PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

A PROGRAM TRANSFORMATION FROM EQUATIONAL PROGRAMS INTO LOGIC PROGRAMS

Atsushi TOGASHI and Shoichi NOGUCHI

Research Institute of Electrical Communication, Tohoku University
2-1-1 Katahira-cho, Sendai 980 Japan

ABSTRACT

In the last years substantial efforts have been made to develop equational programming languages and logic programming languages so called descriptive languages. Both languages are based on some mathematical systems and show certain similarities each other. This indicates some possibility of program transformation. As for program transformation, the equational language concerns with an algebraic specification for abstract data types and a recursive program scheme. The transformation provides introduction of notions such as data abstraction and computation strategies in a logic programming language.

In this paper we propose a transformation algorithm from equational programs written in equational form into logic programs. In order to facilitate program transformation we extend the programming language Prolog which has been used as the promising language in the literature into a logic programming language based on a new computation model. It is shown that any equational program is transformed into an equal or more powerful logic program. For a recursive equational program, there exists a Horn program with the equivalent computational power.

1 INTRODUCTION

Descriptive languages for computation are often organized into two types, a logic programming language and an equational (or functional) programming language. Logic programming languages, such as the Prolog (Clocksin and Mellish 1981, Kowalski 1979), are based on the first order predicate logic, directly concerned with the resolution principle introduced by Robinson in (Robinson 1965). In the language programs are expressed as sentences, their computation process is deriving successively new subgoals from goals by the resolution. Solving problems is to show inconsistency of the given goals with respect to the programs, contradictory instances are the results of the computations. See (Apt and Emden 1982, Clark 1979, Emden and Kowalski 1976, Hogger 1981) for a theoretical treatment, also refer to (Kowalski 1979) for an application to the artificial intelligence.

Equational languages are oriented toward using equations to specify programs, however, form computational point of view equations may be thought of as reduction rules allowing the reduction of the left-hand expression to the right-hand expression. In this way their computation process is repeatedly reducing the given expressions by applying equations to the normal forms which can not be reduced any more. These expressions are the results of the computations. As many researchers noticed in (Hoffmann and O'Donnell 1982, Huet 1980, O'Donnell 1977, Rosen 1973), not only large parts of LISP in the real programming language, but the lambda calculus, the combinator calculus and recursively defined functions in mathematics may be viewed as the special systems of equations.

Though these two languages are based on the slightly different mathematical systems, they show deep similarity each other. This indicates certain possibility of program transformation. The equational language is much concerned with an algebraic specification for abstract data types (Goguen et al. 1978, Guttag et al. 1978) and a recursive program scheme (Arnold and Nivat 1980, Downey and Sethi 1976). The transformation provides

introduction of notions such as data abstraction and computation strategies in a logic programming language.

This paper presents a transformation algorithm from equational programs into logic programs. It is shown any equational program is transformed into an equal or more powerful logic program. When we restrict our attention to recursive equational programs which are the generalization of recursive program schemes any recursive equational program is simulated by some Horn program with the equivalent computational power. In order to facilitate program transformation we extend the programming language Prolog, which was more popular and investigated by many researchers, into a logic programming language based on a new computational model. This language is more suitable for representation of knowledge for predicates, and is oriented to knowledge based programming.

This paper is organized as follows : In chapter 2 some preliminary definition are discussed. The formulation of equational and logic programming languages are described in chapter 3 and in chapter 4, respectively. There some results are also investigated. In chapter 5 we propose a transformation algorithm and validity of the method is proved.

We believe our paper is the first attempt to clarify relationship among descriptive languages. We hope that our results have established a footing for further studies of this field.

2 PRELIMINARY DEFINITIONS

In this chapter we briefly survey some preliminary definitions which will be used throughout this paper. In particular, we state notions signatures, terms, substitutions, and term rewriting systems. See (Goguen et al. 1978, Huet 1980, Huet and Oppen 1980, O'Donnell 1977) for detail discussions.

2.1 Signatures, Terms and Substitutions

It is assumed that we are given a finite set S of *sorts*, which are the names for the various *data types* under consideration. A (S-sorted) *signature* is an indexed family $\{\Sigma_{w,s}\}_{(w,s) \in S^* \times S}$ of disjoint sets $\Sigma_{w,s}$ of symbols, where S^* denotes the set of all finite sequences on S with the *null string* Λ (S^+ is the set of all non null sequences on S). A symbol $\sigma \in \Sigma_{w,s}$ is called a *function symbol* of sort s with *arity* w, sometimes written $\sigma : w \dashrightarrow s$. When $w = \Lambda$, σ is called a *constant*. For ease of notation, let $\Sigma = \bigcup_{(w,s)} \Sigma_{w,s}$, and we use Σ to denote the signature.

Let $X = \bigcup_{s \in S} X_s$ be a disjoint union of denumerable sets X_s of *variables* of sort $s \in S$ and fixed throughout this paper. For a signature Σ, Σ - *terms* (or *terms* whenever Σ is clear from the context) of sort $s \in S$ are defined in the recursive way:

(1) $x \in X$ is a Σ-term of sort s;
(2) $\sigma : \Lambda \dashrightarrow s$ is a Σ -term of sort s;
(3) If $\sigma : s1,\ldots,sn \dashrightarrow s$ is a function symbol and ti are Σ-terms of sorts si, then $\sigma(t1,\ldots,tn)$ is a Σ -term of sort s.

The set of all Σ -terms of sort s is denoted by $T(\Sigma,X)_s$. We define $T(\Sigma,X)$ as the disjoint union of the sets $T(\Sigma,X)_s$ for $s \in S$.

Let N^* be a set of all strings on the set N of positive integers with the null string λ. We shall call the members of N^* *occurrences* and denote them u, v and w, possibly with subscripts. For a Σ-term t we denote by $Ocr(t) \subset N^*$ its set of occurrences and by t/u the *subterm* of t at the occurrence $u \in Ocr(t)$. We say u is the occurrence of the subterm t/u in t. We use Var(t) to denote a set of variables occurring in t, that is, $x \in Var(t)$ if and only if $x \in X$ and there exists $u \in Ocr(t)$ such that t/u = x. For terms t, t' and u in Ocr(t), we define t[u <-- t'] as the term t, in which the subterm t/u at the occurrence u is replaced by t'.

A *substitution* is a mapping θ from the set of variables into the set of terms such that $\theta(x) = x$ almost everywhere, that is the *domain* of θ defined by $Dom(\theta) = \{ x \mid \theta(x) \neq x \}$ is finite. Here we assume that substitutions are sort-preserving, i.e., all variables of sort s are mapped into the terms of same sort. The substitution θ is extended into the terms by

$$\theta(\sigma(t1,\ldots,tn)) = \sigma(\theta(t1),\ldots,\theta(tn))$$

where $\sigma : s1,\ldots,sn \dashrightarrow s$ is the function symbol and ti are the terms of sorts si.

2.2 Term Rewriting Systems

Definition 1 A *term rewriting system* on a signature Σ is a finite set R of *rewriting rules* of the form l --> r such that $Var(l) \supset Var(r)$, where l and r are the Σ-terms of the same sort.

R may be *applicable* to a term t iff there is an occurrence $u \in Ocr(t)$ such that $t/u = \theta(l)$ for some rule l --> r in R and for some substitution θ. In this case, we say that the rule l --> r is applied to the term t to obtain the term $t[u \leftarrow \theta(r)]$. The choice of which rules to apply is made non deterministic. We write $t \Rightarrow_R t'$ to indicate that the term t' is obtained from the term t by a single application of some rule in R. Let $\overset{*}{\Rightarrow}_R$ denote the reflexive and transitive closure of $\Rightarrow_R$. If $t \overset{*}{\Rightarrow}_R t'$ holds we say t' is *derivable* from t in R. R may be omitted from $\overset{*}{\Rightarrow}_R$ and $\Rightarrow_R$ when it is clear from the context. The derivation relation is characterized in a proof system in the following way. See (Huet 1980, Huet and Hullot 1982, O'Donnell 1977, Rosen 1973) for related discussions.

Proposition 1 Let R be a term rewriting system on Σ and t, t' be any Σ-terms. Then $t \overset{*}{\Rightarrow} t'$ holds if and only if the ordered pair of the terms $t \geq t'$ is provable in a *proof system* with the following *inference rules*.

(1) $\dfrac{l \text{ --> } r \in R}{l \geq r}$ (2) $\dfrac{}{t \geq t}$

(3) $\dfrac{t \geq t',\ t' \geq t''}{t \geq t''}$

(4) $\dfrac{ti \geq t'i,\quad \sigma : s1,\ldots,sn \text{ --> } s}{\sigma(t1,\ldots,tn) \geq \sigma(t'1,\ldots,t'n)}$

(5) $\dfrac{t \geq t',\quad \theta : X \text{ --> } T(\Sigma,X)}{t\theta \geq t'\theta}$

proof. Both directions can be easily verified by induction, so we omit the proof.

Remark that the notation $t \geq t'$ for the ordered pairs comes from the fact that the ordered pairs provable in the proof system are characterized by the partial ordering relation on terms. See (Huet 1980, Huet and Oppen 1980).

3 EQUATIONAL PROGRAMS

We formulate an equational program in the framework of a term rewriting system as in (Hoffmann and O'Donnell 1982). The theoretical foundations for computing with equations are treated in detail in (Huet 1980, Huet and Oppen 1980, O'Donnell 1977, Rosen 1973). Hoffmann and O'Donnell (Hoffmann and O'Donnell 1982) illustrated the usefulness of equational programs, and investigated practical solutions to the problems involved in implementing equational programs.

Let Σ be a (finite) S-sorted signature. Following (Huet and Hullot 1982) we assume that the signature Σ is partitioned as $\Sigma = \Sigma^c \cup \Sigma^d$. We shall call the function symbol in Σ^c the *constructor* , and the member in Σ^d the *defined function symbol* .

Definition 2 An *equational program* on the signature Σ is a term rewriting system R in which each rewriting rule is of the form $F(E1,\ldots,En) \text{ --> } En+1$, where F is the defined function symbol and Ei the are Σ-terms.

Constructors create concrete data types. Defined function symbols define some manipulations over the constructed data types; the meaning of them are described by rewriting rules. For notational convenience, the constructors are denoted by the lower case letters, and the defined function symbols by the capital letters such as F, G, H and so on. Similarly we use symbols E, Ei to denote Σ-terms and t, ti to denote Σ^c-terms, called *constructor term* , constructed only by constructors. Of course, both kinds of terms contain variables as constituents.

A certain restriction can be made on the nature of the rewriting rules to give more restricted class of equational programs.

Definition 3 An equational program R is *recursive* if every rule in R is of the form $F(t1,\ldots,tn) \text{ --> } E$

The recursive equational program can be viewed as the generalization of non deterministic recursive program scheme in (Arnold and Nivat 1980).

Let R be an equational program on Σ. A *computation* from an *input term* E0 is a possibly infinite derivation sequence

$$E0 \Rightarrow_R E1 \Rightarrow_R \cdots \Rightarrow_R En \Rightarrow_R \cdots .$$

The computation *successfully terminates* if En is a constructor term t for some $n \geq 0$, hence we can not rewrite En any more by the definition of equational programs. In this case, En = t is the *result* of this computation. Otherwise the computation fails, i.e., it terminates at the term E which contains some defined function symbols or never terminates.

Example 1 An equational program R reversing lists

```
constructors :
     nil : Λ --> list;
     cons : item, list --> list;

defined function symbols :
     Append : list, list --> list;
     Rev : list --> list

rewriting rules :
     Append(nil, x) --> x
     Append(cons(i,x), y
        --> cons(i, Append(x,y))
     Rev(nil) --> nil
     Rev(cons(i,x))
        --> Append(Rev(x), cons(i,nil))
     Rev(Rev(x)) --> x
     Append(Append(x,y), z)
        --> Append(x, Append(y,z))
     Rev(Append(x,y))
        --> Append(Rev(y), Rev(x))
```

What we have defined above is the most general strategy of executing programs. A more restricted strategy is treated here to be simulated by logic programs discussed later. Let R be an equational program. A Σ-term E' is derivable from a Σ-term E in a *primitive execution strategy* denoted by $E \Rightarrow_{(p)} E'$ if there exist a rule $l \to r$, an occurrence $u \in Ocr(E)$ and substitution $\theta : X \to T(\Sigma^C, X)$ with the range the set of only constructor terms such that $E/u = \theta(l)$ and $E[u \leftarrow \theta(r)] = E'$. The computation from E0 in the this strategy and the result for it are defined similarly to the general case.

Corollary 1 Let R be an equational program and E, E' be Σ-terms. $E \overset{*}{\Rightarrow}_{(p)R} E'$ holds if and only if $E \geq E'$ is provable by applying the inference rules (1), (2), (3), (4) mentioned in Proposition 1 together with the rule,

$$(5') \qquad \frac{E \geq E', \quad \theta : X \to T(\Sigma^C, X)}{E\theta \geq E'\theta}$$

4 LOGIC PROGRAMS

In the last few years the programming language Prolog (Apt and Emden 1982, Emden and Kowalski 1976, Kowalski 1979) based on the Horn clauses in the first order logic has been increasingly used, due to the possibility of suitably using it as a specification language (Hogger 1981) and as a practical, efficient programming language (Clocksin 1981). In order to facilitate transformation from equational programs into logic programs, we extend the Prolog into a new logic programming language to have more than one atoms in their left-hand sides of the Horn clauses. Also we introduce inferred variables which will be distinguished from fixed variables. As an example, the left distributive law of the multiplication M(ULT) for the addition A(DD) can be expressed as

```
M(x,y1,*u), M(x,y2,*v), A(*u,*v,z)
     :- M(x,*w,z), A(y1,y2,*w).
```

This corresponds to the usual distributive law of the multiplication "." over the addition "+" written by the equation

$$x \cdot y_1 + x \cdot y_2 = x \cdot (y_1 + y_2).$$

This formula has more than one atoms at the left-hand side and the variables appearing in it are partitioned into two kinds of variables. One is a fixed variable such as x or z, bounded by universal quantifier $\forall$ from outside, the other is an inferred variable such as *u or *v, bounded by existential quantifier $\exists$ from inside. The above formula can be expressed by the usual form such as:

```
∀ x,∀ y1,∀ y2,∀ z:
∃ u, ∃v : M(x,y1,u), M(x,y2,v), A(u,v,z)
   <-- ∃ w : M(x,w,z), A(y1,y2,w).
```

This formula asserts a single concept, hence can not be modified into more than one definite clauses (Apt and Emden 1982) without loosing the meaning it has. This kind of property can be used to simplify subgoals and speed up its computation. The application of the above property is allowed only to subgoals in which there are some atoms identified with its whole left hand side of the rule by

two sorts of substitutions, and results in a parallel rewriting of atoms. In this way, we have modified the way subgoals are computed according to the above extension, so that properties will be applied to get an intelligent, efficient computation.

A (S-sorted) *similarity type* is a pair $d = (\Sigma^c, \Gamma)$, where Σ^c is a S-sorted signature and Γ is a disjoint union of sets Γ_w of predicate symbols P : w for $w \in S^+$.

Definition 4 Let d be a similarity type.
(1) An *atom* is P(t1,...,tn), where P : s1,...,sn is the predicate symbol and ti are the terms of sorts si.
(2) A *cluster formula* (or *cluster*) is a finite set of atoms.

For a cluster M, Var(M) denotes a set of all variables appearing in M. There are two kinds of variables, that is *fixed* variables and *inferred* variables, which correspond the variables bounded by universal quantifier $\forall$ and by existential quantifier $\exists$, respectively. We assume that the set Var(M) is partitioned into the set FIX(M) of fixed variables and the set INF(M) of inferred variables. For simplicity, we will write a cluster C1,...,Ck rather than {C1,...,Ck}. For this reason, the order of atoms in the cluster is not crucial. If x1,...,xm and y1,...,yn are fixed variables and inferred variables of the cluster M = C1,...,Ck, we can read it as "for all x1,...,xm there exist y1,...,yn such that C1 and ... and Ck.

Definition 5 A *cluster sequent* on a similarity type d is an ordered pair of clusters of the form M :- N which satisfies the following two conditions:
(a) All fixed variables appearing in the right hand side also appear in the left hand side, i.e., FIX(M)$\supset$FIX(N);
(b) there is no common variable among FIX(N), INF(M) and INF(N).

The cluster M is called the *conclusion* of the sequent M :- N; N the *premise* of it. For a cluster r = A1,...,Am :- B1,...,Bn, let FIX(A1,...,Am) = { x1,...,xk }, INF(A1,...,Am) = {y1,...,yp}, and INF(B1,...,Bn) = {z1,...,zq}. The cluster sequent r can be interpreted as "for all x1,...,xk; if there exist z1,...,zq such that B1 and ... and Bn, we can assert existence of y1,...,yp such that A1 and ... and Am".

Definition 6 A *definite* sequent is a cluster sequent of the form A :- B1,...,Bn such that INF(A) = $\emptyset$.

Definite sequents correspond to the definite clauses. This alternative formulation of the definite clauses derives from the fact that a universally quantified implication $(\forall x)$: A <-- B1,...,Bn is logically equivalent to the one A <-- $(\exists x)$: B1,...,Bn when x does not occur in A.

Definition 7 A *logic program* on a similarity type d is a finite set £ of cluster sequents. If £ consists only of definite sequents, £ is said to be a *Horn program*.

A *goal* for a logic program is a cluster. Goals describe the problems which will be solved by the execution of programs. In the procedure interpretation a logic program is a *goal reduction* (*replacement*) *system* as discussed in (Kowalski 1979). A computation (or an execution) of programs is initiated by giving an input goal. The computation proceeds by applying some cluster sequents to derive successive new subgoals. In each computation step some subcluster is selected from the subgoal and matched with the left hand side of some cluster sequent by finding two kinds of appropriate substitutions. The subcluster is then replaced by the right hand side of the cluster sequent. The computation successfully terminates for the input goal if the empty goal (terminal goal) is derived. In the following, we will formalize a way goals are computed according to the extention mentioned above.

Definition 8 Let £ be a logic program. A cluster sequent r = A1,...,Am :- B1,...,Bn in £ is *applicable* to a goal M = C1,...,Ck if and only if there exist two substitution θ, η with the conditions

Dom(θ) FIX(A1,...,Am),
Dom(η) INF(C1,...,Ck),

called a *matching* and an *inferring* substitution, respectively such that $(C1,...,Cm)\eta = (A1,...,Am)\theta$ for some subcluster C1,...,Cm of C1,..,Ck.

If so, we say r is *applied* to obtain a new goal N = $(B1,...,Bn)\theta$, $(Cm+1,...,Ck)\eta$. The inferred variables of the newly derived subgoal N are defined by

$$INF(N) = \{ x \in INF(M) \mid x \notin Dom(\eta) \cup INF(B1,\ldots,Bn) \cup \{ x \in Var(A1,\ldots,Am) \mid x \notin Dom(\theta) \}.$$

All other variables appearing in N are fixed variables.

Example 2 Let us consider a logic program consisting of a single cluster sequent A(x,z,f(x,*v)) :- A(g(z),*w,x). In order to distinguish inferred variables with fixed variables we use a symbol "*" in such a way *x stands for that x is an inferred variable. Given a goal A(h(x,*u),*u,*y), B(*y,x) £ is applicable to it, and we obtain the new goal A(g(*z),*w,h(x,*z)), B(f(h(x,*z),*w),x) as the result of application.

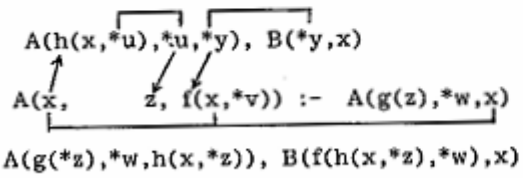

Fig.1 An application of a cluster sequent

For goals M, N $M \Rightarrow_{\mathcal{L}} N$ indicates that N is obtained from M by a single application of some sequent in £. We may write $M \Rightarrow_{\mathcal{L}}^{\eta} N$ to specify the used inferring substitution η. $\overset{*}{\Rightarrow}$ denotes the reflexive, transitive closure of $\Rightarrow$. If $M \overset{*}{\Rightarrow}^{\eta} N$ holds, we say M is *reducible* to N, where η is the composition of used inferring substitutions.

Proposition 2 Let £ be a logic program and M, N be goals. For any substitution ζ with the domain contained by INF(M), we have $M\zeta \Rightarrow^{\eta} N$ implies $M \Rightarrow^{\zeta\eta} N$.

Corollary 2 Under the same condition as Proposition 2 it follows that $M\zeta \overset{+}{\Rightarrow}^{\eta} N$ implies $M \overset{+}{\Rightarrow}^{\zeta\eta} N$, where $\overset{+}{\Rightarrow}_{\mathcal{L}}$ is the transitive closure of $\Rightarrow_{\mathcal{L}}$.

Conversely, we can easily verify the following Proposition by induction on the length of reductions.

Proposition 3 $M\eta \overset{*}{\Rightarrow} N$ follows from the condition $M \overset{*}{\Rightarrow}^{\eta} N$ for all goals M, N.

Let £ be a logic program. A *computation* from a goal M is a reduction sequence

$$M = M0 \Rightarrow^{\eta 1} M1 \Rightarrow^{\eta 2} \ldots .$$

A computation *successfully terminates* if Mn is an empty goal, denoted by e, for some $n \geq 0$, where the *empty goal* is an empty cluster. In this case, the composition $\eta = \eta 1 \ldots \eta n$ is the *answer* substitution and Mη is the *result* for the computation.

5 A TRANSFORMATION ALGORITHM

To transform an equational program into a logic program, at first, we shall show a method for transforming a given Σ-term E into the *cluster* C(E) and the *output term* O(E) associated with E.

Let Σ be a signature for equational programs. The corresponding similarity type $d = (\Sigma^c, \Gamma)$ for logic programs is specified in the following way:

(a) The set Σ^c is identical to the set of constructors in Σ;

(b) The set Γ is defined by $\{F_p : s1,\ldots,sn,s \mid F : s1,\ldots,sn \rightarrow s \in \Sigma^d\}$.

Algorithm A We associate a cluster C(E) and an output term O(E) with a Σ-term E by structural induction on E. For a cluster C(E) the fixed variables are ones which belong to Var(E) and newly introduced variables are the inferred variables.

(1) If E is a variable or constant, define
$$C(E) = e; \quad O(E) = E.$$

(2) Suppose E is of the form σ(E1,...,En). By induction hypothesis, it is assumed that the clusters C(Ei) and the output terms O(Ei) are constructed in such a way that the newly introduced variables are standardized apart one another.

(a) If σ is a constructor f, define

$$C(E) = C(E1),\ldots,C(En);$$
$$O(E) = f(O(E1),\ldots,O(En)).$$

(b) If σ is a defined function symbol F,

$$C(E) = C(E1),\ldots,C(En), F_p(O(E1),\ldots,O(En),*y);$$
$$O(E) = *y,$$

where F_p is the predicate symbol corresponding to F and *y is the new variable never appears in C(Ei) for all $1 \geq i \geq n$.

Example 3 Let us consider a Σ-term

$$E = f(F(G(x), g(a)), H(G(x))),$$

where f, g, a are the constructors (a is the constant) and F, G, H are the defined function symbols with arbitrary types. By applying Algorithm A to this Σ-term we obtain the cluster C(E) and the output term O(E).

$$C(E) = G_p(x,*y3),\ F_p(*y3,g(a),*y1),\ G_p(x,*y4),\ H_p(*y4,*y2)$$
$$O(E) = f(*y1,\ *y2).$$

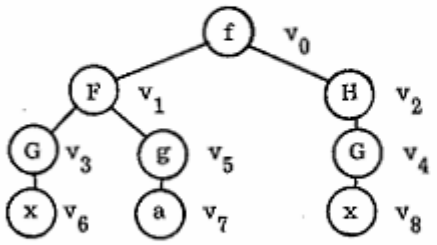

Fig.2 A tree representing a Σ-term E.

By using Algorithm A a transformation algorithm from equational programs into logic programs is described in the following way:

Algorithm B A Transformation Algorithm
Let R be a given equational program. We translate each rewriting rule F(E1,...,En) --> E' in R into a cluster sequent to construct the corresponding logic program.

(i) Constitute the clusters and the output terms for both sides of the rule by using Algorithm A. During execution of it the newly defined variables are standardized apart in both sides. (Let note that the output term of the left hand side must be a variable.)

(ii) The transformed cluster sequent is defined by

$$C(E1),\ldots,C(En),\ F_p(O(E1),\ldots,O(En),O(E')) :- C(E').$$

The set of fixed variables of it is specified by Var(F(E1,...,En)) U Var(O(E')), and the others are inferred variables.

Example 4 If we apply Algorithm B to the equational program in Example 1, we have obtained the following logic program.

```
APPEND(nil, x, x) :- e
APPEND(cons(i,x), y, cons(i,z))
   :- APPEND(x,y,z)
APPEND(x,y,*u), APPEND(*u,z,w)
   :- APPEND(y,z,*v), APPEND(x,*v,w)
REV(nil, nil) :- e
REV(cons(i,x), z)
  :- REV(x, *y),
       APPEND(*y, cons(i,nil),z)
REV(x,*y), REV(*y,x) :- e
APPEND(x,y,*w), REV(*w,z)
   :- REV(y,*v), REV(x,*u),
        APPEND(*v,*u,z)
```

Proposition 4 If R is a recursive equational program, the transformed logic program is a Horn program.

To investigate the relationship between equational programs and the translated logic programs, we shall consider the clusters and the output terms associated with Σ-terms by Algorithm A. In Algorithm A we constructed a cluster C(E) and an output term O(E) for each Σ-term E. Similar to clusters, we partitioned the set Var(O(E)) into fixed variables and inferred variables. Here the fixed variables are the ones which belong to Var(E). For constructor terms t, t' with variables partitioned t is a *variant* of t' if t differs from t' at most in the names of its inferred variables. A variant of a cluster is defined similar to the variant of the constructor term.

Theorem 1 Let R be an equational program and $\mathcal{L}$ the transformed logic program from R. For any Σ-terms E, E' if $E \overset{*}{\underset{(p)}{\Rightarrow}} E'$ in the equational program R, then there exist some variants M' and t' of C(E') and O(E'), respectively, such that $C(E) \overset{*}{\Rightarrow}_{\eta} M'$ in the logic program $\mathcal{L}$ and $O(E)\eta = t'$ for some inferring substitution η.

proof. By Corollary 1 the proof consists of examining each of the rules of inference. As for the rules of inference (1), (2) and (3) the assertion of the theorem is obviously clear from the transformation algorithm. So we discuss only the inference rules (4) and (5').

for the inference rule (4) : Suppose that the given terms are of the form

$$E = \sigma(E1,\ldots,En),\quad E' = \sigma(E'1,\ldots,E'n)$$

for some function symbol σ. By structural induction we can assume that

$$C(Ei) \overset{*}{\Rightarrow}_{\eta i} M'i,\quad O(Ei)\eta i = t'i$$

for some variants M'i, t'i of C(E'i), O(E'i), and for some inferring substitutions ηi for all

$1 \leq i \leq n$. Without loss of generality we can impose the conditions on the clusters and the output terms in such a way that

$$INF(Mi) \cap INF(Mj) = \emptyset,\ INF(ti) \cap INF(tj) = \emptyset;$$
$$INF(Mi) \cap INF(M'j) = \emptyset;\ INF(ti) \cap INF(t'j) = \emptyset$$

for all $i \neq j$, where $Mi = C(Ei)$, and $ti = O(Ei)$. There are two possibilities for σ as the function symbol.

(a) If σ is a constructor f, the associated clusters C(E), C(E') and the output terms O(E), O(E') must be of the form

$$C(E) = C(E1),\ldots,C(E);$$
$$C(E') = C(E'1),\ldots,C(E'n),$$

and

$$O(E) = f(O(E1),\ldots,O(En));$$
$$O(E') = f(O(E'1),\ldots,O(E'n)),$$

respectively. Let us define an inferring substitution η as the composition $\eta = \eta 1 \ldots \eta n$. By the assumption described above we have

$$\begin{aligned} C(E) &= C(E1),\ldots,C(En) \\ &\overset{*}{\Rightarrow}_{\eta 1} M'1,C(E2),\ldots,C(En) \\ &\quad \vdots \\ &\overset{*}{\Rightarrow}_{\eta n} M'1,M'2,\ldots,M'n, \end{aligned}$$

and

$$\begin{aligned} O(E) &= f(O(E1),\ldots,O(En))\eta 1 \ldots \eta n \\ &= f(O(E1)\eta 1,O(E2),\ldots,O(En))\eta 2 \ldots \eta n \\ &= f(t'1,O(E2),\ldots,O(En))\eta 2 \ldots \eta n \\ &\quad \vdots \\ &= f(t'1,\ldots,t'n) \end{aligned}$$

which are the variants of C(E') and O(E'), respectively.

(b) If σ is a defined function symbol F, then the associated clusters and the output terms are of the form

$$C(E) = C(E1),\ldots,C(En),\ F_p(O(E1),\ldots,O(En),*y)$$
$$C(E') = C(E'1),\ldots,C(E'n),\ F_p(O(E'1),\ldots,O(E'n),*y')$$

and

$$O(E) = *y,\quad O(E') = *y'$$

where F_p is the predicate symbol corresponding to the defined function symbol F and *y, *y' are the new inferred variables. As in case (a), we can easily verify that

$$C(E) \overset{*}{\Rightarrow}_{\eta} M' \quad \text{and} \quad O(E)\eta = t'$$

for some variants M', t' of C(E'), O(E'), respectively, and for some inferring substitution η, so we omit the proof of them.

for inference rule (5') : Finally for given Σ-terms E, E' suppose that

$$C(E) =>^{\eta 1} M1 => \ \ldots => \ Mk-1 =>^{\eta k} M',$$

and

$$O(E)\eta 1 \ldots \eta k = t'$$

for some variants M', t' of C(E'), O(E'), respectively. Let $\theta : X \dashrightarrow T(\Sigma^C, X)$ be any substitution with the range the set of constructor terms. Define inferring substitutions $\zeta i = \eta i\theta$, $1 \leq i \leq n$, which map the inferred variables $*y \in Dom(\eta i)$ to the terms $(*y)\eta i\theta$. It is obviously clear by the assumption that

$$C(E\theta) =>^{\zeta 1} \ => \ \ldots =>^{\zeta k} \ M'\theta,$$

and

$$O(E\theta)\zeta 1 \ldots \zeta k = E'\theta.$$

Hence the proof is complete.

Corollary 3 Let E be a Σ-term and t a constructor term. If $E \overset{*}{\Rightarrow}_{(p)} t$ in the equational program R, then $C(E) \overset{*}{\Rightarrow}_{\eta} e$ and $O(E)\eta = t$ in the corresponding logic program £ for some inferring substitution η.

Without loss of generality we can assume that input terms in equational programs are of the form F(t1,...,tn), where ti are the constructor terms. By Proposition 3 and Corollary 3 we have

Corollary 4 If $F(t1,\ldots,tn) \overset{*}{\Rightarrow} t$ in an equational program R, then $F_p(t1,\ldots,tn,t) \overset{*}{\Rightarrow} e$ in the corresponding logic program.

These results indicate that in a primitive execution strategy any equational program is transformed into an equal or more powerful logic program. On the other hand, for a recursive equational program, we can translate into a logic program with the equivalent computation power.

Let W denote a set of all atoms (containing variables) on a similarity type d. With a Horn program £ we associate a mapping T over the power set P(W) of W.

Definition 9 Given a Horn program £, a *mapping* T over P(W) associated with £ is defined as follows : For any subset $V \subset W$

and for any definite sequent in £ B0 :- B1,...,Bn, if there exists a substitution θ such that Biθ V for all i, 1 ≤ i ≤ n, then we have B0θ εT(V).

By definition T is the *continuous* mapping over P(W) with the partial order set-theoretic inclusion among subsets on W. So T has the unique *fixed point* lfp(£) like as the result in (Emden and Kowalski 1976). In fact, lfp(£) turns out to be lfp(£) = $U_{k\geq 1} T^k(\emptyset)$, where ∅ is the empty subset of W. The following the slight modification of the result in (Apt and Emden 1982).

Theorem 2 Let £ be a Horn program and M a goal. If there is a successfully terminating computation from M with an answer substitution η, then Aη ε lfp(£) for every atomic cluster A in M.

proof. Let M0 $\Rightarrow^{\eta 1}$ M1 => ... $\Rightarrow^{\eta k}$ Mk be a successful computation from M with an answer substitution η. Note that M0 = M, Mk = e and η = η1...ηk. To prove the theorem we show by induction on i ≥ 1 that A η k-i+1 ...ηk ε $T^i(\emptyset)$ for any atomic A in Mk-i,

If i = 1, then Mk-1 consists of a single atomic cluster, say A. By the assumption it follows that Aηk = B0θ for some definite sequent B0 :- e in £ and for some matching substitution θ. Hence Aηk ε T(∅) by the definition of T. This is the induction basis.

Let i ≥ 1. Suppose that Aηk-i+1 ...η k ε $T^i(\emptyset)$ for any atomic cluster A in Mk-i. Let

Mk-i-1 = C1,...,Cj,...,Cm
Mk-1 = (C1,...,Cj-1)ηk-i,
(B1,...,Bn)θ,
(Cj+1,...,Cm)ηk-i

for some definite sequent B0 :- B1,...,Bn in £ and for some matching substitution θ, where Cjηk-i = B0θ holds. Let A be any atomic cluster in Mk-i-1. If A ≠ Cj, then Aηk-i is in Mk-i. So by induction hypothesis we have Aηk-i ηk-i+1 ... η k ε $T^i(\emptyset) \subset T^{i+1}(\emptyset)$ since T is monotonic. If A = Cj, then by induction hypothesis we have Bqθηk-i+1 ... η k ε $T^i(\emptyset)$ for all 1 ≤ q ≤ n. So that Aηk-iη k-i+1 ... η k = B0θ ηk-i η k-i+1 ... η k ε $T^{i+1}(\emptyset)$ by the definition of T.

Theorem 3 Let R be a recursive equational program and £ be the translated Horn program from R. Then F(t1,...,tn) $\overset{*}{\underset{(p)}{\Rightarrow}}$ tn+1 for all atoms F_p(t1,...,tn+1) in lfp(£).

proof. We show by induction on i ≥ 1 that F(t1,...,tn) $\overset{*}{\underset{(p)}{\Rightarrow}}$ tn+1 in R for all F_p(t1,...,tn+1) ε $T^i(\emptyset)$.

If i = 1, then F_p(t1,...,tn+1) = F_p(q1,...,qn+1)θ for some definite sequent F_p(q1, ...,qn+1) :- e and substitution θ. This sequent corresponds to the rewriting rule F(q1,...,qn) --> qn+1 by the transformation algorithm. So we have

F(t1,...,tn) = F(q1θ,...,qnθ)
$\overset{}{\underset{(p)}{\Rightarrow}}$ tn+1

by applying the rule F(q1,...,qn) --> qn+1 with the substitution θ.

Let i ≥ 1. Suppose that F(t1,...,tn) $\overset{*}{\underset{(p)}{\Rightarrow}}$ tn+1 for all atoms F_p(t1,...,tn,tn+1) ε $T^i(\emptyset)$. Let F_p(t1,...,tn,tn+1) be any atom in $T^{i+1}(\emptyset)$. By definition of T there is a definite sequent B0 :- B1,...,Bm in £ such that F_p(t1,...,tn,tn+1) = B0θ, $B_j\theta \in T^i(\emptyset)$, 1 ≤ j ≤ m, for some substitution θ. Let F(q1,...,qn) --> E be the rewriting rule from which the definite sequent B0 :- B1,...,Bm is obtained. By applying rule F(q1,...,qn) --> E with the substitution θ to the Σ-term F(t1,...,tn), we can derive a Σ-term Eθ as a result. To prove Theorem it suffices to show that for any subterm G(E1,...,Ek) of Eθ, where G is the defined function symbol, if G_p(p1,...,pk,pk+1) is the corresponding atom which belongs to (B1,...,Bm)θ, then G(E1,...,Ek) $\overset{*}{\underset{(p)}{\Rightarrow}}$ pk+1 follows from Ej $\overset{*}{\underset{(p)}{\Rightarrow}}$ pj, 1 ≤ j ≤ k. This can be easily verified by using induction hypothesis. So we omit the details.

We obtain the following theorem for recursive equational programs from Corollary 4, Theorem 2, and Theorem 3.

Theorem 4 Let R be a recursive equational program and £ the translated Horn program from R. For any Σ-term F(t1,...tn) and for any constructor term t the next two conditions are equivalent.

(1) There is a successful computation of the input term F(t1,...tn) with the result t in the equational program R :
F(t1,...,tn) $\overset{*}{\underset{(p)}{\Rightarrow}}$ t.
(2) There is a successful computation of the

goal $F(t1,\ldots,tn,t)$ in the logic program
$\mathcal{L}$: $F_p(t1,\ldots,tn,t) \overset{*}{=>} e$.

6 CONCLUDING REMARKS

To introduce notions data abstraction and efficient computation strategies into logic programs we have proposed a transformation algorithm from equational programs into logic programs. We believe that the results of this paper clarify relationship between equational programs and logic programs, in particular recursive equational programs and Horn programs.

We could not show that any equational program was transformed into an equivalent logic program. However, we expect that every equational program is simulated by some logic program with the equivalent computational power. We hope that our proposal has established a new path to the further study of this field.

REFERENCES

Apt,K.R., and Van Emden,M.H.: Contributions to the theory of logic programming, J.ACM, 29, pp. 841-862 (1982).

Arnold,A., and Nivat,M.: Formal computations of non deterministic recursive program schemes, Mathematical Systems Theory, 13, pp.219-236 (1980).

Clark,K.L.: Predicate logic as a computational formalism, Research Report, Department of Computing, Imperial College, London (1979).

Clocksin,W.F., and Mellish,C.S.: Programming in Prolog, Springer-Verlag (1981).
Downey,P.J., and Sethi,R.: Correct computation rules for recursive language, SIAM J. Comput., 5, pp.378-401 (1976).

Van Emden,M.H., and Kowalski,R.A.: The semantics of predicate logics as a programming language. J.ACM, 23, pp.733-742 (1976).

Goguen,J.A., Thatcher,J.W., and Wagner, E.G.: An initial algebra approach to the specification, correctness, and implementation of abstract data types, in Current Trends in Programming Methodology, 4, ed. Yeh,R., Prentice-Hall, pp.80-149 (1978)

Guttag,J., Horowitz,E., and Musser,D.R.: The design of data type specifications, in Current Trends in Programming Methodology, 4, ed. Yeh,R., Prentice-Hall,pp.60-79 (1978).

Hoffmann,C.M., and O'Donnell,M.J.: Programming with equations, ACM Trans. on Prog. Lang. and Sys., 4, pp.83-112 (1982).

Hogger,C.J.: Derivation of logic programs, J.ACM, 28, pp. 372-392 (1981).

Huet,G.: Confluent reductions : Abstract properties and applications to term rewriting systems, J.ACM, 27, pp. 797-821 (1980).

Huet,G., and Hullot,J.M.: Proofs by induction in equational theories with constructors, JCSS, 25, pp.239-266 (1982).

Huet,G., and Oppen,D.C.: Equations and rewrite rules. in Formal Language Theory, Academic Press, pp.349-405 (1980).

Kowalski,R.A.: Logic for problem solving, North-Holland (1979).

O'Donnell,M.J.: Computing in systems described by equations, Lec. Notes in Comput. Sci., NO 58 (1977).

Robinson,J.A.: A machine-oriented logic based on the resolution principle, J.ACM, 12, pp.23-41 (1965).

Rosen,B.K. : Tree-manipulating systems and Church-Rosser theorems, J.ACM, 20, pp.160-187 (1973).