数理解析研究所講究録
第625巻 1987年 87-97

Generation of Rewriting Programs from Horn Clause Specifications

富樫 敦　グレン マンスフィールド　野口 正一
Atsushi TOGASHI　Glenn MANSFIELD　Shoichi NOGUCHI
Research Institute of Electrical Communication, Tohoku University
2-1-1 Katahira-cho, Sendai-shi 980 Japan

1. INTRODUCTION

Generation of machine executable programs from specifications written in higher level languages has always been an attractive idea. Several attempts have been made in this direction. The basic approach in program generation is the theorem proving approach. In the theorem proving approach the specification is in the form of a sequent[4], a tableau[10] or a set of non-clausal logic sentences[11]. The program is generated as a by product of the proof of a theorem specifying the program and it generally involves generation of new sequents, tableaus or sentences by application of deduction rules. A singular characteristic of these systems is that they are basically heuristic and rely heavily on human intuition to chose from the exploding number of legal next steps. This has made it extremely difficult to design an algorithm to mechanically generate programs from specifications.

In this paper we address the issue of generating rewriting programs, a kind of functional programs, from specifications given in the form of Horn clauses based on program transformation techniques. Horn clauses are directly executable as logic programs (e.g. Prolog [9]). Our motivation behind this transformation lies in the fact that, in general, functional programs are more efficient than logic programs. The reason being that, unlike logic programs which rely on backtracking, functional programs may be transformed into confluent forms where backtracking is not necessary [7,8]. The key rules used in the generation of rewriting programs from Horn clause specifications are "folding" and "unfolding".

To illustrate our idea of program generation, or program transformation,we take the example of a list reversal program. The "list reversal" program in Prolog is

$$Rev([], []). \quad (1)$$
$$Rev([A \mid X], Z) \text{ :- } Rev(X, Y),\ Ap(Y, [A], Z). \quad (2)$$
$$Ap([], X, X). \quad (3)$$
$$Ap([A \mid X], Y, [A \mid Z]) \text{ :- } Ap(X, Y, Z). \quad (4)$$

where *input-output assignment* of the variables in the *Rev* predicate is given in the form:

$Rev(X{:}\text{in}, Z{:}\text{out})$

This means *Rev* takes a ground term as its first parameter and a variable as the second parameter in every goal. The definition of *rev* in the target (functional) language is given by the clause of the form:

$$rev(X) = Z \leftarrow Rev(X, Z). \quad (5)$$

Step -1: by matching the rhs of (5) with the lhs's of (1) & (2) we obtain

$$rev([]) = []. \quad (6)$$
$$rev([A \mid X]) = Z \leftarrow Rev(X, Y),\ Ap(Y, [A], Z). \quad (7)$$

respectively.

Step -2: by matching the term $Rev(X,Y)$ in (7) with the rhs of (5) we get

$$rev([A \mid X]) = Z \leftarrow rev(X) = Y,\ Ap(Y, [A], Z). \quad (8)$$

Step -3: applying the resultant equality in the body of (7-1) to the entire clause we have -

$$rev([A \mid X]) = Z \leftarrow Ap(rev(X), [A], Z). \quad (9)$$

The definition of the append predicate ap is

$$ap(X,Y) = Z \leftarrow Ap(X,Y,Z). \quad (10)$$

Step -4: as in step -2 matching the rhs's (9) & (10) we obtain

$$rev([A \mid X]) = Z \leftarrow ap(rev(X), [A]) = Z. \quad (11)$$

Step -5: as in step -3 applying the resultant equality to the entire clause we obtain

$$rev([A \mid X]) = ap(rev(X), [A]). \quad (12)$$

In a similar manner using the specifications of append we obtain -

$$ap([], X) = X. \quad (13)$$
$$ap([A \mid X], Y) = [A \mid ap(X,Y)]. \quad (14)$$

As a result of the above exercise we have obtained clauses (6), (12), (13)and (14) which are in the functional form. Thus we have, in effect, generated a rewriting program-

$$rev([]) = []. \quad R1.$$
$$rev([A \mid X]) = ap(rev(X), [A]). \quad R2.$$
$$ap([], X) = X. \quad R3.$$
$$ap([A \mid X], Y) = [A \mid ap(X,Y)]. \quad R4.$$

from the Horn clause specifications of the list reversal program given in (1) - (4).

2. REWRITING PROGRAMS

We assume familiarity with the basic notions of *(many-sorted) equational logic* and *term rewriting systems*. See for instance [8]. For simplicity of notation, we assume we have only one sort; all the results of this paper carry over to many-sorted cases without difficulty.

In this paper, we use several symbols as syntactical meta variables. We use X, Y, Z for *variables*, f, g, h for *function symbols*, a, b, c for *constants*, M, N, L, R, K, l, r for *terms*, s, t for *ground terms* (terms containing no variables), u, v for *occurrences*, and $\theta, \sigma, \eta, \zeta$ for *substitutions*, possibly with primes or subscripts. The symbol $\equiv$ is used to denote the *syntactical identity*. For a term M, we denote by $Ocr(M)$ its set of

occurrences and by M/u the *subterm* of M at the occurrence $u \in Ocr(M)$. We use $Var(M)$ to denote the set of variables occurring in M. Given two terms M, N and an occurrence $u \in Ocr(M)$, we define $M[u \leftarrow N]$ as the term M in which the subterm M/u at the occurrence u is replaced by N, and $M[N]$ as the term M in which some subterm of M is replaced by N.

Definition A *term rewriting system* is a finite set RS of *rewriting rules* of the form $l \rightarrow r$ such that $Var(l) \supset Var(r)$, where l and r are terms.

RS may be *applicable* to a term M if and only if there is an occurrence $u \in Ocr(M)$ such that $M/u = l\theta$, for some rule $l \rightarrow r \in RS$ and for some substitution θ. In this case, we say that the rule $l \rightarrow r$ is *applied* to the term M to obtain the term $M[u \leftarrow r\theta]$. The choice of which rule to apply is made in a non-deterministic way. We write $M \Rightarrow_{RS} N$ to indicate that a term N is obtained from a term M by a single application of some rule in RS. Let ${}^*\Rightarrow_{RS}$ denote the reflexive and transitive closure of $\Rightarrow_{RS}$. If $M {}^*\Rightarrow_{RS} N$ holds, we say M is *reducible* to N in RS. RS may be omitted from $\Rightarrow_{RS}$ and ${}^*\Rightarrow_{RS}$ when it is clear from the context.

We shall formulate a rewriting program- a program which can be regarded as a set of rewriting rules, in the framework of a term rewriting system as in [5] with emphasis on the irreversibility of the rules. The theoretical issues related to computing with rewriting rules have been treated in detail in [7,8,12]. Hoffmann and O'Donnell [5] have illustrated the usefulness of this style of programs, and have also investigated the problems involved in implementing its programs.

Let Σ be a (finite) *signature* of function symbols. Following [15], we assume the signature Σ is partitioned as $\Sigma = \Sigma^c \cup \Sigma^d$. We shall call the function symbols in Σ^c *constructors*, and the elements in Σ^d *defined function symbols*. Constructors create concrete data structures to be processed. Defined function symbols define certain manipulations over the constructed data structures; their meanings are described using rewriting rules.

Definition An *rewriting program* on Σ is a term rewriting system RS such that each rewriting rule is of the form $f(M_1, \ldots, M_n) \rightarrow M$, where f is a defined function symbol.

Let RS be a rewriting program. A *computation* (*sequence*) from a ground term M_0 is a, possibly infinite, reduction sequence $M_0 \Rightarrow M_1 \Rightarrow \ldots \Rightarrow M_n \Rightarrow \ldots$. The computation *succeeds*, or *successfully terminates* if M_n is a ground term t containing no defined function symbols for some $n \geqq 0$; hence, by the definition of rewriting programs, no further rule can be applied to M_n. In this case, $M_n = t$ is the *result* of this successful computation. Otherwise, the computation fails, i.e., either it terminates at a term M which includes some defined function symbols, or it never terminates.

Example 1. Using the list reversal program *R1-R4* derived in section 1, a computation for the term $rev([a|[b|[c|nil]]])$ successfully terminates, and results in $[c|[b|[a|nil]]]$.

3. PROGRAM GENERATION

3.1 Equational Clauses

Definition An *equational clause* is a formula in first order logic (with equality =) of the form

$L_1 = R_1, \ldots, L_m = R_m \leftarrow M_1 = N_1, \ldots, M_n = N_n$,

where each L_i, R_i, M_j, N_j $(1 \leqq i \leqq m,\ 1 \leqq j \leqq n)$ is a term.

Definition An (*oriented*) *equational definite clause* (or a *conditional equation*) is an equational clause of the form

$L = R \leftarrow M_1 = N_1, \ldots, M_n = N_n$.

In this paper, an equational definite clause is implicitly oriented from left to right: the equation $L = R$ in the head (conclusion) part of a clause is treated as the rewriting rule $L \rightarrow R$ rather than the equation $L = R$. Note that an equation $L = R$ (or more explicitly a rewriting rule $L \rightarrow R$) is an equational definite clause without conditions. In equational definite logic, a predicate can be viewed as a truth function; an atomic formula A (in first order logic) is expressed as the equation $A = true$. Hence, a definite clause A :- $B_1, \ldots, B_n$ in Horn clause logic is represented as the equational definite clause $A = true \leftarrow B_1 = true, \ldots, B_n = true$. Throughout this paper, we will often refer to an equational definite clause simply as an equational clause for brevity.

3.2 A Deductive System

Any set S of equational clauses defines a reduction relation on terms. To define the reduction relation associated with S, let us consider a deductive system RD consisting of the following inference rules:

(*reflection*)

$$\frac{}{M \geqq M}$$

(*replacement*)

$$\frac{M \geqq N}{L[M] \geqq L[N]}$$

(*transition*)

$$\frac{M \geqq L \qquad L \geqq N}{M \geqq N}$$

(*substitution*)

$$\frac{M \geqq N}{M\theta \geqq N\theta}$$

(*modus ponens*)

$$\frac{M_1\theta \geqq K_1 \quad N_1\theta \geqq K_1 \quad \ldots \quad M_n\theta \geqq K_n \quad N_n\theta \geqq K_n}{}$$

$$M\theta \geqq N\theta$$

(where $M = N \leftarrow M_1 = N_1, \ldots, M_n = N_n \in S$)

We say $M \geqq N$ is *provable* from a set S of equational definite clauses, denoted by $S \vdash M \geqq N$, if there is a *proof (figure)* of $M \geqq N$ from S in RD. Note that the notation $M \geqq N$ for ordered pairs stems from the fact that the ordered pairs provable from a set of equational clauses in the deductive system are characterized by means of a partial ordering relation on terms when we give an interpretation to each function symbol.

Now, we shall define a reduction relation induced by a set of equational definite clauses.

Definition Let S be a set of equational definite clauses. For terms M and N,

(1) M is *reducible* to N in S, $M \ {}^*\!\!\rightarrow_S N$, if and only if $S \vdash M \geqq N$;

(2) M and N are *converging* in S, $M \downarrow_S N$, if and only if $M \ {}^*\!\!\rightarrow_S L$ and $N \ {}^*\!\!\rightarrow_S L$ for some term L;

(3) M is *reducible* to N *in one step* in S, $M \rightarrow_S N$, if and only if there is an equational clause $L = R \leftarrow M_1 = N_1, \ldots, M_n = N_n \in S$ such that $M/u \equiv L\theta$, $N \equiv M[u \leftarrow R\theta]$ and $M_i\theta \downarrow_S N_i\theta$, $1 \leqq i \leqq n$, for some $u \in Ocr(M)$ and for some substitution θ.

Proposition 1. *The reduction relation* ${}^*\!\!\rightarrow_S$ *is the reflexive and transitive closure of* $\rightarrow_S$: $M \ {}^*\!\!\rightarrow_S N$ *iff* $M \equiv M_0 \rightarrow_S M_1 \rightarrow_S \ldots \rightarrow_S M_n \equiv N$ *for some* $n \geqq 0$.

Proof. *If part* is straightforward by the definition of $\rightarrow_S$. *Only if part* can be verified by structural induction on the proof of $M \ {}^*\!\!\rightarrow_S N$ (i.e., on the proof figure which proves $S \vdash M \geqq N$ by definition).

Proposition 2. *Let HS be a set of definite clauses and S the set of equational definite clauses translated from HS. For any ground atom A,* $A \in Model(HS)$ *iff* $S \vdash A \geqq true$ *(equivalently* $A \ {}^*\!\!\rightarrow_S true$*), where Model(HS) denotes the least Herbrand model of HS.* (Refer to [9] for the precise definition of the least Herbrand model of definite clauses.)

Proposition 3. *Let RS be a rewriting program. For any terms M and N,* $M \ {}^*\!\!\Rightarrow_{RS} N$ *iff* $RS \vdash M \geqq N$ $(M \ {}^*\!\!\rightarrow_{RS} N)$.

3.3 Generation Rules

Our aim is to generate rewriting programs from Horn clause specifications. Our approach is based on program transformation. It adopts the "unfolding" and "folding" techniques in [14] as key generation rules. The basic structure of this approach is the triple $<S; D; P>$, where

1. S is a Horn clause specification, i.e., a set of oriented equational definite clauses obtained from a given set HS of Horn clauses by converting each definite clause A :- $B_1, \ldots, B_n \in HS$ into the form $A = true \leftarrow B_1 = true, \ldots, B_n = true$.

2. D is a set of definitions, each of them is of the form

$$f(X_1, \ldots, X_k) = Z \leftarrow M_1 = N_1, \ldots, M_m = N_m,$$

where f is a new function symbol not appearing in HS. Definitions explicitly define functions. The head part $f(X_1, \ldots, X_k) = Z$ specifies input-output assignment to the variables $X_1, \ldots, X_k$ and Z. The body part $M_1 = N_1, \ldots, M_m = N_m$ specifies the conditions which should be satisfied for the head part to be true.

3. P is a set of oriented equational definite clauses, to which generation rules are applied until every clause in P is altered into an equation.

How to Generate

An essential distinction between logic and functional languages is input-output directionality as discussed in [13]. Functional languages are directional in that the programs make an explicit commitment about which quantities are inputs and which are outputs. Logic programs do not make such a commitment. However, a logic program usually has a predicate which possesses, from its own purpose, an implicit commitment about input-output assignment to its parameters. For example, in the list reversal program in section 1, for the purpose of computing the reverse list of a given list, a goal of the form $rev(L, Z)$ is imposed, where L is a list and Z is a variable. This goal provides input for the first parameter and reflects an expectation that the second parameter is an output. We can thus say the list reversal program has the main predicate *rev* with mode (in, out). Here, a *mode* is an assignment of input and output to the parameters of a predicate symbol [13].

Let HS, a set of definite clauses, be the specification of a rewriting program. We assume that there is a main predicate p with some mode **m** assigning one output and the rest inputs to the arguments of p. Without loss of generality, we assume that the last parameter of p is output in **m**, i.e., $p(X_1, \ldots, X_k$: in; Z: out).

The generation process proceeds as follows.

Procedure: *Rewriting Programs*
Input: [*HS*: *Horn Clause Specifications*]
Output [*P* : *Rewriting Programs*]

Convert HS into the set S of equational definite clauses and set.

$$D := \{f_p(X_1, \ldots, X_k) = Z \leftarrow P(X_1, \ldots, X_k, Z) = true\};$$
$$P := \{f_p(X_1, \ldots, X_k) = Z \leftarrow P(X_1, \ldots, X_k, Z) = true\}$$

Apply the following rules non-deterministically until every clause in the current set P becomes a rewriting rule. The resulting P is the desired rewriting program.

- Definition Rule
- Unfolding Rules
- Folding Rule
- Substitution Rules
- Elimination Rule
- Splitting Rule
- Deletion Rule

The rules used in the procedure translate the current sets $S; D; P$ into new ones $S; D'; P'$, this is depicted as:

$$\frac{S; D; P}{S; D'; P'}$$

Now, we shall describe each rule in the sequel. We use letters Γ, Δ, and Λ, possibly with primes or subscripts, to denote the sequences of equations. In the following, two groups of function symbols are provided: constructors and defined function symbols. Constructors are ones appearing in the original specification *HS* as function symbols and defined function symbols are ones introduced by the *Definition Rule.*

The Definition Rule

The definition rule introduces new definitions of functions expressed as:

$$\frac{S;\ D;\ P}{S;\ D \cup \{f(X_1, \ldots, X_k) = Z \leftarrow \Gamma\};\ P \cup \{f(X_1, \ldots, X_k) = Z \leftarrow \Gamma\}}$$

where f is a new function symbol not appearing in $S \cup D \cup P$, which is treated as a defined function symbol, and Γ is a sequence of equations constructed from function symbols in $S \cup D$, variables $X_1, \ldots, X_k, Z$ and other variables.

Application of any of the following rules modifies P only. While the sets S and D remain intact. So the description of S and D are omitted in these rules.

The Unfolding Rules

The unfolding rules unfold clauses in P with equational clauses from $P \cup S$. The *lhs Unfolding Rule* replaces the left hand sides of the equations in the body parts of clauses expressed as:

$$\frac{P \cup \{M = N \leftarrow \Gamma, L = R, \Delta\}}{P \cup \{(M = N \leftarrow \Gamma, L[u_i \leftarrow N_i] = R, \Lambda_i, \Delta)\theta_i \mid 1 \leqq i \leqq k\}}$$

such that $(L/u_i)\theta_i \equiv M_i\theta_i$, for each $M_i = N_i \leftarrow \Lambda_i \in S \cup P$. On the other hand, the *rhs Unfolding Rule* replaces the right hand sides:

$$\frac{P \cup \{M = N \leftarrow \Gamma, L = R, \Delta\}}{P \cup \{(M = N \leftarrow \Gamma, L = R[u_i \leftarrow N_i], \Lambda_i, \Delta)\theta_i \mid 1 \leqq i \leqq k\}}$$

such that $(R/u_i)\theta_i \equiv M_i\theta_i$, for each $M_i = N_i \leftarrow \Lambda_i \in S \cup P$.

The Folding Rule

The folding rule folds equational clauses in P with definitions in D

$$\frac{P \cup \{M = N \leftarrow \Gamma, \Lambda\theta, \Delta\}}{P \cup \{M = N \leftarrow \Gamma, (f(X_1, \ldots, X_k) = Z)\theta, \Delta\}}$$

where $f(X_1, \ldots, X_k) = Z \leftarrow \Lambda \in D$.

The Substitution Rules

The substitution rules substitute a term K for every occurrence of a variable X in a clause in P, whenever the equation $X = K$ ($K = X$) appears in the body part.

$$\frac{P \cup \{M = N \leftarrow \Gamma, X = K, \Delta\}}{P \cup \{M = N \leftarrow \Gamma, \Delta\}(K/X)} \qquad \frac{P \cup \{M = N \leftarrow \Gamma, K = X, \Delta\}}{P \cup \{M = N \leftarrow \Gamma, \Delta\}(K/X)}$$

where X does not appear in K.

The Elimination Rule

The elimination rule eliminates an identity equation $K = K$ from the body part of a clause in P.

$$\frac{P \cup \{M = N \leftarrow \Gamma, K = K, \Delta\}}{P \cup \{M = N \leftarrow \Gamma, \Delta\}}$$

The Splitting Rule

This rule is used to decompose a complex equation into several component equations in the body part of the clause

$$\frac{P \cup \{M = N \leftarrow \Gamma, c(M_1, \ldots, M_n) = c(N_1, \ldots, N_n), \Delta\}}{P \cup \{M = N \leftarrow \Gamma, M_1 = N_1, \ldots, M_n = N_n, \Delta\}}$$

where c is a constructor.

The Deletion Rule

This rule allows the deletion of a clause that contains mismatching constructors in an equation in its body

$$\frac{P \cup \{M = N \leftarrow \Gamma, c(M_1, \ldots, M_n) = c'(N_1, \ldots, N_n), \Delta\}}{P}$$

where c and c' are distinct constructors.

The following definitions are technical to prove the partial correctness of the generation procedure.

Definition Let Q be a set of equational definite clauses.

1. An equational clause $M = N \leftarrow M_1 = N_1, \ldots, M_n = N_n$ is *emulated* by Q iff for any ground substitution θ, $M\theta \rightarrow_Q N\theta$ if $M_i\theta \downarrow_Q N_i\theta$, for every i, $1 \leqq i \leqq n$.
2. A set P of equational clauses is e*mulated* by Q iff every clause in P is *emulated* by Q.

Lemma 1. *Let P and Q be sets of equational clauses such that P is emulated by Q. Then $M *\rightarrow_P N$ implies $M *\rightarrow_Q N$, for every ground term M, N.*

Proof. Suppose $M *\rightarrow_P N$, then there is a proof figure π which proves $P \vdash M \geqq N$ by definition. Without loss of generality, we can chose a proof figure π such that every term appearing in π is a ground term. The lemma is verified by structural induction on π.

Suppose $M \geqq N$ is proved from P by applying (*modus ponens*) depicted as:

$$\frac{M_1\theta \geqq K_1 \quad N_1\theta \geqq K_1 \quad \ldots \quad M_n\theta \geqq K_n \quad N_n\theta \geqq K_n}{M \geqq N}$$

where $M = L\theta$ and $N = R\theta$ for some clause $L = R \leftarrow M_1 = N_1, \ldots, M_n = N_n \in P$. By induction hypothesis, $M_i\theta *\rightarrow_Q K_i$ and $N_i\theta *\rightarrow_Q K_i$, hence $M_i\theta \downarrow_Q N_i\theta$, for every i, $1 \leqq i \leqq n$. Since P is emulated by Q, $M \rightarrow_Q N$. Hence, $M *\rightarrow_Q N$.

The other cases are similar to the above and left to the reader. []

Lemma 2. *Suppose $<S; D'; P'>$ is obtained from $<S; D; P>$ by applying a single rule. If P is emulated by $S \cup D$, then P' is emulated by $S \cup D'$.*

Proof. Let $<S; D'; P'>$ be the triple that results from $<S; D; P>$ by applying a single rule. Now, suppose P is emulated by $S \cup D$. We have several cases depending on which rule is applied. Say it is the *lhs Unfolding Rule*. Suppose the situation is depicted as:

$$\frac{S; D; P_1 \cup \{M = N \leftarrow \Gamma, L = R, \Delta\}}{S; D; P_1 \cup \{(M = N \leftarrow \Gamma, L[u_i \leftarrow N_i] = R, \Lambda_i, \Delta)\theta_i \mid 1 \leqq i \leqq k\}}$$

where $(L/u_i)\theta_i \equiv M_i\theta_i$, for each $M_i = N_i \leftarrow \Lambda_i \in S \cup P$. Note that every equational clause in P_1 is emulated by $S \cup D$ by assumption. Let θ be any ground substitution. Suppose that $K\theta_i\theta \downarrow_{S \cup D} K'\theta_i\theta$, for every equation $K = K'$ in Γ, Λ_i, Δ and $L[u_i \leftarrow N_i]\theta_i\theta \downarrow_{S \cup D} R\theta_i\theta$.

If $M_i = N_i \leftarrow \Lambda_i \in S$, then $M_i\theta_i\theta \rightarrow_{S \cup D} N_i\,\theta_i\theta$ by definition. If $M_i = N_i \leftarrow \Lambda_i \in P$, then $M_i\theta_i\theta \rightarrow_{S \cup D} N_i\,\theta_i\theta$ since P is emulated by $S \cup D$. In either case, $M_i\theta_i\theta \rightarrow_{S \cup D} N_i\,\theta_i\theta$, hence $L\theta_i\theta \rightarrow_{S \cup D} L[u_i \leftarrow N_i]\theta_i\theta$. This implies $L\theta_i\theta \downarrow_{S \cup D} R\theta_i\theta$ since $L[u_i \leftarrow N_i]\theta_i\theta \downarrow_{S \cup D} R\theta_i\theta$. Hence, $M\theta_i\theta \rightarrow_{S \cup D} N\,\theta_i\theta$. Because $M = N \leftarrow \Gamma, L = R, \Delta$ is emulated by $S \cup D$. These discussions show the equational clause $(M = N \leftarrow \Gamma, L[u_i \leftarrow N_i] = R, \Lambda_i, \Delta)\theta_i$ is emulated by $S \cup D$, for every i, $1 \leqq i \leqq k$.

The other cases can be verified similarly to the above. This completes the proof. []

Now, we will prove the partial correctness of the generation procedure.

Theorem 1. *Let HS be a Horn clause specification with a main predicate p and S be the set of equational definite clauses converted from HS. Suppose the procedure produces a rewriting program P as a result from the initial setting* $<S; \{f_p(X_1, \ldots, X_k) = Z \leftarrow p(X_1, \ldots, X_k, Z)\}; \{f_p(X_1, \ldots, X_k) = Z \leftarrow p(X_1, \ldots, X_k, Z)\}>$. *For any ground terms* $t_1, \ldots, t_k$, *and* t, *if* $f_p(t_1, \ldots, t_k) \ {}^*\Rightarrow_P t$ *(in P), then* $p(t_1, \ldots, t_k, t) \in Model(HS)$.

Proof. Let $<S; D; P>$ be the last structure returned by the procedure. It is trivial that $f_p(X_1, \ldots, X_k) = Z \leftarrow p(X_1, \ldots, X_k, Z)$ is emulated by $S \cup \{f_p(X_1, \ldots, X_k) = Z \leftarrow p(X_1, \ldots, X_k, Z)\}$. By using Lemma 2 same times as the number of applying generation rules, it follows that P is emulated by $S \cup D$. Thus, by Lemma 1, $M \ {}^*\rightarrow_{S \cup D} N$ if $M \ {}^*\rightarrow_P N$, for every ground term M, N. Suppose $f_p(t_1, \ldots, t_k) \ {}^*\Rightarrow_P t$, where $t_1, \ldots, t_k$, *and* t are ground terms. Proposition 3 claims that if Q is a set of equations two reduction relations ${}^*\Rightarrow_Q$ and ${}^*\rightarrow_Q$ are identical. Hence, $f_p(t_1, \ldots, t_k) \ {}^*\rightarrow_P t$. Since P is emulated by $S \cup D$, $f_p(t_1, \ldots, t_k) \ {}^*\rightarrow_{S \cup D} t$. Recall the definition $f_p(X_1, \ldots, X_k) = Z \leftarrow p(X_1, \ldots, X_k, Z) = true$ specifying the defined function symbol f_p is unique in D and every defined function symbol appearing in D does not appear in S. So that $p(t_1, \ldots, t_k, t) \ {}^*\rightarrow_S true$ must hold. Thus the ground atom $p(t_1, \ldots, t_k, t)$ belongs to the least Herbrand model $Model(HS)$ by Proposition 2.

4. CONCLUDING REMARKS

The approach outlined here is an attempt to generate machine executable programs from specifications written in higher level languages. The early works in program generation relied strongly on theorem proving techniques. On the other hand, this approach is based on program transformation techniques. Several generation rules, a kind of transformation rules, are introduced to generate rewriting programs from Horn clause specifications.

As further works (or as continuation of this work), this paper may involve further considerations: proving the total correctness of the generation process, extending the specification language or the target language, or refinement of the generation process. However, we believe that out approach is more suitable than theorem proving approach to design a mechanical algorithm generating programs form specifications. We hope that this paper has opened up an new idea for further studies in this field.

REFERENCES

1. Chang, A. K. and Lee, R. C. T., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

2. Fribourg, L. , Oriented equational clauses as a programming language, *J. Logic Programming*,1, pp.165-177, 1984.

3. Goguen, J. A. and Meseguer, J., Equality, Types, Modules and Generics for Logic Programming, *J. Logic Programming*, 1: 179-210 (1984).

4. Hsiang, J. and Plaisted, D., Deductive Program Generation, draft paper , 1985.

5. Hoffmann, C. M. and O'Donnell, M. J., Programming with Equations, *ACM Trans. on Prog. Lang. and Systems*, 4: 83-112 (1982).

6. Hogger, C. J., Derivation of Logic Programs, *J. ACM*, 28: 372-392 (1981).

7. Huet, G., Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems, *J. ACM*, 27: 797-821 (1980).

8. Huet, G. and Oppen, D. C., Equations and Rewrite Rules, *in Formal Language Theory*, Academic Press, 349-405 (1980).

9. Lloyd, J. W., *Foundation of Logic Programming*, Springer-Verlag, 1984.

10. Manna, Z. and Waldinger, R., A Deductive Approach to Program Synthesis, *ACM Trans. on Programming Language and Systems*, 2, pp.90-121, 1980.

11. Murray, N.V., Completely Non-Clausal Theorem Proving, *Artificial Intelligence*, 18, pp.67-85, 1982.

12. O'Donnell, M. J., Computing in Systems described by Equations, *Lec. Notes in Computer Science*, No. 58, 1977.

13. Reddy, U. S., On the Relationship between Logic and Functional Languages, in DeGroot, D. and Lindstrom, G. (eds.), *Logic Programming, Functions, Relations and Equations*, Prentice-Hall, 3-36 (1985).

14. Tamaki, H. and Sato, T., Compatibility of Replacement Rules with Unfold / Fold Transformation, draft paper, 1986.

15. Togashi, A. and Noguchi, S., A Program Transformation from Equational Programs into Logic Programs, to appear in *J. Logic Programming*.