TRANSFORMATIONAL LOGIC PROGRAM SYNTHESIS

Taisuke SATO
Electrotechnical Laboratory,
1-1-4, Umezono, Sakura-mura,
Niihari-gun, Ibaraki, 305
JAPAN

Hisae TAMAKI
Ibaraki University, 4-12,
Nakanarusawa, Hitachi-shi,
Ibaraki, 316 JAPAN

ABSTRACT: A new approach to logic program synthesis from the first order specifications is presented. Our synthesis process starts with a specification for p(X) of the form p(X) :< : formula(X), where formula(X) is a first order formula and p(X) is an atomic formula. We assume that the predicate symbol "p" does not occur in formula(X). We also assume that every predicate in formula(X) is already defined by some logic program S (a set of definite clauses) and given an interpretation by the least model m(S) of S. Then, the interpretation of formula(X) by m(S) denotes a relation. Our objective is to demonstrate a method for synthesizing a logic program for "p" that computes the same relation as denoted by formula(X). The method relies on a "negation technique" [Sato 84] that eliminates negations and universal quantifiers from formula(X).

The negation technique derives a program S: that computes the relation q(X) from a program S for q(X) when S satisfies certain conditions. In order to satisfy these conditions which will be described later, an equivalence preserving program transformation [Tamaki 84] is often performed before the application of the negation technique.

Since negation technique plays a major role in our approach, we first describe it in section 3 and its application in section 4. Then we present a sample synthesis, i.e. the synthesis of an N-queens program in section 5.

1. INTRODUCTION

Although a lot of effort has been devoted to the problem of program synthesis, it remains a challenging problem. One of the reasons is the semantic gap between specification languages and programming languages. In this respect, it is advantageous to deal with the problem within the logic programming paradigm because logic programs are not only executable but also highly declarative.

In this paper we propose a new approach to logic program synthesis from the first order specifications. Our synthesis process starts with a specification of the form p(X) :< : formula(X), where formula(X) is a first order formula and p(X) is an atomic formula. We assume that the predicate symbol "p" does not occur in formula(X). We also assume that every predicate in formula(X) is already defined by some logic program S (a set of definite clauses) and given an interpretation by the least model m(S) of S. Then, the interpretation of formula(X) by m(S) denotes a relation. Our objective is to demonstrate a method for synthesizing a logic program for "p" that computes the same relation as denoted by formula(X). The method relies on a "negation technique" [Sato 84] that eliminates negations and universal quantifiers from formula(X).

The negation technique derives a program S: that computes the relation q(X) from a program S for q(X) when S satisfies certain conditions. In order to satisfy these conditions which will be described later, an equivalence preserving program transformation [Tamaki 84] is often performed before the application of the negation technique.

Since negation technique plays a major role in our approach, we first describe it in section 3 and its application in section 4. Then we present a sample synthesis, i.e. the synthesis of an N-queens program in section 5.

2. BACKGROUND

We assume that programs and formulas in this paper are written in a many sorted first order language L. We fix the language L and use U to denote the Herbrand universe, i.e. the set of all ground (variable free) terms in L. By convention, terms beginning with upper case letters are variables and those beginning with lower case letters are constants, function symbols or predicate symbols.

A definite clause is a formula of the form p0 :- pl & ... & pm (m>0) where pi (0<i<=m) is an atom (atomic formula). A logic program S is a finite set of definite clauses. The meaning of a program S is defined as the set of all ground atoms provable from S. We define:

\text{success}(S) = \{ p \mid p \text{ is a ground atom in } L \text{ and } S \vdash p \}.
Success(S) is called the success set of S. We use failure(S) to denote the finite-failure set of S as defined by Apt and van Emde Boas [Apt et al. 84]. Roughly speaking, it is the set of all ground atoms p such that SLR-refutation of C-p fails in a finite number of steps. The set failure(S) is a subset of all of the ground atoms unprevented from S. We define an interpretation m(S) over U as follows:

for any ground atom p in L
m(S) |= p  iff (if-and-only-if)
p is in success(S).

m(S) thus defined becomes the least model over U of S. That is, for any model I over U of S, if m(S) |= p then I |= p holds for every ground atom p. Since success(S) and m(S) are essentially the same, we use them interchangeably. In light of our program semantics, two programs S1 and S2 are equivalent iff m(S1) = m(S2).

3. NEGATION TECHNIQUE

The negation technique is a kind of program transformation. It is a procedure to derive a program S' from a given program S such that:

(1) Predicate names in S and those in S' have one-to-one correspondence. If we present the correspondence as, p in S $$\longleftrightarrow$$ p' in S', p and p' have the same arity k(x).  

(2) For any ground atom p(t1,...,tk) and p'(t1,...,tk),
not(S |= p(t1,...,tk)) if S' |= p'(t1,...,tk)

If S' satisfies (1) and (2), it is called a dual program of S. Moreover, when "if" in (2) can be replaced "iff", we call S' a complement program of S.

Successful computations by S' mimic failed computations by S. In this sense S' can be regarded as the procedural negation of S. We would like to show the negation technique by an example for saving space. Details are described in [Sato 84]. Given the program:

S = { mem(H,[H,[]]), mem(X,[H,[]]) $$\land$$ mem(X,[]) } ... (3-1)

where [a][b] stands for the term cons(a,b). The second argument of mem is a list and mem(a,b) is intended to mean that a is a member of list b. As stated before, S defines the binary relation 'mem' over U through m(S) so that the 'mem' relation is also defined. Since no clause in S has internal variables an internal variable is one occurring only in the body of a clause such as V in a(X)$$\leftarrow$$ b(X,Y)), we are able to obtain a program for 'mem' by applying the negation technique to S in the following way. First we apply steps 1-6 to each predicate in S.

[STEP 1] Construct an IPP-definition [Apt et al. 82] of the selected predicate. We obtain,

mem(A,B)<-
(exist H,L)(A,B)=$$\langle$$H,[H,[]]$$\rangle$$)
or
(exist X,H,L)(A,B)=$$\langle$$X,[H,L]$$\rangle$$& mem(X,L))
... (3-2)

where "=" means the syntactic identity in U and <a,b><c,d> is a shorthand for (a=b)&(c=d). We sometimes consider <a1,...,an> as a vector of terms for convenience.

[STEP 2] Negate both sides of the IPP-definition. (a,b)/=/<c,d> is a shorthand for $$\neg$$((a=b)&(c=d)).

~mem(A,B)<-
(all H,L)(A,B)$$
\langle$$H,[H,L]$$\rangle$$)
&
(all X,H,L)(A,B)$$
\langle$$X,[H,L]$$\rangle$$ or ~mem(X,L))
... (3-3)

[STEP 3] Transform every conjunct on the right side of the result of [STEP 2] which has the form,

(all X1,...,Xn)<A1,...,An>=/<c1,...,tk>
or
~pl or ... or ~pm

(m>0) to,

(all X1,...,Xn)<A1,...,An>=/<c1,...,tk>
or (exist X1,...,Xn)

$$\langle$$A1,...,An>=/<c1,...,tk>
& ~pl(X)
... or (exist X1,...,Xn)

$$\langle$$A1,...,An>=/<c1,...,tk>
& ~pm(X).

In this case we obtain,

~mem(A,B)<-
(all H,L)(A,B)$$
\langle$$H,[H,L]$$\rangle$$)
&
(all X,H,L)(A,B)$$
\langle$$X,[H,L]$$\rangle$$
or
(exist X,H,L)

$$\langle$$A,B)$$
\langle$$X,[H,L]$$\rangle$$&~mem(X,L))
... (3-4)

[STEP 4] Transform the right hand side to a disjunctive form.

~mem(A,B)<-
(all H,L)(A,B)$$
\langle$$H,[H,L]$$\rangle$$
&
(all X,H,L)(A,B)$$
\langle$$X,[H,L]$$\rangle$$
or
(exist X,H,L)

$$\langle$$A,B)$$
\langle$$X,[H,L]$$\rangle$$&~mem(X,L))
... (3-5)

[STEP 5] At this step we consider, for example, all H,L)(A,B)$$
\langle$$H,[H,L]$$\rangle$$ as defining a new unary predicate with a vector argument $$\langle$$A,B$$\rangle$$. In addition, we assume that there is a logic program which computes that predicate. This is possible and sound because for any ground term a and b, all H,L)(a,b)$$
\langle$$H,[H,L]$$\rangle$$ holds iff the two term vectors (a,b) and $$\langle$$H,[H,L]$$\rangle$$ are unifiable and unifiability is a recursive relation over U. (Practically speaking, this is computed by
"negation as failure rule" [Clark 78] such as not(a,b, -CH, [H[L]]) = true. For convenience, we introduce a parameterized predicate ununi(\langle A_1, ..., A_k \rangle, \langle T_1, ..., T_k \rangle) (k\geq 0) where \langle A_1, ..., A_k \rangle is an argument and a vector term \langle T_1, ..., T_k \rangle is a parameter. We stipulate that for any ground term \alpha \equiv \langle \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \..
specification.

\(-\text{arl}(L,N) \iff (\exists X)(\text{mem}(X,L) \land (\text{num}(X) \leq N))\)  \(\text{(4-2)}\)

Then we consider "\text{arl}" as a new predicate symbol and (4-2) as a specification for \(-\text{arl}(L,N)\). (4-2) is apparently satisfied by a logic program:

\(-\text{arl}(L,N) \iff (\exists X)(\text{mem}(X,L) \land (\text{num}(X) \leq N))\)
  \text{clauses for "mem", "\text{num}".}  \(\text{(4-3)}\)

If we can apply the negation technique to (4-3), we will obtain a correct program for the specification (4-2). However, the existence of the internal variable X in (4-3) is an obstacle to the application (see Step 1). We use the logic program transformation system [Tamaki 84] to eliminate internal variables. This system has two basic transformations: One is "unfolding" or "unfolding and one unfolding"; the other is "folding" or "folding goals." Each of these procedures yields an equivalent program that includes:

\(-\text{arl}(H[0],N) \iff (\exists X)(\text{num}(X) \leq N))\)
  \text{clauses for "\text{num}".}  \(\text{(4-4)}\)

This program is guaranteed to be equivalent to (4-3) with respect to "\text{arl}". Moreover it has no internal variables so that we can apply negation techniques. We get:

\(-\text{arl}(L,N) \iff (\exists X)(\text{num}(X) \leq N))\)
  \text{clauses for "\text{num}".}  \(\text{(4-5)}\)

Suppose that a specification \(p(X_1, \ldots, X_n) \iff \text{formula}(X_1, \ldots, X_n)\) is given in conjunction with a program \(S_0\) which defines the primitive predicates used in \(\text{formula}(X_1, \ldots, X_n)\). And suppose that we have successfully synthesized a program \(S_1\) for \(p(X_1, \ldots, X_n)\). We say that \(S_1\) is partially correct with respect to the specification if for any ground term \(t(i = c, \ldots, a_n)\) implies \(m(S_1) = p(a_1, \ldots, a_n)\) iff \(p(a_1, \ldots, a_n)\) holds. When the equivalence, \(m(S) = \text{formula}(a_1, \ldots, a_n)\) holds, we say that \(S_1\) is totally correct with respect to the specification or \(S_1\) realizes the specification \(p(X_1, \ldots, X_n) \iff \text{formula}(X_1, \ldots, X_n)\).

We show that (4-5) is totally correct with respect to (4-4). First (4-5) is a complementary program of (4-4) because (4-4) is dichotomous. Second (4-4) is equivalent to (4-3) with respect to "\text{arl}" and (4-3) is totally correct with respect to (4-2). Therefore, the complement of the "\text{arl}" relation computed by (4-5) satisfies (4-2).

In other words, the relation computed by (4-5) satisfies (4-1) and, in addition, since (4-5) is obtained regardless of the content of the 2-ary predicate \(1 = \text{num}(X)\), (4-5) remains totally correct even when \(1 = \text{num}(X)\) is replaced by some other predicate. This fact will be used in Section 5.

As this example shows, the problem of synthesizing a program for a universally quantified specification like \(p(X) \iff (\forall Y)\text{formula}(X,Y)\) can be solved in three steps. The first step is the logical negation of a given specification and its realization by a program \(S_1\). The second is the transformation of \(S_1\) to an equivalent program \(S_2\) to which negation technique is applicable. The third is the application of the negation technique to \(S_2\). This method is called the double negation technique. If the transformation from \(S_1\) to \(S_2\) is successful, the result of the double negation technique is partially correct with respect to the given specification. In addition, if \(S_2\) is dichotomous, the result becomes totally correct.

5. SYNTHESIS OF AN N-QUEENS PROGRAM

In this section we demonstrate a synthesis of an N-queens program which searches for a "mutually non-attacking arrangement of N queens" on an N by N chess board. We assume that the primitive relations defined by the following self-explanatory program are available.

[ Given ]

\{ \text{len}(\text{[]},0), \text{len}(\text{[H]},1) < \text{len}(\text{[L]}, N), \}
  \text{mem}(H,\text{[H][1]}, \text{mem}(X,\text{[H][1]}, \text{mem}(X,\text{[H][1]}, \text{mem}(X,\text{[H][1]}, \text{ap}(I,X,X), \text{ap}(I,X,Y), \text{[H][2]}, \text{ap}(X,Y,Z),
  \text{add}(O, X, Y), \text{add}(X, Y, X), \text{add}(X, Y, Z), \text{check}(N, A, B) \iff \text{add}(A, B, Z) \iff \text{check}(N, A, B), \text{check}(N, A, B) \iff \text{add}(A, B, Z) \iff \text{check}(N, A, B)\}
\}

Here \(X_{i+1}\) stands for \(s(X_i)\) and "\text{mem}" is a successor function. Using these predicates we specify the N-queens problem in a top-down manner as follows.

[ Specifications ]

\(\text{queen}(L, N) \iff \text{arl}(L, N) \land \text{safe}(L)\)  \(\text{(5-1)}\)
\(\text{arl}(L, N) \iff\)
\(\text{len}(L, N) \land (\forall X) (\text{mem}(X, L) \longrightarrow \text{within}(X, N))\)  \(\text{(5-2)}\)
\(\text{within}(X, N) \iff\)
\(\text{exist}(Y, Z) (\text{add}(Y, X, Z) \land \text{add}(Z, X, N))\)  \(\text{(5-3)}\)
\(\text{safe}(L) \iff\)
\(\text{all}(A, B, N) (\text{diff}(A, B, N) \longrightarrow \text{check}(N, A, B))\)  \(\text{(5-4)}\)
\(\text{diff}(A, B, L, N) \iff\)
\(\text{exist}(X, Y, Z) (\text{ap}(X, A, Y) \land \text{ap}(Z, B, W) \land \text{len}(A, V, N))\)  \(\text{(5-5)}\)
queen(L,N) means that list L is an answer to the N-queens problem and it is defined by ar(L,N) and safe(L). ar(L,N) means that list L has length N and each integer in L satisfies within(X,N). within(X,N) means that 1<=X<=N. safe(L) means that any pair of queens in list L are mutually non-attacking. It is defined using dif(A,B,L,N) which means that the distance between A and B in list L is N. Every argument is supposed to have an appropriate sort.

Our synthesis process proceeds in a bottom up manner, i.e., in the order (5-3), (5-2), (5-5), (5-4) and finally (5-1). We first start with (5-3).

within(X,N)<- (exist Y,Z)(add(L,Y,X)&add(Z,X,N)) ... (5-3)

Program synthesis of this type which has the form p(X) <- (exist Y) q(X,Y) is straightforward. A program \{ p(X) <- \neg q(X,Y), clauses for "\neg q" \} realizes the specification so that (5-3) is realized by:

\{ within(X,N)<-add(L,Y,X)&add(Z,X,N), clauses for "add" \}

This is optimized by an equivalence preserving transformation system [Tamaki 64]. After one unfolding and one folding operation, we obtain:

\{ within(N,N)<-add(L,Y,N),
within(X,N+1)<-within(X,N),
classes for "add" \} ... (5-3')

Next we move to the synthesis of (5-2).

ar(L,N)<- len(L,N)(\forall X)(mem(X,L) -> within(X,N)) ... (5-2)

This requires the synthesis of ar(L,N) <- (all X)(mem(X,L) -> within(X,N)) which has been done already in section 4 using the double negation technique. The result is (4-5) where we identify \( 1<=X<=N \) with within(X,N). Therefore, a program,

\{ ar(L,N)<-len(L,N)&ar(L,N),
classes for "len" \} U (4-5) U (5-3')

constitutes a totally correct program with respect to the specification (5-2). An efficient program, however, is more desirable. So we start the equivalence preserving transformation process by setting.

ar(L,N,X)<- (exist N)(len(L,N)&add(M,X,N)&ar(L,N)).

It is obvious that ar2(L,N,C) <- len(L,N) & ar(L,N) holds. After 3 unfolding operations and 1 folding operation, we reach a totally correct program with respect to the specification (5-2).

\{ ar(L,N)<-ar2(L,N,O),
ar2([],N,N),
ar2([H|L],N,2)<-within(H,N)&ar2(L,N,X+1)
within(N,N)<-add(Y,N),
within(X,N+1)<-within(X,N),
classes for "add" \} ... (5-2')

After having realized (5-2) as a program, we move to the synthesis of (5-4). First we realize the specification (5-5) by the program:

\{ dif(A,B,L,N)<-
ap(X,[A|Y],2)&nap(Z,[B|W],L)&\neg len([A|Y],N),
classes for "\neg" and "len" \}

Second we transform this program to an equivalent one that has no internal variables. This requires 7 unfolding operations, 3 folding operations and 1 introduction of a new predicate dif(L,B,N) whose definition is:

dif(L,B,N)<- (exist Y,W,\neg ap(Y,[B|W],L)&\neg len(Y,N).

dif(A,B,[H|L],N)<-dif(A,B,L,N)
dif(A,B,[A|L],N)<-dif(L,B,W),
dif(L,B,[B|L],O),
dif(L,[H|L],N)<-dif(L,B,N) \} ... (5-5')

We next apply the double negation technique to (5-4). First we consider the specification,

\{ exist A,B,N (dif(A,B,L,N)&\neg check(N,A,B)) \} ... (5-6')

and realizes it by a program that has no internal variables. This requires 2 unfolding operations, 3 folding operations and 1 introduction of a new predicate \neg safe(L,A,L,X) whose definition is \neg safe(L,A,L,X) <- (exist B,N) & dif(L,B,N) & \neg within(N,X,N) & \neg check(M,A,B) ( safe(L,A,L,X) <- (all B,N) (dif(L,B,N) & \neg within(N,X,N) -> check(M,A,B) ).

Here we consider \neg safe, \neg safel as new predicate symbols. The resulting program is:

\{ \neg safe([H|L]<-\neg safe(L),
\neg safe([H|L]<-\neg safel([H|L],1),
\neg safel([A|B|L],K)<-check([A|B],K),
\neg safel([A|H|L])<-\neg safel([A|L],K+1) \} ... (5-4')

Note that this program has no internal variables. In addition, it is dichotomous. By applying the negation technique to this program, we finally obtain:

\{ safe([L]),
safe([H|L]<-safel([H|L],1)&safe(L),
safe([A|B|L],K)<-check(M,A,B),
safe([A|H|L])<-safel([A|L],K+1) \} ... (5-6'')

which is totally correct with respect to the initial specification (5-4). Thus we have
realized the specifications (5-2), (5-4) by (5-2') (5-4'') respectively. Now, the
top-most specification (5-1) is realized by

\[ \text{queen}(L,N) \land \neg \text{ar}(L,N) \land \text{safe}(L) \]

\[ \lor (5-2') \lor (5-4'') \ldots \ldots \ldots (5-1') \]

Although (5-1') is totally correct with
respect to (5-1), it is too brute a generate-and-test program. We attempt an
improvement by transformation. The transformation begins with a new predicate

\[ \text{queen}(L,N,X) \land \neg \text{ar}(L,N,X) \land \text{safe}(L) \]

After 3 unfolding operations, 3 folding
operations and 1 introduction of a new
predicate \[ \text{queen}(H,N,L,Y) \leftrightarrow \text{within}(H,N) \land \text{safe}(N,L,Y) \], we reach the following program

[ Synthesized Program ]

\[ \text{queen}(L,N) \rightarrow \text{queen}(L,N,0), \]
\[ \text{queen}(L,N,0), \]
\[ \text{queen}(Q,L,N,X) \rightarrow \text{queen}(L,N,X+1) \land \text{queen}(Q,L,N,X), \]
\[ \text{queen}(Q,N[L],X) \rightarrow \text{within}(Q,N), \]
\[ \text{queen}(Q,N[R],[L],X) \rightarrow \text{queen}(Q,N,L,X+1) \land \text{check}(X,Q,D), \]
\[ \text{within}(N,W) \leftrightarrow \text{add}(1,2,Y,N), \]
\[ \text{within}(X,W) \leftrightarrow \text{within}(X,N), \]

clauses for "add", "check" ... | ... (5-1'')

When \text{queen}(L,N) is called with specified
N, the program chooses queen (an integer Q,
l=Q< N) and place them on the chess board
by one. Whenever a queen is placed, it checks
whether or not it is mutually non-attacking to
the existing queens on the board.

To derive this program, 17 unfolding
operations, 12 folding operations, 11
introductions of new predicates and 2
applications of the negation technique were
required. We do not evaluate whether or not
the derived program is worthy of those costs.
However, we emphasize that it is guaranteed to
be not only partially correct but also totally
correct with respect to (5-1). For during the
synthesis process, every intermediate program
to which the negation technique was applied
was dichotomous, and the transformation system
preserves program equivalence.

6. DISCUSSION

We have illustrated a transformational
approach to logic program synthesis based on
the negation technique. It is summarized as
follows. Suppose that \textbf{a}(X), \textbf{b}(X), \textbf{c}(X,Y) are
predicates defined by some logic program \textbf{S}
through its least model. Then, a
specification for a predicate \textbf{p}(X) in the left
side is realized by a program in the right
side.

\[ p(X) \rightarrow a(X) \land b(X) \]
\[ \rightarrow p(X) \rightarrow c(X,Y) \]
\[ \rightarrow p(X) \rightarrow a(X) \]
\[ \rightarrow p(X) \rightarrow (\forall Y) c(X,Y) \]

When a specification \textbf{p}(X) \rightarrow \textbf{formula}(X)
is given, a partially or totally correct program
with respect to the specification can be
synthesized by recursive application of this
table to the subformulas of \textbf{formula}(X) with
the help of the equivalence preserving
transformation as seen in section 5. Our
method has the following interesting features.

First our system does not suffer from
determinacies caused by the deduction in a
formal system as compared with the deductive
approach to logic program synthesis [Clark et
al. 77], [Eriksson et al. 82], [Hansson et al.
79], [Hogger 81]. Instead, we have to cope
with determinacies in the transformation
process in the double negation technique or
those in optimization. But the skeletal
process is deterministic and has no need for a
search process such as 'guess step' in [Shibut
80].

Second the system synthesizes not only a
function but also a (non-deterministic)
program for a relation and it does not require
any existence proof of the object to be
synthesized unlike [Manna 81], [Sato 79].
Induction plays only a secondary role in our
approach though it may be used to establish,
for example, \( (X+Y)+Z = X+(Y+Z) \).

Third it has simple and clear semantic
basis which elucidates the meaning of the
synthesis process. For example, every
predicate introduced during the synthesis has
a first order specification. Such
specification can be helpful in the
optimization stage.

On the other hand since our method is a
one-to-one mapping from a subformula in the
given specification to a program, we may lose
opportunities to shorten the path to the final
program by 'macro processing' of the
specification. Moreover the output program
tends to have a flavor of generate-and-test so
that the subsequent optimization becomes very
important as is exemplified in section 5.

We hope that the synthesis method
presented here will contribute one step toward
(a semi-) automatic programming environment
which logic programming aspires to achieve.
ACKNOWLEDGEMENT: The authors are grateful to Monica Strauss for detailed corrections. Thanks are due also to the members of Machine Inference Section of Electrotechnical Laboratory and the working groups of the Fifth Generation Computer Project.

REFERENCES:


