ICOT Technical Report: TR-018 JUL. 1983

A Transformation System for Logic Programs Which Preserves Equivalence

by

Hisao Tamaki

(Ibaraki University, Ibaraki, Japan)

Taisuke Sato

(Electrotechnical Laboratory, Ibaraki, Japan)

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

03-456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Transformation System for Logic Programs which Preserves Equivalence

Hisao TAMAKI

(Ibaraki University, Ibaraki, Japan)

Taisuke SATO

(Electrotechnical Laboratory, Ibaraki, Japan)

ABSTRACT

A program transformation system for logic programs based on the fold/unfold technique is defined and proved to preserve the equivalence of programs.

* This is a completely rewritten version of the earlier draft dated Nov. 1982.

1. Introduction

One claim in favor of logic programming languages like Prolog[Cl] is that they support a declarative programming style; the programmer only specifies the relation between input and output, leaving the processor to deal with how output is computed from input. But this advantage is not sufficiently exploited in current programming practice in Prolog. Anyone who writes a non-trivial program in Prolog is forced to consider control aspects in detail, if he wishes to achieve reasonable efficiency.

To make the declarative programming style a real advantage of logic programming, we need a programming environment where lucid, specification-like programs are automatically or semi-automatically transformed into less lucid, efficiency-oriented programs. Fortunately, the semantics of (pure) Prolog is simple and clear, making programs easy to manipulate mechanically.

This paper aims to give a system of basic rules for program transformation and prove that they preserve the equivalence of programs. Our system is similar to that of Burstall and Darlington[Bu] which is for programs written in a functional language. Their system consists of a small set of basic transformation rules including the well-known folding and unfolding rules. They present many examples to show the practical power of their system. We are not concerned here to repeat the same set of examples to prove that our system is equally practical. (For a practical comparison see [Sa].) Rather, our emphasis is on the total correctness of transformation, which is not guaranteed in their system. Though all the examples they give are totally correct, their formulation of the system does not allow a general correctness proof. To make a general proof possible, we need a more elaborate formulation and a formal semantics of the language. What we present here is such a formulation, in a language with an especially simple formal semantics, namely pure Prolog. We suggest later that a similar reformulation is possible in their own language.

2. Pure Prolog

The logic programming language we choose is pure Prolog, which is a Prolog without 'cut', 'not', 'assert' or any other extra-logical features. It is essentially the language in Kowalski's proposal [Kow] of logic programming.

To describe the syntax of pure Prolog, we first assume three sets of symbols: the set of *predicate symbols*, the set of *function symbols*, and the set of *variables*. As meta-symbols to represent the symbols, we use letters P and Q for predicate symbols, f and g for function symbols and x, y and z for variables, all possibly with subscripts. Each predicate or function symbol has a fixed arity.

A *term* is either a variable or an expression of the form $f(t_1,..,t_n)$ where f is an n-ary function symbol and $t_1,..,t_n$ are terms. A term consisting of a nullary function symbol is called a *constant*.

An *atomic formula* is an expression of the form $P(t_1,..,t_n)$ where P is an n-ary predicate symbol and $t_1,..,t_n$ are terms. An atomic formula is also called a *goal*. A *goal set* is a finite multi-set of goals, i.e. we allow multiple occurrences of a syntactically identical goal in a goal set.

A *definite clause* is a pair consisting of a goal, called a *head*, and a goal set, called a *body*. In this paper, we refer to a definite clause simply as a *clause*. A clause with an empty body is called a *unit clause*. A *program* is a set of clauses.

We use letters G, H and K for goals, Γ and Δ for goal sets, A, B and C for clauses and S for programs. A clause whose head is G and body is $\Gamma = \{G_1,..,G_n\}$ is expressed as $G \leftarrow G_1,..,G_n$ or $G \leftarrow \Gamma$.

A *substitution* is a finite set of variable-term pairs such that no two pairs share a common variable part. The result of applying a substitution to a term t, denoted by $t\theta$, is t in which for every pair (x_i, t_i) in θ each occurrence of x_i is replaced by t_i. A term t is an *instance* of a term t' if

there is some substitution θ such that $t = t'\theta$. *Composition* of two substitutions θ and σ, denoted by $\theta\sigma$, is a substitution such that $(t\theta)\sigma = t(\theta\sigma)$ for every term t. A substitution θ is called a *unifier* of two terms t_1 and t_2 if $t_1\theta = t_2\theta$. It is a *most general unifier (mgu)* if any unifier θ' of t_1 and t_2 can be represented as $\theta\sigma$ by some substitution σ. Substitution, unifiers etc. for goals and clauses are defined similarly.

Let C be a clause $G \leftarrow \Gamma$, H a goal in Γ, and C' a clause $K \leftarrow \Delta$ such that H and K have an mgu θ. *Resolution*[Ro] in our context is an operation which deduces a new clause $\Gamma\theta \leftarrow (\Gamma - \{H\})\theta \cup \Delta\theta$ called a *resolvent* from C and C'. We say the goal H is *resolved upon* in the resolution.

To define the standard semantics of pure Prolog, we regard a program as a ground atomic theory, where a clause is considered as an inference rule to deduce a ground (variable-free) goal from a ground goal set.

Definition 2.1 Derivation

A finite set T of finite trees whose nodes are labeled with ground goals is called a derivation in S if for every nonterminal node, labeled with G, and its son nodes, labeled with $G_1,..,G_n$, there is a clause C in S such that the clause $G \leftarrow G_1,..,G_n$ is an instance $C\theta$ of C.

We say that the clause C is *used* in the derivation *at* the node labeled with G, *with* substitution θ. We define some notations:

$\Gamma(T)$: the multi-set of labels of all roots of T

$\Delta(T)$: the multi-set of labels of all terminal nodes of T

$\Sigma(T)$: the set of all subtrees obtained from T by removing all roots.

If Λ is a subset of $\Delta(T)$ such that every goal in $\Delta(T) - \Delta$ is an instance of a head of some unit clause in S, then we say T is a derivation *of* $\Gamma(T)$ *from* Λ in S.

Definition 2.2 Concatenation of derivations

Let T_1, T_2 be derivations in S such that $\Gamma(T_1)$ and $\Gamma(T_2)$ are identical as sets (not necessarily as multi-sets). Then the *concatenation* of T_1 and T_2 is a derivation of $\Gamma(T_1)$ constructed by replacing every terminal node of T_1 by a tree in T_2 whose root is labeled with the same goal.

Definition 2.3 Proof

A derivation T in S is called a *proof* of $\Gamma(T)$ in S if it is a derivation from an empty goal set. In other words, if every goal in $\Delta(T)$ is an instance of a head of some unit clause in S. If T consists of just one tree, we call it a *single proof*.

If a ground goal set Γ has a proof in S then we say it is *provable* in S.
Now we define the semantics of pure Prolog.

Definition 2.4 Meaning of a program

The meaning $M(S)$ of a program S is the set of ground goals defined by

$$M(S) = \{G \mid G \text{ is a ground goal such that } \{G\} \text{ is provable in } S\}.$$

$M(S)$ so defined coincedes with the minimum Herbrand model of S[Em].
Two programs S_1 and S_2 are said to be *equivalent* if $M(S_1) = M(S_2)$.

Note that if a goal set is provable then it has a proof T which satisfies the following condition.

(*) If a goal G labels more than one node of T, then the corresponding sub-proofs of G are all identical.

So in the sequel, we consider only proofs which satisfy (*). This restriction is necessary to validate the arguments about the *goal merging* transformation rule introduced later.

3. Example

The following is a program for Fibonacci numbers.

C1. F(0,1) ←

C2. F(1,1) ←

C3. F(x+2,n) ← F(x+1,n1),F(x,n2),Plus(n1,n2,n)

Here we abbreviate successor function aplications s(0), s(x), s(x(x)) as 1, x+1, x+2. We assume the program includes a subprogram S_{prim} for primitive predicates such as Plus for natural number addition, = for equality and ≠ for inequality. This program requires an exponential number of additions when executed on ordinary top-down interpreters. We transform it into a linear order program. (Cf. the example in [Bu])

First we *define* a new predicate G by the following clause.

C4. G(x,n1,n) ← F(x+1,n1),F(x,n2),Plus(n1,n2,n)

Then *unfold* C4 at the first goal F(x+1,n1). That is, replace C4 by the set of all possible resolvents resulting from resolving upon the goal.

C5. G(0,1,n) ← F(0,n2),Plus(1,n2,n)

C6. G(x+1,n1,n) ← F(x+1,n3),F(x,n4),Plus(n3,n4,n1),
F(x+1,n2),Plus(n1,n2,n)

Unfold C5 at the first goal and then at the second goal.

C7. G(0,1,2) ←

Case split C6 by n2=n3 and n2≠n3.

C8. G(x+1,n1,n) ← F(x+1,n2),F(x,n4),Plus(n2,n4,n1),
F(x+1,n2),Plus(n1,n2,n)

C9. G(x+1,n1,n) ← n2≠n3,F(x+1,n3),f(x,n4),Plus(n3,n4,n1),
F(x+1,n2),Plus(n1,n2,n)

Delete C9 knowing that it always fails.

Merge identical goals in C8.

C10. G(x+1,n1,n) ← F(x+1,n2),F(x,n4),Plus(n2,n4,n1),Plus(n1,n2,n)

Fold C10 by C4. That is, we recognize the first three goals of C10 as an instance of the body of C4, and replace it with the corresponding instance of its head.

C11. G(x+1,n1,n) ← G(x,n2,n1),Plus(n1,n2,n)

Fold C3 by C4.

C12. F(x+2,n) ← G(x,n1,n)

The resulting program is

C1. F(0,1) ←

C2. F(1,1) ←

C12. F(x+2,n) ← G(x,n1,n)

C7. G(0,1,2) ←

C11. G(x+1,n1,n) ← G(x,n2,n1),Plus(n1,n2,n)

together with S_{prim}.

The theorem proved in the next section ensures the equivalence of the final program with the original one plus C4.

4. Definition and Correctness Proof of the System

Before describing each transformation rule, we sketch the transformation process as a whole.

We start from a basic program S_0 and a set D of clauses which define new predicates in terms of predicates in S_0. We let $S_1 = S_0 \cup D$ and transform S_1 into S_2, S_2 into S_3, and so on, using the transformation rules described below. The correctness proof of our system consists in showing that every S_i $(i = 1,2,..)$ is equivalent to S_1. The discrimination of definitions from other clauses plays a central role in our formulation and proof.

In practice, we wish to expand the set D during the process to introduce auxiliary predicates, as we did in the example. This causes no problems because we can treat every definition introduced during the process as if it were in D from the first.

Definition 4.1 Definition set

Let S be a program and P an n-ary predicate symbol not in S. We say a clause $G \leftarrow \Gamma$ is a *definition of P over S* if G is of the form

$$P(x_1,..,x_n)$$

where $x_1,..,x_n$ are distinct variables, and every predicate symbol of Γ occurs in the head of some clause in S. Those variables in Γ other than $x_1,..,x_n$ are called the *internal variables* of the definition. A finite set D of clauses is a *definition set over S* if each clause in D is a definition defining a distinct predicate over S.

Definition 4.2 D-expansion

Let $G = P(t_1,..,t_n)$ be a goal whose predicate symbol P is defined in a definition set D over S by a definition $P(x_1,..,x_n) \leftarrow \Gamma$. A *D-expansion* of G is an instance $\Gamma\theta$ of Γ where θ is a substitution which maps each x_i to t_i and internal variables of the defintion to

arbitrary terms. It is a *most general D-expansion* if θ maps internal variables to distinct new variables. We refer to those variables as *internal variables of the most general D-expansion.*

Note that any *D*-expansion can be obtained by substituting some terms for internal variables in a most general *D*-expansion.

For example, let *D* be {C4} with C4 as in the example in the previous section. The goal set {F(2,2),F(1,1),Plus(2,1,n)} is a *D*-expansion of the goal G(1,2,n). The goal set {F(2,2),F(1,n2),Plus(2,n2,n)} is the most general expansion of the same goal where n2 is the internal variable.

Each of the following transformations carries a program *S* into a new program *S'*. We assume that there is a fixed set of predicate symbols called *primitive predicate symbols* and those parts of programs which implement primitive predicates are not subjected to transformation.

Transformation 1. Unfolding

Let *C* be an arbitrary clause in *S* with an arbitrary goal *G* in its body. Let $C_1', .., C_k'$ be all the clauses in *S* whose heads are unifiable with *G*. Then

$$S' = (S - \{C\}) \cup \{C_1, .., C_k\}$$

where each C_i is a resolvent of *C* with C_i'.

Note. Variables in C_i' should be renamed so that they are not shared by *C*.

Transformation 2. Folding

Let *C* be an arbitrary clause in *S* of the form $G \leftarrow \Gamma$, and *H* be a goal with a predicate symbol defined in *D* such that Γ includes a most general *D*-expansion Δ of *H*. We require the following two conditions to be satisfied.

(1) Internal variables of Δ do not occur in G or $\Gamma - \Delta$.

(2) C is not in D, or

$\Gamma - \Delta$ is not empty.

Then

$$S' = (S - \{C\}) \cup \{C'\}$$

where C' is a clause with a head G and a body $\Gamma \cup \{H\} - \Delta$.

Transformation 3. Deletion

$$S' = S - \{C\}$$

where C is a clause in S such that no proof in S uses C.

Transformation 4. Goal merging

$$S' = (S - \{C\}) \cup \{C'\}$$

where C is a clause in S whose body contains more than one copy of a syntactically identical goal and C' is the result of merging those copies into one.

Transformation 5. Case splitting

We say two predicate symbols P and Q are *complementary* in S if

(1) P and Q are of the same arity n, and

(2) for any ground terms $t_1,..t_n$, $P(t_1,..,t_n)$ is provable in S if and only if $Q(t_1,..,t_n)$ is not provable in S.

Let C be an arbitrary clause of the form $G \leftarrow \Gamma$ in S, C_1 and C_2 be clauses

$$G \leftarrow \Gamma \cup \{P(t_1,..,t_n)\}$$

$$G \leftarrow \Gamma \cup \{Q(t_1,..,t_n)\}$$

where P and Q are complementary *primitive* predicate symbols in S and $t_1,..,t_n$ arbitrary terms. Then

$$S' = (S - \{C\}) \cup \{C_1, C_2\}.$$

The following lemma corresponds to the partial correctness of Burstall and Darlington's system informally stated in [Bu].

Lemma 4.1

Let S_0 be a program, D a definition set over S_0, and S a program equivalent to $S_0 \cup D$. If S' is a program obtained by applying one of the five transformation rules to S, then $M(S') \subset M(S)$.

Proof

Let T' be a proof in S'. We must show that there is a proof T in S such that $\Gamma(T) = \Gamma(T')$. The cases of deletion, goal merging and case splitting are trivial. For the case of unfolding, it suffices to note that any single-step derivation which uses a resolvent can be converted into a two-step derivation using resolved clauses.

In the case of folding, we argue by induction on the structure of T'. The only non-trivial case is when T' is a single proof and the clause used at the root of T' is the clause C' introduced by the folding transformation. Let C, Γ, Δ, G and H be as in the definition of the folding rule. Remember $C = G \leftarrow \Gamma$ is the folded clause, $\Delta \subset \Gamma$ the goal set folded into the goal H. Let θ be a most general substitution such that $\Gamma(T') = \{G\theta\}$ and $\Gamma(\Sigma(T')) = (\Gamma - \Delta)\theta \cup \{H\theta\}$. Note that θ does not affect the internal variables of the D-expansion Δ of H. Let T_1' be the proof of $\{H\theta\}$ included in $\Sigma(T')$. By the induction hypothesis there is a proof T_1 of $\{H\theta\}$ in S. Because S is equivalent to $S_0 \cup D$, there is a proof T_1'' of $\{H\theta\}$ in $S_0 \cup D$. The clause used at the root of T_1'' must be the definition involved in the folding. So there is a substitution σ which instantiates the internal variables of the D-expansion $\Delta\theta$ of $H\theta$ such that $\Sigma(T_1'')$ is a proof of $\Delta\theta\sigma$. Again by equivalence of S and $S_0 \cup D$, there is a proof T_2 of $\Delta\theta\sigma$ in S.

Now applying the induction hypothesis to $\Sigma(T') - T_1'$, we get a proof T_3 of $(\Gamma - \Delta)\theta$ in S. Concatenating a single step derivation using C with the proof

T_2 U T_3 of $\Gamma\theta\sigma$, we get a proof T in S of $\{G\theta\sigma\}$, which is actually $\{G\theta\}$. See Fig. 1. (end of proof)

To prove the other direction $M(S') \supset M(S)$, we need a stronger condition for S than merely being equivalent to S_0 U D. To present the idea clearly, we first ignore the last transformation rule, namely, case splitting.

Definition 4.3 D-simulation

Let D be a definition set over a program S_0, T_0 a proof in S_0, G a goal and T a proof of $\{G\}$ in some program S. T is called a *D-simulation* of T_0 if the following two conditions hold.

(1) $\Gamma(T_0) = \{G\}$ or

$\Gamma(T_0)$ is a D-expansion of G.

(2) Let $\Sigma(T) = T_1$ U.. U T_n where each T_i is a distinct single proof. Then there is a derivation T_0' and proofs $T_1',..,T_n'$ in S_0 such that

(a) T_0 is a concatenation of T_0' with T_1' U .. U T_n', and

(b) each T_i is a D-simulation of T_i' $(i=1,..,n)$.

When $n = 0$, (a) and (b) unconditionally hold for $T_0' = T_0$, which makes the base case of the inductive definition.

Fig. 2 shows an example of D-simulation.

Definition 4.4 D-extension

Let S_0, S and D be as in definition 4.3. S is called a *D-extension* of S_0 if the following two conditions hold.

(1) For any single proof T_0 in S_0, there is a proof T of $\Gamma(T_0)$ in S which is a D-simulation of T_0.

(2) For any proof T_0 in S_0 such that $\Gamma(T_0)$ is a D-expansion of some goal G, there is a proof T of $\{G\}$ in S which is a D-simulation of T_0.

Lemma 4.2

If S is a D-extension of S_0 then $M(S) \supset M(S_0 \cup D)$.

Proof

Let T_0 be a proof in $S_0 \cup D$ of $\{G\}$. If the predicate of G is defined in D, then $\Gamma(\Sigma(T))$ must be a D-expansion of G. Moreover $\Sigma(T)$ is a proof in S_0. So there is a D-simulation T of $\Sigma(T)$ in S, which is a proof of $\{G\}$. The other case is trivial by definition. (end of proof)

Now our problem reduces to the following lemma.

Lemma 4.3'

Let S_0 be a program, D a definition set over S_0, and S a D-extension of S_0. If S' is a program obtained from S, applying one of the first four transformation rules, S' is also a D-extension of S_0.

Proof

If the rule applied is deletion or goal merging then the proof is obvious. So we deal with only cases of unfolding and folding.

Unfolding.

Let C, $C_1', \ldots, C_k'$, $C_1, \ldots, C_k$ and G be as in the definition of the unfolding rule. We inductively define a mapping u from the set of proofs in S to the set of proofs in S'.

(case 1) T is an empty set.

$u(T)$ is also an empty set.

(case 2) T consists of just one tree.

Let $\{H\}$ be $\Gamma(T)$ and Γ be $\Gamma(\Sigma(T))$.

(case 2.1) $H \leftarrow \Gamma$ is not an instance of C, the unfolded clause.

$u(T)$ is a proof determined by

$\Gamma(u(T)) = \{H\}$, and

$\Sigma(u(T)) = u(\Sigma(T))$.

(case 2.2) $H \leftarrow \Gamma$ is an instance of C.

There must be some G' in Γ which is an instance of the goal G, the unfolded goal. Let T' be a single proof in $\Sigma(T)$ whose root is labeled with G'.

$u(T)$ is a proof determined by

$\Gamma(u(T)) = \{H\}$, and

$\Sigma(u(T)) = u(\Sigma(T) - T') \cup u(\Sigma(T'))$.

(case 3) T consists of more than one tree.

Let T be $T_1 \cup .. \cup T_n$ where each T_i is a single proof.

$u(T) = u(T_1) \cup .. \cup u(T_n)$.

To show that $u(T)$ is realy a proof in S', it suffices to note that in case 2.2, the clause with head H and body $(\Gamma(\Sigma(T)) - \{G'\}) \cup \Gamma(\Sigma(T'))$ is an instance of some C_i which is introduced by the unfolding transformation.

It is obvious that if a proof T is a D-simulation of a proof T_0, then so is $u(T)$. This means that if S is a D-extension of S_0 then so is S'.

Folding.

We define a well-founded ordering $<$ on the set of all pairs of a proof in S_0 and a proof in S. We let $(T_0,T) < (T_0',T')$ if and only if

size of T_0 < size of T_0', or

size of T_0 = size of T_0' and size of T < size of T'.

We show by induction based on this ordering that for every proof T_0 in S_0 and a D-simulation T of T_0 in S, there is a D-simulation T' of T_0 in S' such that $\Gamma(T') = \Gamma(T)$. Let $\Sigma(T)$ be $T_1 \cup .. \cup T_n$ where each T_i is a single proof. By the definition of D-simulation there is a derivation T_0' and proofs $T_1',..,T_n'$ in S_0 such that

(a) T_0 is a concatenation of T_0' with $T_1' \cup .. \cup T_2'$, and

(b) each T_i is a D-simulation of T_i'.

Let C_0 be the clause in S used at the root of T with substitution θ.

(case 1) C_0 is C, the folded clause.

There is a subset T_Δ of $\Gamma(T)$ such that $\Gamma(T_\Delta)$ is an instance $\Delta\theta$ of Λ, the folded goal set. We assume $T_\Delta = T_1 \cup .. \cup T_m$ for some m without loss of generality. Because none of the predicates of Δ are defined in D, $\Gamma(T_1' \cup .. \cup T_m')$ is also $\Delta\theta$, which is a D-expansion of the goal $H\theta$. Furthermore condition (2) of the folding rule ensures that the size of $T_1' \cup .. \cup T_m'$ is strictly less than that of T_0. By assumption there is a D-simulation T'' of $T_1' \cup .. \cup T_m'$ in S such that $\Gamma(T'') = \{H\theta\}$. So by the induction hypothesis, there is a corresponding D-simulation T''' in S'. Also by the induction hypothesis there is a D-simulation T_i'' of each T_i' $(i=m+1,..,n)$ such that $\Gamma(T_i'') = \Gamma(T_i)$. Concatenating a single-step derivation which uses C', the clause introduced by folding, with the proof $T''' \cup T_{m+1}'' \cup .. \cup T_n''$, we obtain a D-simulation T' of T_0 in S'. See Fig. 3.

(case 2) C_0 is not C.

We apply the induction hypothesis to each pair (T_i', T_i) to get a D-simulation T_i'' of T_i' corresponding to T_i. This is possible because the size of T_i' is less than or equal to the size of T_0 and the size of T_i is strictly less than that of T. (end of proof)

The example in fig. 4 illustrates why we had to take the size of D-simulation into account in the induction.

To adapt the above proof to include the case splitting rule, we must extend the concept of D-simulation so that any proof obtained by putting a proof for a primitive predicate to a D-simulation is also a D-simulation. It is clear that for any D-simulation which uses a clause C at the root, there is a corresponding D-simulaiton which uses either of the case-split clauses of C.

The above proofs for folding and unfolding rules are still valid for the extended concept of D-simulation and D-extention. Thus, we have the complete version of the lemma, hence our main theorem.

Lemma 4.3

Let S_0 be a program, D a definition set over S_0, and S a D-extension of S_0 (in the extended sense). If S' is a program obtained from S applying one of the five transformation rules, S' is also a D-extension of S_0.

Theorem

Let S_0 be a program, D a definition set over S_0, and $S_1 = S_0 \cup D$. Let S_i $(i=2,..,n)$ be the result of applying one of the transformation rules to S_{i-1}. Then each S_i is equivalent to S_1.

Proof

We prove by induction on i that S_i is a D-extension of S_0 and $M(S_i) \subset M(S_1)$, which together imply $M(S_i) = M(S_1)$ by lemma 4.2. The base case $i=1$ is trivial and the induction step is nothing but lemmas 4.1 and 4.3.

(end of proof)

5. Relation to Other Work

We should make clear what made possible the total correctness result absent in Burstall and Darlington's original system. First, their transformation rules are *rules of introduction* while ours are *rules of replacement*. Remember that a program consists of a set of equations in their system and of clauses in ours. Their rules deduce from the old set new equations which can be added to the set, but tell nothing about which equations can be discarded. As long as no equation is discarded, and the rules are sound as inference rules, it is trivial that the new set is equivalent to the original set. But then the result is a redundant program from which we have to extract the non-redundant part without weakening the program. So we must introduce *rules of removal* or refine the rules of introduction into rules of replacement.

Clearly, some of their rules such as *instantiation* or *unfolding* cannot be used as rules of replacement by themselves. But their examples show that instantiation and unfolding are always applied together to perform exhaustive expansion, which corresponds to our unfolding rule.

If we are to formalize transformation rules as rules of replacement, then the discrimination of definitions from other clauses (or equations) become necessary because otherwise definitions are lost before they can be used in folding transformations.

The above observations suggest that reformulation similar to that presented here is possible in their own language, thus ensuring total correctness. The case of proving correctness depends on the semantics we choose. For call-by-value semantics, our result can immediately be applied. To cope with more general semantics, some restrictions may be required on programs or on transformation rules.

Scherlis[Sc1][Sc2] develops a similar transformation system in a functional language which allows definitions of *expression procedures*, and proves its total correctness. Using expression procedures has a similar effect to distinguishing

definition (for folding) from other clauses in a program.

Kott[Ko] has also studied the correctness of unfold/fold transformation and states that a general correctness proof is impossible. This does not contradict our result because his formulation based on recursive program scheme is different from ours.

Hogger's paper on derivation of logic programs[Ho] refers to program transformation on the source (Prolog) level, which uses the unfold/fold technique. His correctness argument relies on the fact that if a program S' is deduced from S then what S' computes is correct with respect to S. Accordingly his method in general ensures only partial correctness.

5. Conclusion

We have defined a system of transformation rules for logic programs and proved that it preserves the equivalence of programs. This provides a firm foundation for automatic or semi-automatic source-level program optimization.

Our notion of total correctness does not refer directly to termination. This attitude is completely sound as long as we use a complete Prolog implementation. Further investigation is required to satisfy ourselves regarding the usual imcomplete implementations. (An alternative is to search for efficient complete implementations.)

It remains to extend the system to cope with reasonable extensions of pure Prolog such as negation as fialure and infinite processes.

Acknowledgement

This work is based on the activities of the working groups of the Fifth Generation Computer Project.

References

[Bu] R.M. Burstall, J. Darlington, A Transformation System for Developing Recursive Programs, Journal of the ACM, Jan. 1977

[Cl] W.F. Clocksin, C.S. Mellish, Programming in Prolog, Springer-Verlag, 1981

[Em] M.H. van Emden, R.A. Kowalski, The Semantics of Predicate Logic as a Programming Language, Journal of the ACM, Oct. 1976

[Ho] C.J. Hogger, Derivation of Logic Programs, Journal of the ACM, Apr. 1981

[Ko] L. Kott, Unfold/Fold Program Transformations, INRIA R.R. 155, Aug. 1982

[Kow] R.A. Kowalski, Predicate Logic as a Programming Language, Information Processing 74, North-Holland, Amsterdam, 1974

[Ro] J.A. Robinson, A Machine Oriented Logic based on the Resolution Principle, Journal of the ACM, Jan. 1965

[Sa] T. Sato, H. Tamaki, Program Transformation in Prolog (in Japanese) Logic Programming Conference 83, Tokyo, Mar. 1983

[Sc1] W.L. Scherlis, Program Improvement by Internal Specialization, 8th POPL symposium, Williamsbug, Virginia, Jan. 1981

[Sc2] W.L. Scherlis, Expression Procedures and Program Derivation, STAN-CS-80-818, Stanford, 1980

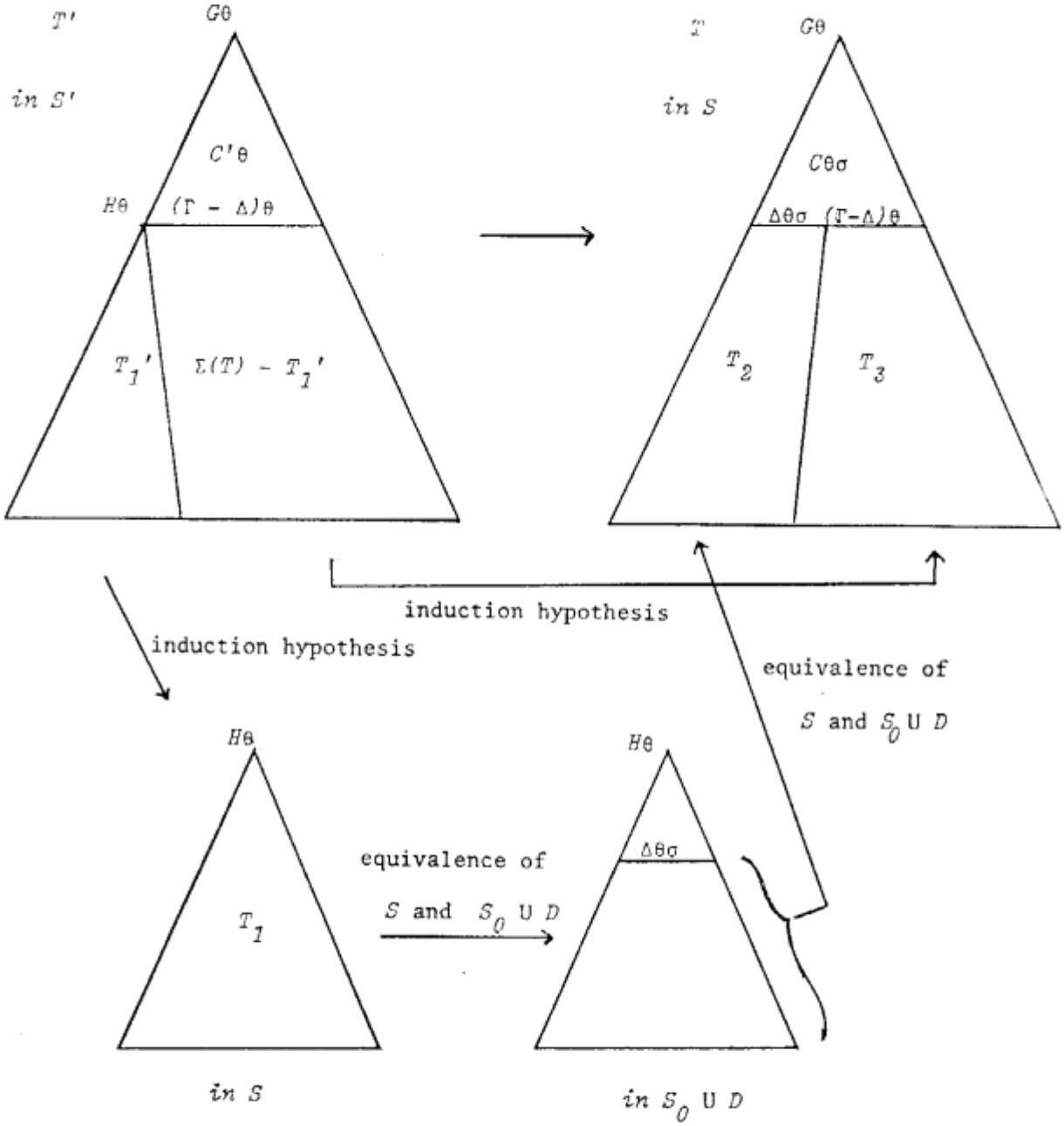
T'
Gθ
in S'
C'θ
Hθ
(Γ - Δ)θ
T1'
Σ(T) - T1'
T
Gθ
in S
Cθσ
Δθσ
(Γ-Δ)θ
T2
T3
induction hypothesis
induction hypothesis
equivalence of
S and S0 U D
Hθ
T1
in S
equivalence of
S and S0 U D
Hθ
Δθσ
in S0 U D

Fig. 1

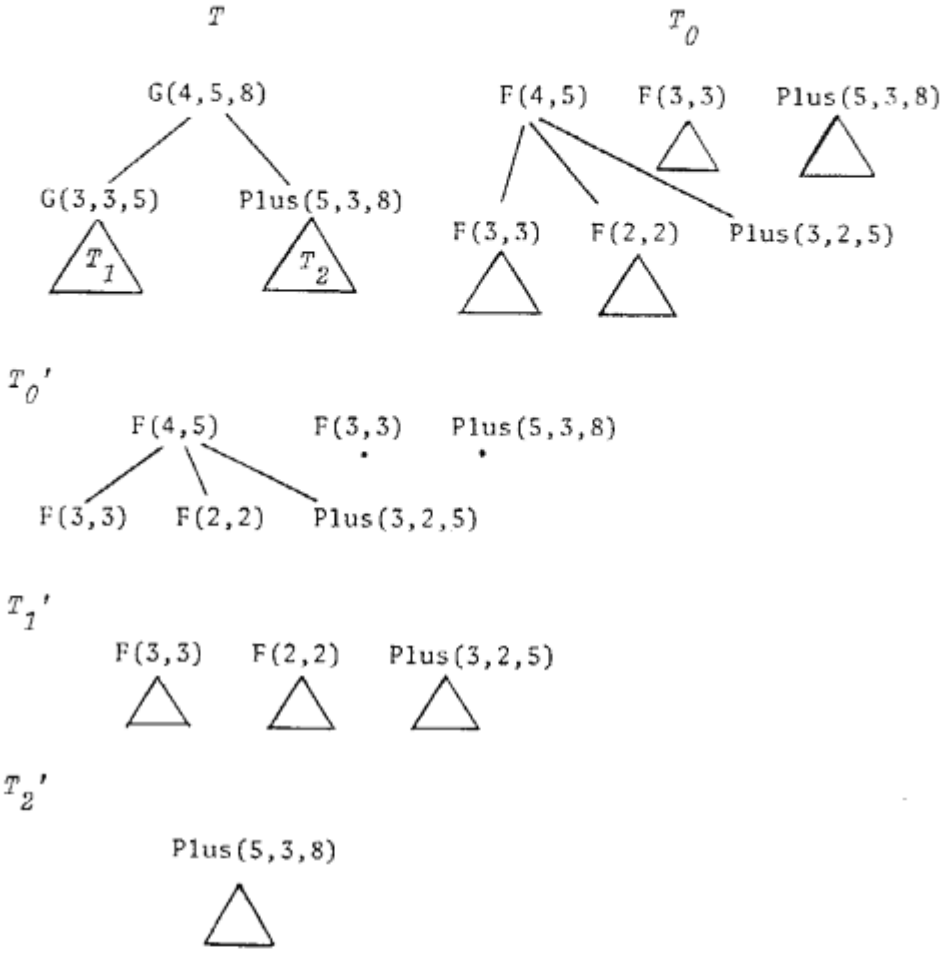

Fig. 2 T is a D-simulation of T_0 provided T_1' and T_2' are D-simulations of T_1 and T_2 respectively.

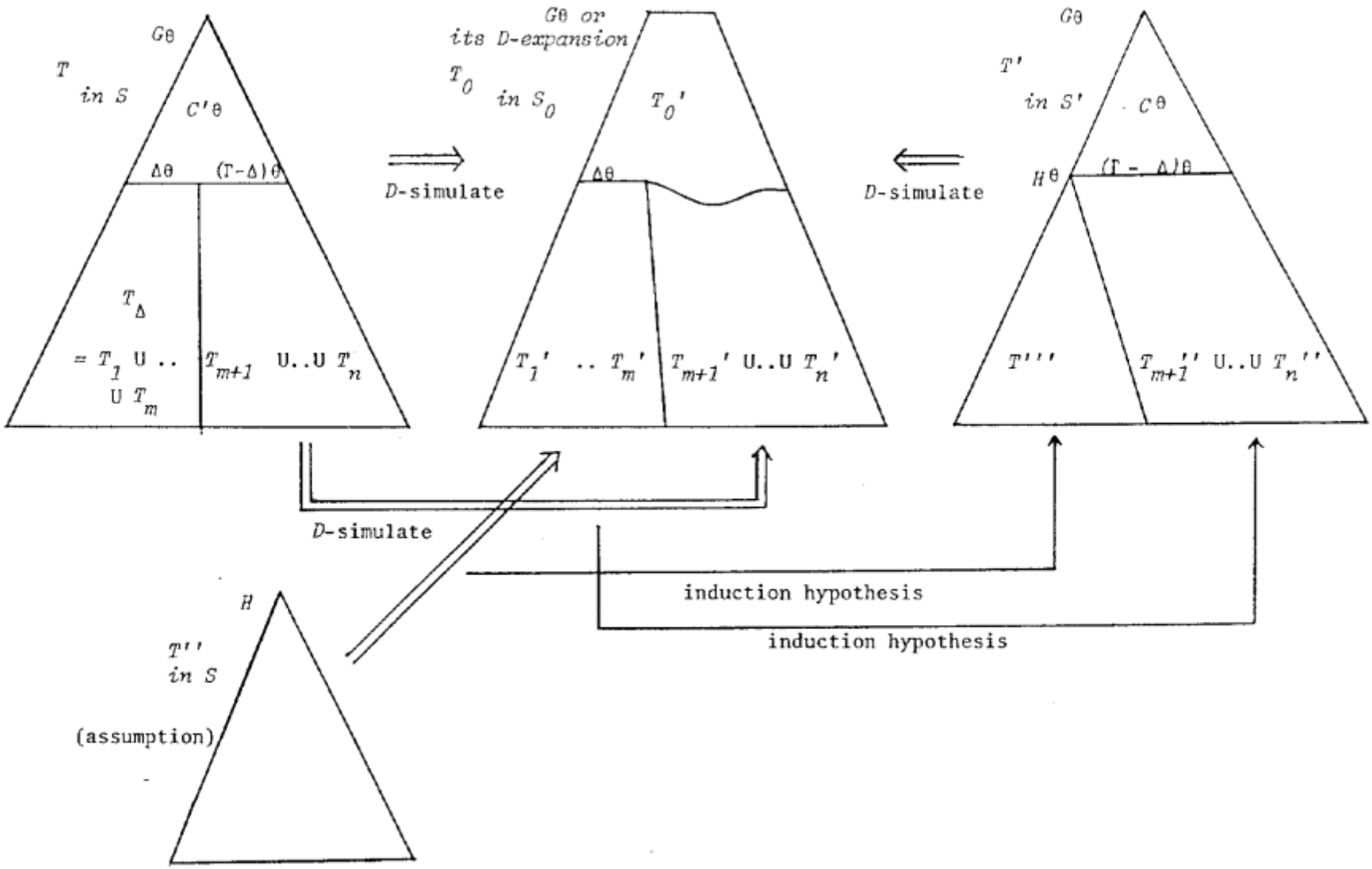
Gθ
T
in S
C'θ
Δθ
(T-Δ)θ
T_Δ
= T_1 U .. U T_m
T_{m+1} U..U T_n
D-simulate
Gθ or
its D-expansion
T_0
in S_0
T_0'
Δθ
T_1' .. T_m'
T_{m+1}' U..U T_n'
D-simulate
Gθ
T'
in S'
Cθ
Hθ
(T - Δ)θ
T'''
T_{m+1}'' U..U T_n''
D-simulate
induction hypothesis
induction hypothesis
H
T''
in S
(assumption)

Fig. 3

D : {P(x) ← Q(f(x))}

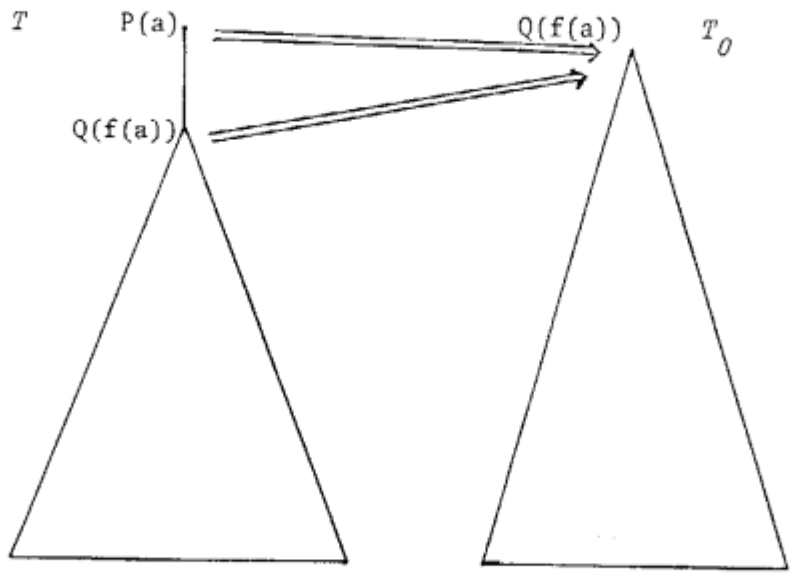

Fig. 4