived between these variables and those input variables in clause head. If the corresponding literal is a system predicate such as read(input, Ch) in the clause get(Ch) :- read(input, Ch), Ch # EOF. These variables are considered as an independent input which has the same effect as those in the clause head. Otherwise, if output variables in clause head do not depend on these variables, then these variables are redundant and can be removed. In other cases, the clause does not provide sufficient information for a correct computation and cannot be translated into a correct procedure. The other reason may be that the body literals fails to produce the results required by the clause head, which is most likely due to programming errors.

We conclude this section with the following two observations:

Observation 1: An execution sequence for the literals in a clause can be derived from the variable dependence graph constructed for the clause if the modes of all predicates in it are known and all predicates are used unidirectionally.

Observation 2: Any violation of the above rule will result in anomaly in the variable dependence graph, including cycles or missing links.

4 Procedure Formation

A target procedure is obtained by merging a set of OR-related clauses after the execution sequence of individual clauses in the set have been parsed. A procedure tree is formed for each set of OR-related clauses. The major steps for forming a procedure tree for each set of OR-related clauses is provided below.

Step 1: Identify the sets of OR-related clauses in the program.

Step 2: Unify the head literals of the clauses in each set and form the root of the tree accordingly.

Step 3: Create a condition node as a child of the root node for each clause in the set.

Step 4: Parallel scan the execution sequences of all clauses in the set, one literal for each clause at a time starting with the literal immediately following the head literal. (There will be one pointer for each clause in the set.)

Step 5: Create a node for each literal under the current pointer of each clause and insert it to its own branch. Fill in the literal and record the type of the literal on the node. Advance respective pointer for each clause.

Step 6: Repeat Step 5 until all literals of each clause in the set are processed.

The above process constructs a tree in which each node is associated with an operation to be performed. A branch in a procedure tree represents an alternative clause in the OR-related set. Each procedure tree can, therefore, be mapped directly to a conditional statement which becomes the basic structure of the target procedure.

The example in Figure 4 illustrates the formation of a procedure using the method described above. Clauses of the corresponding logic program are defined below:

1. b(Q,X) :- r(Q,Z), u(Z,X).
2. b(P,Y) :- t(P,W), v(W,Y).
3. t(Q,W) :- W := Q^2.
4. u(Y,Z) :- Y >= 90, Z := Y^1.1 + 5.
5. u(Y,Z) :- 50 < Y < 90, Z := Y^1.2 + 4.
6. v(Y,Z) :- 5 < Y <= 10, Z := 2 * Y + 3.
7. v(Y,Z) :- Y <= 5, Z := 10.
8. r(P,Y) :- P = 1, Y := 2.
10. r(P,Y) :- P = 3, Y := 12.
13. r(P,Y) :- P = 6, Y := 42.
14. r(P,Y) :- P = 7, Y := 56.
15. r(P,Y) :- P = 8, Y := 72.
16. r(P,Y) :- P = 9, Y := 90.
17. r(P,Y) :- P = 10, Y := 110.
Figure 4: An example of procedure formation

In the figure, each procedure is represented by a tree. Each node in a procedure tree is associated with an operation whose type is also identified. Each branch is headed by a condition that controls when the program execution has to take this route. A condition is usually represented by a subset of the domain of the control variable. Branching conditions are not specified in the original logic programs, as the program is nondeterministic. To make a procedure deterministic, variables that can be used for branching control have to be identified, and disjoint conditions for sibling branches have to be formulated. Once all branching conditions in a tree have been obtained, the tree can be translated into a deterministic procedure directly. For example, procedure b(P,Y) in the figure can be translated into the following code.

```plaintext
Procedure b(P, Y)
  real, P, Y, S, W
  begin
    if (condition 1) then begin
      call t(P, W);
      S := W+5;
      call V(S, Y);
      return b := "true";
    end
    else if (condition 2) then begin
      call r(P, X);
      call u(X, Y);
      return b := "true";
    end
    else error!
  end b.
```

The procedure formation method is straightforward. A more sophisticated one may merge common sequence appears in different subtrees. Merging alternative branches into single thread is a common practice in procedure programming. However, it does not appear to be crucial at the procedure formation level in general since most clauses are short. Further, in the target program, merging is implicitly done when different branches call the same procedure. Nevertheless, one potential problem that may arise from this is that the target program may be full of small procedures. This is one of the issues to be dealt with in our future research.

5 Facts for Deterministic Branching

In this section a method for determining branching conditions for each set of OR-related clauses is described.

Observation: Program execution should follow a particular branch only when certain input parameters are instantiated with certain values such that operations performed by that branch is capable of producing correct results.

The observation has two implications: one is that the branching condition of a branch can be deduced from the facts existing somewhere in the branch, and the other is that an alternative construct can be made deterministic when conditions associated with the branches are disjoint with respect to certain input variables which can be evaluated at the branching point. According to such an observation, each branching condition can be obtained by forming a set (called viable set) from the facts existing in the corresponding branch. The 'facts' mentioned above has a much broader meaning that the usual definition of facts in logic programming (which are simply unit clauses).

Definition 5 ('Facts' for Branching Control)

A fact which is usable as a branching condition is a clause, a literal, or a constraint which can be evaluated at compile time.

So, what a fact is really depends on how intelligent the translator is, i.e., whether the translator can evaluate it or not. Ground unit clauses, brother(John,Bill) for example, are facts. This kind of fact can be easily recognized by any dumb translator. However, equality such as $X^2 = 10$ can be considered as a fact by some translators but not by every one. More generally, an equation such as $f(X, Y) = g(U, W)$ can also be used as a fact when variables $X, Y, U,$ and $W$ are inputs. Virtually, any condition test statement can be used as a fact. In the following, we will describe, using some typical examples, on how facts can be used for deducing branching conditions.

Example: A Factorial Procedure

1. factorial(0,1), and
2. factorial(N,F) :- N > 0, N1 := N - 1, factorial(N1,F), F := N * F1.
Clause 1 can be converted into its canonical form as

\[ 1a. \text{factorial}(N,F) :- N=0, F:=1. \]

It is clear now that the two inequalities in clauses 1a and 2 can be used as branching conditions for the procedure.

Example: Clauses with Lists

The same simple principle can be applied to procedures which handles lists. For example the procedure length() defined by the two clauses below:

1. \( \text{length}([], 0) \).
2. \( \text{length}([\text{Tail}], N) :- \text{length}(\text{Tail}, N1), N:=N1+1. \)

The canonical form of these two clauses are:

1a. \( \text{length}(\text{Ptr}, N) :- \text{Ptr}=	ext{NIL}, N:=0. \)
2a. \( \text{length}(\text{Ptr}, N) :- \text{Ptr} \neq \text{NIL}, \text{Ptr}:=\text{next}, \text{length}(\text{Ptr}, N1), N:=N1+1. \)

It also clear from Clauses 1a and 2a that variable \( \text{Ptr} \) can serve as a control variable.

Facts identified in the previous examples, including simple inequalities and ground unit clauses, are some typical ones that can be recognized easily. In spite of its simplicity, we have found that they can be applied to determine branching conditions successfully for most logic programs. More sophisticated fact recognition capabilities (e.g., general condition tests) can be achieved with some enhancements to the proposed method. However, if the input logic program is incomplete in itself, then a procedure cannot be made deterministic no matter how intelligent the fact propagation scheme is. In this case, more information has to be provided by the user. For example, suppose the original version of a logic program includes the following segment:

Clause 1: \( p(X,Y) :- q(X,W), Y:=1/W. \)
Clause 2: \( p(X,Y) :- s(X,W), Y:=4+W. \)
Clause 3: \( q(X,W) :- X > 1, W:=X-1. \)
Clause 4: \( s(X,W) :- X < 5, W:=2-X. \)

In this segment, branching control is ambiguous when \( 1 < X < 5 \). Program execution will succeed, in this situation, in either one of the two branches corresponding to Clauses 1 and 2, but will deliver different results. The translator will report this kind of ambiguity to the user. The user then can refine Clauses 1 and 2 to, say

Clause 1': \( p(X,Y) :- q(X,W), X > W, Y:=1/W \)
Clause 2': \( p(X,Y) :- s(X,W), x \leq W, Y:=1/(-W) \)

or modified Clause 4 to

Clause 4': \( s(X,W) :- X \leq 1, W:=2-X. \)

In either way, the resulting program becomes deterministic: the branch corresponding to Clause 1 will be executed when \( X > 1 \), and the other will be executed when \( X \leq 1. \)

6 Concluding Remarks

In this paper we present a method as part of a program transformation system which is aimed for translating logic specifications into procedural programs. The method is applicable for a subclass of Horn clauses in which the mode information for each predicate is known. The method is composed of the following major steps. First the set of logic clauses input to the transformation system are converted to a form called \textit{clause canonical form} as a preparation for further analysis and translation. After the clauses are put into canonical form, each clause is analyzed to reveal variable-dependence relations that exist among the variables in the clause. Such relations are used to sequence procedure calls in each clause.

Each set of OR-related clauses are then organized into a tree in which each branch represents a clause in the set. In order that each procedure tree can be mapped to a target procedural module, the branching conditions are determined for each branch in the tree by extracting 'facts' from the procedure tree itself or other places in the forest via fact propagation. It is possible that during the process of sequencing procedure calls or determining the branching conditions that some inconsistency is discovered or that some crucial information should be provided by the program designer but was not. Under such circumstances, advice from the program designer is solicited. After the branching conditions for all trees are properly determined, each tree is readily mappable to a target procedure, and the final target program is the collection of all such procedures. This method can be potentially used for (1) automatic program synthesis where design specifications are either written in or translatable into Horn logic clauses, and (2) adaptation of existing logic programs to facilitate software reusability.

References