TR-131

Representation Theorems and Primitive Predicates for Logic Programs

by
Takashi Yokomori
(Fujitsu Ltd.)

August, 1985

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

Representation Theorems and Primitive Predicates for Logic Programs

by

Takashi YOKOMORI

August 1985

International Institute for Advanced Study of
Social Information Science, Fujitsu Limited
140 Miyamoto, Numazu, Shizuoka 410-03 JAPAN

ABSTRACT

We present several representation theorems for logic programs in terms of formal grammatical formulation. First, for a given logic program P the notion of the success language of P is introduced, and based on this language theoretic characterization of a logic program several types of representation theorems for logic programs are provided. Main results include that there effectively exists a fixed logic program with the property that for any logic program one can find an equivalent logic program such that it can be expressed as a conjunctive formula of a simple program and the fixed program. Further, by introducing the concept of an extended reverse predicate, it is shown that for any logic program there effectively exists an equivalent logic program which can be expressed as a conjunctive formula consisting of only extended reverse programs and append programs.

1. Introduction

Since, needless to say the original work of Colmerauer and Kowalski([Co 70],[Ko 74]), a recent world-wide trend on FGCS conception has been one of the primary subjects, there are numerous work on logic programming languages and the theory of logic programs. It is well accepted that, among others, the research on a subset of first-order predicate logic called Horn clause logic has taken the central position in this area because of its importance of providing an interesting formal computation model for a programming language PROLOG. As is well-known, PROLOG, based on the procedural interpretation to Horn clause logic, has an operational semantics determined by the resolution principle. In the context of the semantics of predicate logic as a programming language van Emden and Kowalski ([EK 76]) have studied on model-theoretic, operational and fixedpoint semantics of logic programs, while using a Turing machine formulation Shapiro ([Sh 84]) has defined and argued a kind of model-theoretic semantics of logic programs.

In this paper we are concerned with establishing several representation theorems for "logic programs(Horn clause programs)" in terms of formal language theoretic formulation. In course of the formal grammatical treatment of logic programs we introduce the notion of the success language of a logic program over a finite alphabet, which turns out to be another way of providing a model-theoretic semantics for logic programs. Here, by formal grammars we mean generative grammars of Chomsky, and it should be remarked that the theory of formal languages([Sa 73],[Ha 78],[HU

79]) has been well-developed enough in itself to make a lot of contributions to other research areas such as the theory of logic programming. This view may be supported, for example, when we think of the similarity between the refutation process in logic programs and the derivation steps in context-free grammars, and note that logic programs can be regarded as a kind of an extention of context-free grammars. In fact, Shapiro investigates the computational complexity of logic programs using the similarity of their operational behaviors to those of alternating Turing machines.([Sh 84])

With the help of an encoding technique it is shown that one can associate a logic program with a formal language (the success language mentioned above) over a finite alphabet. This leads to a semantic characterization of logic programs as previously mentioned, although that is not our primary concern in the current paper. This kind of semantic approach to logic programs has been already preceeded by the paper [Yo 85]. It has been shown that any recursively enumerable language can be specified as a conjunctive formula of two deterministic logic programs and one simple logic program that serves as a mapping on the set of words. The work in this paper is motivated by the result above and extends it to present a variety of the ways of representing logic programs.

In this paper we present several representation theorems for logic progams which assert that there effectively exists a fixed logic program (we may call it generator program) with the property that for any logic program one can find an equivalent logic

program such that it can be expressed as a conjunctive formula of a simple program and the fixed program.

Further, by analysing components in the representation results, it is shown that the "filtering function" serving as a homomorphism mapping and the "merging function" are sufficiently primitive in the sense that for any logic program there is an equivalent logic program which can be expressed within the use of combination of these two programs. By introducing the concept of "extended reverse", it is also proved that for any logic program one can find an equivalent logic program expressed as a conjunctive formula consisting of only "extended reverse" programs and "append" programs.

This paper is organized as follows. Section 2 is concerned with terminology, basic notions and results needed through the paper. In Section 3 several representation theorems for logic programs are established. Section 4 deals with the problem of what operations (predicate) is primitive for the representation formula obtained in Section 3. Concluding remarks and the future reseach direction are briefly given in Section 5. Appendix provides a proof for Lemma 3.4 which is used in the text to derive a representation result of logic programs.

2. Preliminaries

2.1 Formal Grammars and Their Languages

We now introduce a generative device which plays the main

role in all of subsequent sections in this paper.

Definition.

A generative grammar is an ordered quadruple $G = (N, T, P, S_0)$ where N and T are disjoint finite alphabets, S_0 is in N, and P is a finite set of production rules of the form $Q_1 \to Q_2$ such that Q_2 is a word over the alphabet $V = N \cup T$ and Q_1 is a word over V containing at least one symbol of N. The elements of N are called nonterminals and those of T terminals; S_0 is called the initial symbol.

A word u generates directly a word v, in symbols, $u \Rightarrow v$, if and only if there are words u', u", Q_1, Q_2 such that $u = u'Q_1u"$, $v = u'Q_2u"$, and $Q_1 \to Q_2$ belongs to P. Thus, $\Rightarrow$ is a binary relation on the set V^* (the set of all words over V including empty word e). We denotes $V^* - \{e\}$ by V^+. Let $\Rightarrow^*$ be the reflexive, transitive closure of $\Rightarrow$. The language L(G) generated by G is defined by

$$L(G) = \{ w \text{ in } T^* \mid S_0 \Rightarrow^* w \}.$$

L(G) is called a language over T (or on T^*).

Grammars are, in general, classified by the form of production rules, which yields a hierarchy of corresponding language families.

Definition.

A generative grammar is also called phrase structure grammar. Let $G = (N,T,P,S_0)$ be a phrase structure grammar. Then, G is called

(i) context-free if each production rule is of the form X -> Q, where X in N, and Q in V^*,

(ii) regular if each production rule is of one of the two form X->a or X->aY, where a in T and X,Y in N, with the possible exception on the production rule S_0 -> e whose occurrence in P implies that S_0 does not occur on the right hand side of any rule in P.

Definition.

(1) Let G = (N,T,P,S_0) be a context-free grammar with the property that (i) every rule in P is of the form A -> ax, where A in N, a in T, x in N^*,and (ii)for all A in N, a in T, A->ax and A->ay in P implies x=y. Then, G is called simple deterministic.

(2) A context-free grammar G = (N,T,P,S_0) is called linear if P consists of the rules of the form : A -> uBv, or A -> w, where A,B in N, u,v,w in T^*.

Definition.

Let L be a subset of T* for some alphabet T, and let X be in {phrase structure, context-free, simple deterministc,linear, regular}. Then, L is called an X language if L=L(G) for some X grammar G. Further, a language generated by a phrase structure grammar is also called recursively enumerable.

Let $r \geq 1$ and $T_r = \{a_1,\ldots,a_r\}$. Further, let $G_r = (\{S_0\}, T, P, S_0)$ be a context-free grammar, where $T=T_r \cup \{\tilde{a} \mid a \text{ in } T_r\}$, and $P = \{S_0 \to S_0 a_i S_0 \tilde{a}_i S_0 \mid 1 \leq i \leq r\} \cup \{S_0 \to e\}$. Then, $L(G_r)$ is called the Dyck

language over T_r and denoted by D_r.

Definition.

Let T be an alphabet. For each a in T, let f(a) be a word (possibly over a different alphabet from T). Then, let f(e) = e, f(xy) = f(x)f(y) (x,y in T^*). The mapping f is extended to the power set of T^* as follows: for each L over T, $f(L) = \{f(w) \mid w \text{ in } L\}$. The mapping f is called a homomorphism on T^*. Let f be a homomorphism on T^*, and let K be the alphabet of the range of f. Then, a mapping f^{-1} defined by

for L over K, $f^{-1}(L) = \{ w \text{ in } T^* \mid f(w) \text{ in } L \}$,

is called the inverse homomorphism of f.

Definition.

A homomorphism f on T^* is called

(1) a coding if for each a in T, f(a) is a symbol,

(2) a weak coding if for each a in T, f(a) is either a symbol or the empty word e,

(3) a weak identity if for each a in T, f(a) is either the symbol a itself or the empty word e.

Definition.

A deterministic generalized sequential machine(dgsm) with acccpting states is a 6-tuple $A = (Q,T,D,d,q_0,F)$, where

Q : a finite set of states, T : a finite set of input symbols,

D : a finite set of output symbols, d : transition function from $Q \times T$ to $Q \times D^*$, q_0 : the initial state in Q, and

F : a subset of Q (a set of final states).

The function d is extended to $Q \times T^*$ as follows: for q in Q, x in T^*, a in T,

$d(q,e) = (q,e)$,

$d(q,ax) = (r,y)$

where

$y = w_1 w_2$

$d(q,a)=(p,w_1)$, $d(p,x) = (r,w_2)$ for some p in Q, w_1, w_2 in D^*.

Let f_A be a mapping defined by

$f_A(x) = y$ iff $d(q_0,x) = (p,y)$ for some p in F.

The mapping f_A so defined is called a dgsm mapping of A.

Notation.

Let T be a finite alphabet. For a word $w = a_1 \cdots a_n$ $(n \geq 0)$ in T^*, the $(\tilde{})$-version $\tilde{w}$ denotes $\bar{a}_1 \cdots \bar{a}_n$. Further, w^R denotes the reverse $a_n \cdots a_1$.

2.2 Logic Programs and Their Languages

This subsection introduces the concepts of a logic program and its associated language we shall deal with in the subsequent sections. We assume the reader to be familiar with the rudiments of mathematical logic.

Definition

A logic program is a finite set of Horn clauses, which are universally quantified logical sentences of the form

$$A \leftarrow B_1,\ldots,B_n \quad (n \geq 0) \qquad (C)$$

where the A and the B's are atomic formulae. In the above clause (C) A is called the clause's head, while B's are called the clause's body. If n = 0, then we simply donote it by A instead of A <-.

Atomic formulae occurring in a logic program are called goals. A program is said to be dominated by a goal if the predicate name of the goal occurs only once as the head of a clause in the program.

[Notational Convention]

(i) We use upper-case letters such as X, Y, Z for variable symbols and lower-case letters such as x, y, z for ground terms. For terms, letters t, s, r are often used. The boldface versions like **P**, **Q** are used for logic programs, while normal upper-case letters like P, Q are used for goals, and lower-case letters p,q for goal names.

(ii) For a logic program dominated by a goal, we sometimes refer to the program in terms of the name of the goal. In such a case it is assumed that the program name is the capital letter P of the goal name "p".

Definition.

Let **P** be a logic program and Q a goal. If there is a refutation of a goal Q from **P**, then we say P succeeds on Q, or Q succeeds (in P).

In this paper we are concerned with logic programs whose data domains are finitely generated by a fixed set of symbols.

Definition.

Let P be a logic program. The Herbrand universe of P is the set of all ground terms constructable from the set of constants C and the set of function symbols F occuring in P, and we denote it by D(F,C). Then, a logic program P is called a logic program over C if F comprises only one function symbol, and its Herbrand universe is denoted by D(C).

As shown below, any Herbrand universe for a logic program can be coded in an appropriate manner into the domain D(T) constructed from some fixed finite set of symbols T. In other words, any ground term which possibly appears in a program can be taken as a word over some finite alphabet T.

Lemma 2.1

There exist a fixed finite set of symbols T and a one-to-one mapping f such that for any logic program P with the domain D(F,C) and for any goal $p(X_1,\ldots,X_n)$ there exist a logic program P' with the domain D(T) and a goal p'(X) with the property that P succeeds on $p(x_1,\ldots,x_n)$ iff P' succeeds on p'(x), where $x=f(x_1,\ldots,x_n)$.

Proof.

Let $g_1,g_2,\ldots$ be an enumeration of all function symols occurring in D(F,C) of P. (Note that a constant k can be taken as a 0-ary function symbol as in k().)

Introduce a mapping c from the set D(F,C) to the set of lists as follows :

for a term $t = g_i(s_1,...,s_m)(m \geq 0)$,

$c(t) = [\%,@^i,\$,c(s_1),...,c(s_m),\$]$.

where "[" and "]" are the list notation ,$,$,@,% are new symbols, and $@^i$ denotes a sequence @,...,@ of i @s.

Further, for an n-tuple of terms $(t_1,...,t_n)$, let f be defined by

$f(t_1,...,t_n) = flatten([c(t_1),\#,...,\#,c(t_n)])$,

where "flatten" is a mapping of flattening lists,

is a new symbol(argument separator).

Define p'(X) as follows :

$p'(X) \leftarrow flat(X_1,...,X_n,X),\ p(X_1,...,X_n)$ --- (C_0)

where $flat(X_1,...,X_n,X)$ succeeds iff $X = f(X_1,...,X_n)$.

Further, let P' be $P \cup \{C_0\}$. Then, it is easily seen that P' succeeds on $p'(f(x_1,...,x_n))$ iff P succeeds on $p(x_1,...,x_n)$ for x_i in D(F,C). Let T = {#,$,$,@,%,NIL}, where NIL denotes empty list, then D(T) is the set of lists constructed from T and the unique function symbol of the list constructor. Obviously, this satisfies the desired conditions. □

Thus, it is sufficient for general discussion to deal with only logic programs over some fixed finite set.

[Conventions]

(1) In what follows, it may be assumed that (i)a logic program over T has the domain of all lists constructed from a finite set of constants T, and (ii) otherwise specified, a goal is assumed

to be a 1-ary predicate.

(2) As a notation, given finite set of symbols T and a word $w = a_1 \cdots a_n$ on T^*, the boldface **w** denotes the list version $[a_1,\ldots,a_n]$.

Logic programs together with goals are classified by the types of their associated languages.

Definition.

Let P be a logic program over a finite set of symbols T and Q (=q(X)) be a goal in **P**.

(i) A language over T defined by

$$L(P,Q,T) = \{ w \text{ in } T^* \mid P \text{ succeeds on } q(\mathbf{w}) \}$$

is called the success language of Q in **P**. In this case L(P,Q,T) is often denoted by L(P,q,T). If P is dominated by p(X) or a program "**P**" is named after the goal name "p", then we simply write L(P,T) and call it the success language of **P**.

Further,

(ii) A logic program P is called X if L(P,Q,T) is an X language for all goal Q in **P**.

(iii) Let p(X,Y) be a goal dominating **P**, and for x in T^*, let $f_P(x) = \{y \text{ in } T^* \mid P \text{ succeeds on } p(x,y)\}$. Then, a logic progam **P** is called

(1) homomorphism if f_P is a homomorphism,

(2) (weak) coding if f_P is a (weak) coding,

(3) weak identity if f_P is a weak identity,

on T^*.

Finally,

(iv) Let **P** and **P'** be two logic programs over T, and let p(X) and p'(X) be goals in **P**, **P'**, respectively. Then **P** with p(X) and **P'** with p'(X) are <u>equivalent</u> if L(P,p,T) = L(P',p',T).

We end this section with presenting a result showing the expressive capability of logic programs we are dealing with in this paper.

It has been shown in literature ([Tä 77],[Yo 85]) that for any recursively enumerable language L over T, there exist a logic program P over T and a goal Q such that L is the success language of Q in P. Conversely, it is shown that for any logic program **P** over T and a goal Q, the success language L(P,Q,T) is a recursively enumerable language, which is proved by constructing a Turing machine simulating the resolution process for Q from **P** and accepting the success language of Q in P ([Sh 84]). (Note that a language is recursively enumerable if and only if it is accepted by a Turing machine.)

Hence, we have the following :

Theorem 2.1

The class of success languages of logic programs is equal to the class of recursively enumerable languages.

It may be possible to state that the success language of a

logic program provides us a kind of model-theoretic semantics (or denotational semantics) for logic programs.

3. Representation Theorems

In this section several representation theorems for logic programs are presented. Most of them are easily obtained from the corresponding results in formal language theory.

3.1 Generator Programs for Logic Programs

We shall show that there exists a fixed logic program from which for any logic program an equivalent logic program can be obtained in terms of the composition of simpler logic programs. Such a fixed logic program may be called generator program.

[1] Generator Program R_0

First we shall show that there exists a fixed simple program which plays a role of generator for the class of logic programs. Such a program can be obtained by making a slight modification to "reverse" program.

Lemma 3.1

For any recursively enumerable language L over an alphabet T there exist a simple deterministic language Sp on $K^+\tilde{K}^+$ (for some

alphabet K including T), and a weak identity h such that L = $h(\{w\tilde{w}^R \mid w \text{ in } K^+\} \cap Sp)$, where $Sp = \{x\not\ddagger\bar{y}^R \mid f(x)=y\}$, f is a dgsm mapping of $A = (Q,K,D,d,q_0,F)$ depending on L, h preserves the alphabet of L and erases other symbols.

(See Theorem 11 in [ER 80])

Theorem 3.1(Representation Theorem 1)

Let T be a fixed alphabet. Then, there exists a fixed logic program $\mathbf{R_0}$ with the property that for any logic program P over T with a goal p(X) one can find an equivalent logic program **P'** with a goal p'(X) such that it can be expressed by

$$p'(X) \leftarrow r_0(X,Y),\ s_P(Y) \qquad (3\text{-}1)$$

for some simple deterministic program $\mathbf{S_P}$.

Proof.

From Theorem 2.1 and Lemma 3.1, for any logic program P over T with a goal p(X) there is a simple deterministic language Sp on $K^+\bar{K}^+$ and a weak identity h such that $L(P,p,T) = h(\{w\tilde{w}^R \mid w \text{ in } K^+\} \cap Sp)$, where $Sp = \{x\not\ddagger\bar{y}^R \mid f(x)=y\}$, f is a dgsm mapping of $A = (Q,K,D,d,q_0,F)$ depending on L(P,p,T), and h(a) = a (for all a in T), h(a) = e (otherwise).

Construct three logic programs so that $\mathbf{M_T}$, $\mathbf{I_T}$, and $\mathbf{S_P}$ may determine the language $M_0(=\{w\tilde{w}^R \mid w \text{ in } K^+\})$, h, and Sp, respectively.

(1) $\mathbf{M_T}$ is defined as follows :

$$m_T(X) \leftarrow m1(s_1,X,[\,])$$

$$m1(s_1,[a|X],Y) \leftarrow m1(s_1,X,[a|Y]) \quad \text{(for all a in K)}$$

$$m1(s_1,[\bar{a}|X],Y) \leftarrow m1(s_2,[\bar{a}|X],Y) \quad \text{(for all a in K)}$$

$m1(s_2,[\,],[\,])$

$m1(s_2,[\tilde{a}|X],[a|Y])$ <- $m1(s_2,X,Y)$ (for all a in K)

Clearly $\mathbf{M_T}$ determines the mirror image language, i.e.,

$L(\mathbf{M_T},K \cup \tilde{K}) = \{\, w\tilde{w}^R \mid w \text{ in } K^+ \,\}$.

(2) $\mathbf{I_T}$ is defined as follows :

$i_T([\,],[\,])$

$i_T([a|X],[a|Y])$ <- $i_T(X,Y)$ (for all a in T)

$i_T(X,[a|Y])$ <- $i_T(X,Y)$ (for all a not in T)

$\mathbf{I_T}$ is a simple projection mapping which preserves symbols from T and erases others.

(3) $\mathbf{S_P}$ is defined as follows :

$s_P(X)$ <- $s1(q_1,X,[\,])$

$s1(q_1,[a|X],Y)$ <- $s1(q_1,X,[a|Y])$ (for all a in $K \cup \{¢\}$)

$s1(q_1,[\tilde{¢}|X],[¢|Y])$ <- $s1(q_f,X,Y)$ (for all q_f in F)

$s1(q_0,[\,],[\,])$

$s1(q,[\tilde{w}^R|X],[a|Y])$<-$s1(p,X,Y)$ (for all $d(p,a) = (q,w)$)

where $A = (Q,K,D,d,q_0,F)$ is a dgsm A given in Lemma 3.1.

Then, $L(\mathbf{S_P}, K \cup \tilde{K} \cup \{¢,\tilde{¢}\}) = \{x¢\tilde{¢}\tilde{y}^R \mid f(x)=y,\ x \text{ in } K^*\}$

Let P' be a logic program defined by $p'(X)$<-$i_T(X,Y)$, $m_T(Y)$, $s_P(Y)$. It is easily seen that for x in T^*,

x is in $L(\mathbf{P},p,T)$ iff there exists y such that $x = h(y)$ and

y is in $M_0 \cap Sp$

iff there exists y such that

$\mathbf{I_T}$ succeeds on $i_T(\mathbf{x},\mathbf{y})$,

$\mathbf{S_P}$ succeeds on $s_P(\mathbf{y})$, and

M_T succeeds on $m_T(y)$

iff P' succeeds on p'(x).

Let R_0 be defined by $r_0(X,Y) \leftarrow i_T(X,Y), m_T(Y)$.
(Since T is fixed, R_0 is a fixed program.) Thus, p'(X) can be expressed as the desired form (3-1). □

[2] Generator Program M_0

We show that a kind of "merge" program can also play a role of generator as well as the program R_0.

Lemma 3.2

For any recursively enumerable language L over an alphabet T there exist a weak identity h and a regular language R such that L = h(shuffle(K)∩ R), where K is some alphabet including T, shuffle(K) = $\{x_1\tilde{y}_1 \cdots x_n\tilde{y}_n \mid x_1 \cdots x_n = y_1 \cdots y_n \text{ in } K^*\}$, R = f($K^*$), f is a mapping induced by a dgsm B = (Q, K, K∪ $\tilde{D}$, d', q_0, F) defined by a dgsm A =(Q, K, D, d, q_0, F) depending on L, d'(q,a) = (p,a$\tilde{w}$) iff d(q,a) = (p,w), h preserves the alphabet T and erases other symbols.
(See Theorem 13 in [ER 80])

Theorem 3.2(Representation Theorem 2)

Let T be a fixed alphabet. Then, there exists a fixed program M_0 with the property that for any logic program P over T with a goal p(X) one can find an equivalent logic program P' with p'(X) such that it can be expressed by

$$p'(X) \leftarrow m_0(X,Y),\ r_p(Y) \qquad (3\text{-}2)$$

for some regular program $\mathbf{R_p}$.

Proof.

Analogous to the proof of Therem 3.1, it suffices to show that the following three logic programs satisfy the condition stated in Lemma 3.2.

(1) $\mathbf{ME_T}$ is defined by

$me_T(X) \leftarrow me1(X,[\,],[\,])$

$me1([\,],X,X)$

$me1([\tilde{a}|X],Y,Z) \leftarrow me1(X,Y,[a|Z])$ (for all a in K)

$me1([a|X],Y,Z) \leftarrow me1(X,[a|Y],Z)$ (for all a in K).

$\mathbf{ME_T}$ determines what is called the twin shuffle language, i.e.,

$L(\mathbf{ME_T}, T \cup \tilde{T}) = \{ x_1\tilde{y}_1 \cdots x_n\tilde{y}_n | x_1 \cdots x_n = y_1 \cdots y_n \text{ in } K^* \}$.

(2) $\mathbf{I_T}$ is the same as the one defined above in the proof for Theorem 3.1.

(3) $\mathbf{R_p}$ is defined :

Let $A = (Q,K,D,d,p_0,F)$ be a given dgsm in Lemma 3.2.

$r_p(X) \leftarrow r1(p_0,X,[\,])$

$r1(p,[a|X],Y) \leftarrow r1(p_a,X,[a|Y])$ (for all $d(p,a) = (q,w)$)

$r1(p_a,[\tilde{w}|X],[a|Y]) \leftarrow r1(q,X,Y)$ (for all $d(p,a) = (q,w)$)

$r1(p_f,[\,],[\,])$ (for all p_f in F)

$L(\mathbf{R_p}, K \cup \tilde{K}) = \{a_1\tilde{w}_1 \cdots a_n\tilde{w}_n \mid d(p_0,a_1 \cdots a_n) = (q,w_1 \cdots w_n), q \text{ in } F\}$

$= f(K^*)$.

Let $\mathbf{M}_0$ be defined by

$m_0(X,Y) \leftarrow i_T(X,Y), me_T(Y)$.

(Again, since T is fixed, $\mathbf{M}_0$ is a fixed program.)

Further, let P' be defined by $p'(X) \leftarrow m_0(X,Y), r_P(Y)$.

To complete the proof it suffices to check if the following relation holds :

for x in T^*, P' succeeds on p'(x) iff x is in L(P,p,T). □

[3] Generator Program D_o

It is demonstrated that a program which behaves as a checker for well-pairedness can be a generator for the class of logic programs.

Lemma 3.3

For any recursively enumerable language L over T, there exist a linear grammar $G_L = (\{S\}, T' \cup \tilde{T}', P_L, S)$ and a weak coding h satisfying the following properties that

(i) $L = h(D_r \cap L(G_L))$,

(ii) T is a subset of T', and $h(a) = a$ (for all a in T),

$h(a) = e$ (for all a in $T' \cup \tilde{T}' - T$),

(iii) $P_L = \{S \rightarrow u_i S v_i \mid 1 \leq i \leq n\} \cup \{S \rightarrow w\}$,

where $w, u_i, v_i (1 \leq i \leq n)$ are in $(T' \cup \tilde{T}')^*$, D_r is the Dyck language over T'(r:the cardinality of T'), $\tilde{T}' = \{\tilde{a} \mid a \text{ in } T'\}$.

(See [HOY 85] for the proof.)

[Important Remarks]

(1) A linear grammar G_L, which is called a minimal linear grammar [CS 62], depends on L, while h depends on only T.

(2) A careful and patient observation of the proof for Lemma 3.3 in [HOY 85] leads to the fact that by making a slight modification one can obtain another G_L with its additional property, that is,

(iv) none of the two among w, $u_i, v_i (1 \leq i \leq n)$ is identical, each of them is nonempty and w does not depend on L.

Theorem 3.3 (Representation Theorem 3)

Let T be a fixed alphabet. Then, there exists a fixed program $\mathbf{D_0}$ with the property that for any logic program P over T with a goal p(X) one can find an equivalent logic program **P'** with a goal p'(X) such that it can be expressed by

$$p'(X) \leftarrow d_0(X,Y),\ lin_P(Y) \qquad (3\text{-}3)$$

for some linear program $\mathbf{LIN_P}$.

Proof.

From Theorem 2.1 and Lemma 3.3 for any logic program P over T anda goal p(X) there exist a homomorphism h from $(T' \cup \tilde{T}')^*$ to T^* and a linear grammar G_L with the property described above, and that x is in L(P,p,T) iff there is y such that h(y) = x and y is in $D_r \cap L(G_L)$, where r is the cardinality of T'.

Construct two logic programs $\mathbf{D_T}$, $\mathbf{LIN_P}$ so that it may hold that (i) $L(\mathbf{D_T}, T' \cup \tilde{T}') = D_r$, and (ii) $L(\mathbf{LIN_P}, T' \cup \tilde{T}') = L(G_L)$:

(1) $d_T(X) \leftarrow dyck(X,[])$

dyck([],[])

dyck([a|X],Y) <- dyck(X,[a|Y]) (for all a in T')

dyck([$\tilde{a}$|X],[a|Y]) <- dyck(X,Y) (for all a in T')

(2) $lin_P(X)$ <- $lin(p_1,X,[])$

$lin(p_1,[w|X],Y)$ <- $lin(p_2,X,Y)$

(w is the word such that S -> w in P_L)

$lin(p_2,[],[])$

$lin(p_1,[u_i|X],Y)$ <- $lin(p_1,X,[u_i|Y])$

(for all S -> u_iSv_i in P_L of G_L)

$lin(p_2,[v_i|X],[u_i|Y])$ <- $lin(p_2,X,Y)$

(for all S -> u_iSv_i in P_L of G_L).

Since it is almost obvious that $L(\mathbf{D}_T,T'\cup\bar{T}')= D_r$, we shall check that $L(\mathbf{LIN}_P,T'\cup\bar{T}') = L(G_L)$. For any x in $L(G_L)$, there is a sequence of production rules $r_1,\ldots,r_k,r_0$ such that

$x = u_{i1}\cdots u_{ik}wv_{ik}\cdots v_{i1}$, r_j : $S\to u_{ij}Sv_{ij}(1\leq j\leq k)$

and r_0 : S -> w.

Let $x=x_1wx_2$, where $x_1=u_{i1}\cdots u_{ik}$, $x_2=v_{ik}\cdots v_{i1}$, then we have that $lin_P(x)$ succeeds if $lin(p_1,x,[])$ succeeds, and that $lin(p_1,x_1wx_2,[])$ succeeds if $lin(p_1,wx_2,f(x_1))$ succeeds,

if $lin(p_2,x_2,f(x_1))$ succeeds,

if $lin(p_2,[],[])$ succeeds,

where f is defined by $f(e)=e$, $f(u_jx)=f(x)f(u_j)$ for u_j in $\{u_1,\ldots,u_n\}$, x in $\{u_1,\ldots,u_n\}^*$.

Thus, we eventually have $lin_P(x)$ succeeds. The converse relation is straightforwardly proved. Hence, it is obtained that $L(\mathbf{LIN}_P,T'\cup\bar{T}') = L(G_L)$.

Now, let $\mathbf{D}_0$ be defined by $d_0(X,Y)$ <- $i_T(X,Y),d_T(Y)$, where $i_T(X,Y)$ is a predicate already appeared in Theorem 3.1 and Theorem 3.2. (Since T is fixed, so is $\mathbf{D}_0$.) Further let $p'(X)$ <- $d_0(X,Y)$,

$lin_P(Y)$. To complete the proof, we have only to show that p'(x) succeeds iff x is in L(P,p,T), and this is easily checked in the following way :

p'(x) succeeds iff there is y such that $i_T(x,y)$, $d_T(y)$, and
$lin_P(y)$ succeed
iff there is y such that h(y)=x, y in $L(D_T,T'\ \tilde{T}')$,
and y in $L(G_L)$
iff x is in L(P,p,T). □

[Remark]

(i) A program D_T whose success language is a Dyck language works for checking "well-pairedness" of an input string in D_0.

(ii) A program structure of LIN_P is quite similar to that of S_P in Theorem 3.1.

Later we will discuss the close relationships among these generator programs and what operations are really primitive for expressing logic programs.

3.2 Decomposing Logic Programs

As we have seen in the previous subsection, a logic program can be expressed as a conjunctive formula comprising a simpler program and a fixed program consisting of two components. Further, one of the two is quite simpler than the other in that it just works as a simple homomorphism(actually, a weak identity mapping).

We shall show a representation theorem for logic programs in which for any logic program one can find an equivalnt logic program expressed as a conjunctive formula of two fixed programs and three simple homomorphism programs. Exactly, one of the three can be fixed.

Lemma 3.4

For any simple deterministic (context-free) language L, there exist a coding f and a homomorphism h such that $L = f(h^{-1}(\not{c}D_2))$, where D_2 is a Dyck language, $\not{c}$ is a (new) symbol.

(The way of the proof for Lemma 3.4 is similar to that of the proof for the main theorem in [Gr 73]. See Appendix for the proof.)

This lemma leads to another representation for logic programs which may be called "decomposition theorem" for logic programs.

Theorem 3.4 (Representation Theorem 4)

Let T be a fixed alphabet. Then, there exist fixed logic programs I,D and M with the property that for any logic program P over T with a goal p(X) one can find an equivalent logic program P' with a goal p'(X) such that it can be expressed by

$$p(X) \leftarrow i(X,Y),m(Y),f_P(Y,V),h_P(V,Z),d(Z) \qquad (3\text{-}4)$$

for some coding program F_P and a homomorphism program H_P.

Proof.

From the proof of Theorem 3.1 there exist fixed programs M_T and I_T such that for a given logic program P with a goal p(X) one

can have an equivalent logic program P' with a goal p'(X) such that it is expressed by p'(X) <- $i_T(X,Y)$, $m_T(Y)$, $s_p(Y)$, for some simple deterministic program S_p. Further, Lemma 3.4 tells that there exist a coding f_L and a homomorphism h_L such that L (=L(S_p, $K \cup \bar{K}$))=$f_L(h_L^{-1}(\not{c}D_2))$. Let I be I_T and M be M_T. Then, it suffices to show that one can construct a coding program F_p, a homomorphism program H_p and a fixed program D such that (i) D determines the Dyck language $\not{c}D_2$, and (ii) $s_p(Y)$ can be expressed as a conjunctive formula of $f_p(Y,V)$,$h_p(V,Z)$ and d(Z).

Define F_p,H_p and D as follows :

$f_p([\,],[\,])$

$f_p([a|X],[b|Y])$ <- $f_p(X,Y)$ (for all $f_L(b) = a$)

$h_p([\,],[\,])$

$h_p([b|X],[x_1,\ldots,x_m|Y])$ <- $h_p(X,Y)$ (for all $h_L(b) = x_1 \cdots x_m$).

d(X) <- unif(X,[$\not{c}$|Y]), dyck(Y,[ ])

dyck([ ],[ ])

dyck([a_1|X],Y) <- dyck(X,[a_1|Y])

dyck([a_2|X],Y) <- dyck(X,[a_2|Y])

dyck([$\tilde{a}_1$|X],[a_1|Y]) <- dyck(X,Y)

dyck([$\tilde{a}_2$|X],[a_2|Y]) <- dyck(X,Y)

where unif(X,Y) succeeds iff X and Y are unifiable.

Clearly, L(D,{a_1,a_2,$\tilde{a}_1$,$\tilde{a}_2$,$\not{c}$}) = $\not{c}D_2$, and it is easily seen that S_p succeeds on $s_p(y)$
iff y is in L(S_p, $K \cup \bar{K}$)

iff there exist v and z such that $f_L(v) = y, h_L(v) = z$

iff there exist v and z such that

F_P succeeds on $f_P(y,v)$,

H_P succeeds on $h_P(v,z)$, and

D succeeds on d(z).

This implies that $s_P(Y)$ <- $f_P(Y,V)$, $h_P(V,Z)$, d(Z). Thus, eventually, we have that p(X) <- i(X,Y), m(Y), $f_P(Y,V)$, $h_P(V,Z)$, d(Z). This completes the proof. □

[Remark]

The teaching of Theorem 3.4 is that using two fixed logic programs **M**(a modified "reverse" program) and D (a "checking well-pairedness" program) any logic program P can be reducible into three homomorphism programs **I**, F_P and H_P that have a very simple structure.

4. What are primitives ?

We have seen in Section 3 that several specific types of logic programs can play a significant role as a generator in expressing logic programs. In this section we shall discuss this issue on generator in more detail.

4.1 Primitives for Generators

Getting back to the representation theorems, a generator

program R_0 in (3-1) of Theorem 3.1 was constructed from a weak identity program I_T and a logic program M_T, i.e.,

$r_0(X) \leftarrow i_T(X,Y), m_T(Y)$

where

[0] $i_T([],[])$

$i_T([a|X],[a|Y]) \leftarrow i_T(X,Y)$ (for all a in T)

$i_T(X,[a|Y]) \leftarrow i_T(X,Y)$ (for all a not in T), and

we observe that $m_T(X)$ can be re-defined as follows :

[1] $m_T(X) \leftarrow$ append(Y,Z,X), copy(Y,Y'), reverse(Y',Z)

copy([],[])

copy([a|X],[$\tilde{a}$|Y]) <- copy(X,Y) (for all a in K)

reverse([],[])

reverse([X|Y],Z) <- reverse(Y,T), append(T,[X],Z).

Similarly, from an observation of a generator program M_0 in (3-2) of Theorem 3.2 we have :

$m_0(X) \leftarrow i_T(X,Y), me_T(Y)$

where

[2] $me_T(X) \leftarrow$ merge(Y,Z,X), copy(Y,Z)

merge(X,[],X)

merge([],X,X)

merge([a|X],Y,[a|Z]) <- merge(X,Y,Z) (for all a in K)

merge(X,[$\tilde{a}$|Y],[$\tilde{a}$|Z]) <- merge(X,Y,Z) (for all a in K)

Further, a generator program D_0 in (3-3) of Theorem 3.3 is analysed as follows :

$d_0(X) \leftarrow i_T(X,Y), d_T(Y)$

where

[3] $d_T(X)$ <- dyck(X,[])

dyck([],[])

dyck([a|X],Y) <- dyck(X,[a|Y]) (for all a in T')

dyck([$\tilde{a}$|X],[a|Y]) <- dyck(X,Y) (for all a in T').

It is easily seen that each generator contains a common homomorphism (exactly weak-identity) program I_T which serves as a kind of "filter". That means the essentially unique parts of generator programs are $m_T(X)$, $me_T(X)$ and $d_T(X)$.

Thus, it is possible to say that "append", "copy","merge", "dyck" are all primitives for a generator program in the representation theorem. However, noting that "copy" is a special type of a homomorphism program and "append" and "dyck" are restricted versions of "merge", we may conclude that the filtering function ("homomorphism") and the merging function("merge") are fully primitive for expressing logic programs.

4.2 Extended Reverse Programs

We shall show there exists a type of logic program which can take the place of various basic programs appearing in the representation results.

Let f be a mapping from T^* to K^*. Then, consider a logic program dominated by a predicate "(f)-reverse(X,Y)", which is defined by (f)-reverse(**x**,y) succeeds iff so does reverse(**f(x)**,y).

We call this extended reverse program. (Notice that if f is an identity, then (f)-reverse(X,Y) is an ordinary "reverse" predicate.)

Example 1.

Let f be defined by f(a)=$\tilde{a}$, f(b)=$\tilde{b}$, f(c)=$\tilde{c}$. Then, (f)-reverse (X,Y) may be, for example, defined as follows:

(f)-reverse(X,Y) <- rev(X,[ ],Y)

rev([ ],X,X)

rev([a|X],Y,Z) <- rev(X,[$\tilde{a}$|Y],Z)

rev([b|X],Y,Z) <- rev(X,[$\tilde{b}$|Y],Z)

rev([c|X],Y,Z) <- rev(X,[$\tilde{c}$|Y],Z).

Let p(X) <- append(Y,Z,X), (f)-reverse(Y,Z), then the success language of this program $\{w\tilde{w}^R \mid w \text{ in } (a,b,c)^*\}$ is context-free.

Now, let us see the next one.

Example 2.

Let f be a mapping defined by $f(x) = \tilde{x}^R$, for all x in T^*. Then, it is seen that

(f)-reverse(x,y) succeeds iff reverse(**f(x)**,y) succeeds

iff reverse($\tilde{x}^R$,y) succeeds

iff $\tilde{x}$ = y.

Let P be a program dominated by p(X) <- append(Y,Z,X), (f)-reverse(Y,Z). Then, the success language $L(P,T \cup \tilde{T})$ is $\{w\tilde{w} \mid w \text{ in } T^*\}$ which is context-sensitive.

Thus, (f)-reverse can define a number of different classes of logic programs by varing a mapping f.

Now we wish to call back one's attention to the representation theorems. In the representation formula (3-1) of Theorem 3.1 a logic program can be expressed by

$p(X) \leftarrow r_0(X,Y), s_p(Y)$, where

(0) $r_0(X,Y) \leftarrow i_T(X,Y), m_T(Y)$

(1) $s_p(X) \leftarrow s1(q_1,X,[\,])$

$s1(q_1,[a|X],Y) \leftarrow s1(q_1,X,[a|Y])$ (for all a in $K \cup \{\not\epsilon\}$)

$s1(q_1,[\not\epsilon|X],[\not\epsilon|Y]) \leftarrow s1(q_f,X,Y)$ (for all q_f in F)

$s1(q_0,[\,],[\,])$

(2) $s1(q,[\tilde{w}^R|X],[a|Y]) \leftarrow s1(p,X,Y)$ (for all $d(p,a)=(q,w)$)

where $A=(Q,K,D,d,q_0,F)$ is a dgsm.

Let f_T be defined by $f_T(a)=\tilde{a}$(for all a in T). Then, it is easily seen that

$m_T(X) \leftarrow$ append(Y,Z,X), (f_T)-reverse(Y,Z) --- (F_1).

Further, letting f_p be a mapping defined by $f_p(x)= f(x)$, where f is a dgsm mapping induced by A, then we have

$s_p(X) \leftarrow$ append(Y,Z,X), (f_p)-reverse(Y,Z) --- (F_2).

Recall the representation formula (3-3) of Theorem 3.3 in which a logic program can be expressed by

$p(X) \leftarrow d_0(X,Y), lin_p(Y)$

where

(2) $lin_P(X)$ <- $lin(p_1,X,[])$

for each rule S->uSv in P_L of G_L,

$lin(p_1,[u|X],Y)$ <- $lin(p_1,X,[u|Y])$

$lin(p_1,[w|X],Y)$ <- $lin(p_2,X,Y)$ (S->w in P_L)

$lin(p_2,[],[])$

$lin(p_2,[v|X],[u|Y])$ <- $lin(p_2,X,Y)$.

$G_L=(\{S\},T',P_L,S)$, $P_L=\{S->u_1Sv_1,\ldots,S->u_nSv_n,S->w\}$.

Let f be defined as follows:

$f(u) = v^R$ for all S->uSv in P_L

$f(uu') = f(u)f(u')$ for all u,u' in $\{u_1,\ldots,u_n\}^*$

Here we claim that

$lin_P(x)$ succeeds iff append(y,wz,x) and (f)-reverse(y,z) succeed for some y,z.

Since $L(\mathbf{LIN}_P,T'\cup\bar{T}') = L(G_L)$, which is proved in the proof of Theorem 3.3, for the purpose of verifying the claim it suffices to show that x is in $L(G_L)$ iff append(y,wz,x) and (f)-reverse(y,z) succeed for some y,z in T'^*. For any x in $L(G_L)$, there is a sequence of rules $r_1,\ldots,r_k,r_0$ such that

$x = u_{i1}\cdots u_{ik}wv_{ik}\cdots v_{i1}$, r_j: S -> $u_{ij}Sv_{ij}$ $(1\leq j\leq k)$

and r_0 : S -> w.

Hence, let $x=x_1wx_2$, where $x_1=u_{i1}\cdots u_{ik}$, $x_2=v_{ik}\cdots v_{i1}$, then we have that append(x_1,wx_2,x) succeeds and (f)-reverse (x_1,x_2) is invoked. By the definition of f, $f(x_1)=f(u_{i1}\cdots u_{ik}) = v_{i1}{}^R\cdots v_{ik}{}^R$ $=(v_{ik}\cdots v_{i1})^R=x_2{}^R$. Since ($f_2$)-reverse($x_1$,$x_2$) succeeds iff reverse

$(f(x_1),x_2)$ succeeds, we have that (f)-reverse(x_1,x_2) succeeds. The converse relation is proved in a similar manner. Thus, we eventually have

$$lin_p(X) \leftarrow append(Y,[w|Z],X),\ (f)\text{-}reverse(Y,Z) \quad ---\quad (F_3)$$

It should be noted that for a homomorphism h, if one define a mapping f_h by $f_h(x) = h(x)^R$, then (f_h)-reverse(x,y) succeeds iff h(x)=y. Hence, a weak identity program I_T dominated by $i_T(X,Y)$ and involved in all the representation results is expressed by

$$i_T(X,Y) \leftarrow (f_h)\text{-}reverse(Y,X) \quad ---\quad (F_4)$$

Summarizing our argument on the use of extended reverse programs for expressing various types of basic elements in the representation results, from $(F_1),(F_2),(F_4)$ and (3-1) we obtain another representation theorem for logic programs.

Theorem 4.1(Representation Theorem 5)

Let T be a fixed alphabet. Then, there exist mappings f_h, f_T with the property that for any logic program P over T with a goal p(X) one can find an equivalent logic program P' with a goal p'(X) such that it is expressed by

$$p'(X) \leftarrow (f_h)\text{-}reverse(Y,X), append(Z_1,Z_2,Y), (f_T)\text{-}reverse(Z_1,Z_2),$$
$$append(W_1,W_2,Y),\ (f_p)\text{-}reverse(W_1,W_2),$$

for some mapping f_p.

5. Concluding Remarks

Through the formal language theoretic formulation, we have shown several representation theorems for logic programs. First, we introduced the concept of the success language of a logic program, and associating a logic program with its success language we gave a formal language theoretic semantics of logic programs.

Further, using the language theoretic semantics several representation theorems for logic programs were provided in which some types of fixed logic programs called generator programs play central roles in the representation.

Then, it has been considered the problem of what operation is primitive for the representation of logic programs. It was shown that the filtering function by a homomorphism and the merging function are sufficiently primitive in the sense that for any logic program one can find an equivalent logic program which is expressed within the use of combination of these two programs.

Finally, by introducing the concept of an extended reverse predicate, it has been proved that one need only "append" and "extended reverse" functions in representing logic programs.

For the future research in this direction, using a model-theoretic semantics in terms of the success language one may discuss many issues on the properties of a logic program such as program transformation, program classification, program synthesis, and so forth, some of those which we are about to work on.

ACKNOWLEDGEMENTS

The author is indebted to Dr.Tosio Kitagawa, the president of IIAS-SIS, Dr.Hajime Enomoto, the director of IIAS-SIS, for their useful suggestion and warm encouragement.

He is also grateful to his colleagues, Toshiro Minami,Taishin Nishida who worked through an earlier draft of the paper and suggested the present improved formulation.

This is a part of the work in the major R&D of the Fifth Generation Computer Project, conducted under program set up by MITI.

REFERENCES

[Co 70]Colmeraurer, A., Les systemes-Q ou un formalisme pour analyser et synthetiser des phrases sur ordinateur, Internal publication no.43, Dept. d'Informatique, Universite de Montreal, Canada, September, 1970.

[CS 62]Chomsky,N. and Schutzenberger,M.P., The algebraic Theory of context-free languages, in "Computer Programming and Formal Systems(Braffort and Hirschberg, eds.), North-Holland, Amsterdam, 118-161, 1962.

[EK 76]van Emden,M.H. and Kowalski,R.A., The Semantics of Predicate Logic as a Programming Lanquage, JACM 23: 733- 742 (1976).

[ER 80]Engelfriet,J.and Rozenberg,G., Fixed Point Languages, Equality Languages, and Representation of Recursively

Enumerable Languages, JACM 27: 499-518 (1980).

[Gr 73]Greibach,S.A.,The Hardest Context-free Language, SIAM J. Computing 2: 304-310 (1973).

[Ha 78]Harrison,M.A., Introduction to Formal Language Theory, Addison-Wesley, 1978.

[HOY 85]Hirose,S.,Okawa,S. and Yoneda,M., A Homomorphic CharacterizationofRecursivelyEnumerableLanguages, Theoretical Computer Science 35, 261-269 (1985).

[HU 79]Hopcroft,J.E.and Ullman,J.D., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.

[Ko 74]Kowalski,R., Predicate logic as a programming language, In Proc. of IFIP-74, 569-574.

[Sa 73]Salomaa, A., Formal Languages, Academic Press, 1973.

[Sh84]Shapiro,E.Y.,AlternationandtheComputational Complexity ofLogicPrograms, J. LogicProgramming 1: 19-33 (1984).

[Tä77]Tärnlund,S.A.,Horn ClauseComputability,BIT 17: 215 -226(1977).

[Yo 85] Yokomori, T., A Logic Program Schema and Its Applications, to appear in "Proc. of 9th IJCAI", UCLA, CA, Aug. 1985.

APPENDIX [The proof of Lemma 3.4]

We show the following: for any simple deterministic grammar G, there exist a simple deterministic grammar G_0, a coding f and a homomorphism h such that $L(G) = f(L(G_0))$ and $L(G_0) = h^{-1}(\not{c}D_2)$. This immediately completes the proof.

Let $G = (N, T, P, S_0)$ be a simple deterministic grammar such that $L = L(G)$, where $N = \{A_1(=S_0),\ldots,A_n\}$. We may assume that S_0

does not appear in the right-hand side of any rule in P.

Construct a simple deterministic grammar $G_0 = (N,T',P',S_0)$ as follows : $T' = \{[A,a] \mid A \to ax \text{ in } P\}$, $P' = \{A \to [A,a]x \mid A \to ax \text{ in } P\}$. Define f by

$f([A,a]) = a$ for $[A,a]$ in T'.

Then, it is obvious that G_0 is simple deterministic and $L(G) = f(L(G_0))$ holds.

Now, since G is simple deterministic, one can define a homomorphism h from T'^* into $\{a_1,a_2,\tilde{a}_1,\tilde{a}_2,\not{\phi}\}^*$ by

$h([A_i,a]) = \tilde{a}_1\tilde{a}_2^{i}\tilde{a}_1a_1a_2^{j_m}a_1 \cdots a_1a_2^{j_1}a_1$,

if $A_i \to aA_{j1} \cdots A_{jm}$ in P and $i \neq 1$,

$h([A_1,a]) = \not{\phi}a_1a_2^{j_m}a_1 \cdots a_1a2^{j_1}a_1$,

if $A_1 \to aA_{j1} \cdots A_{jm}$ in P,

$h([A_i,a]) = \tilde{a}_1\tilde{a}_2^{i}\tilde{a}_1$ if $A_i \to a$ in P and $i \neq 1$,

$h([A_1,a]) = \not{\phi}$ if $A_1 \to a$ in P.

It suffices to show that $L(G_o) = h^{-1}(\not{\phi}D_2)$ holds.

We claim the following: for $b_1,\ldots,b_k$ in T', $A_{i1},\ldots,A_{ir}$ in $N-\{A_1\}$, we have

$A_1 \overset{k}{\underset{L}{\Longrightarrow}} b_1 \cdots b_kA_{i1} \cdots A_{ir}$ $(r \geq 0)$ in G_o iff

(1) $h(b_1 \cdots b_k) = \not{\phi}y_1 \cdots y_k$ is a prefix of a word in $\not{\phi}D_2$,

and

(2) $red(\not{\phi}y_1 \cdots y_k) = \not{\phi}a_1a_2^{i_r}a_1 \cdots a_1a_2^{i_1}a_1$,

where "red" is a mapping defined by

$red(e) = e$,

$red(\not{\phi}) = \not{\phi}$,

for $i=1,2$

$red(xa_i) = red(x)a_i$,

$red(x\tilde{a}_i)=red(x)\tilde{a}_i$ if $red(x)$ not in $\{a_1,a_2,\tilde{a}_1,\tilde{a}_2\}^*\{a_i\}$,

$red(x\tilde{a}_i)=x'$ if $red(x)=x'a_i$.

(Note that $\Rightarrow_L^k$ indicates the k step left-most derivation, i.e., k consecutive rewriting steps in which the left-most nonterminal is always rewritten, and it is well-known that any word generated by a simple deterministic grammar has the unique left-most derivation for it. Further, from the property of a simple deterministic grammar, the length of a word generated exactly equals to the number of derivation steps used. A mapping image red(w), the reduced word, is the final resultant obtained by repeatedly cancelling all pairs $a_i\tilde{a}_i$.)

It should be noted that the claim suffices to prove the lemma. We shall prove the claim by induction on the length of derivation steps.

[k = 1] Suppose that $A_1 \Rightarrow b_1$ or $A_1 \Rightarrow b_1A_{i1}\cdots A_{ir}$. There exists $A_1 \to b_1$ or $A_1 \to b_1A_{i1}\cdots A_{ir}$ in P'. Then, $h(b_1) = \phi$ or $h(b_1) = \phi a_1a_2^{ir}a_1 \cdots a_1a_2^{i1}a_1$. Clearly condition (2) holds for either case. Conversely assuming (1) and (2) for k=1 gives us that $h(b_1) = \phi y_1$ is a prefix of a word in ϕD_2 and $red(\phi y_1) = \phi a_1a_2^{ir}a_1 \cdots a_1a_2^{i1}a_1$. From the way of constructing h, if $red(\phi y_1) = \phi$ (r=0), i.e., y_1 is in D_2, then we have $A_1 \to b_1$ is in P', leading to $A_1 \Rightarrow b_1$. Otherwise, $h(b_1) = \phi y_1 = \phi a_1a_2^{ir}a_1 \cdots a_1a_2^{i1}a_1$ implies that $A_1 \to b_1A_{i1}\cdots A_{ir}$ is in P'. This verifies the case k=1.

[Induction step] Suppose that $A_1 \Rightarrow_L^k b_1\cdots b_kA_{i1}\cdots A_{ir}$ ($r\geq 1$) and $A_{i1} \to b_{k+1}A_{j1}\cdots A_{jm}$ ($m\geq 0$) is used at the (k+1)-th step. Let $h(b_{k+1})=y_{k+1}$. By the induction hypothesis,

$red(\phi y_1 \cdots y_k) = \phi a_1 a_2^{ir} a_1 \cdots a_1 a_2^{i1} a_1$. Then, we have

$$red(h(b_1 \cdots b_{k+1})) = red(\phi y_1 \cdots y_{k+1})$$
$$= \phi a_1 a_2^{ir} a_1 \cdots a_1 a_2^{i2} a_1 a_1 a_2^{jm} a_1 \cdots a_1 a_2^{j1} a_1.$$

(Note that $y_{k+1} = \tilde{a}_1 \tilde{a}_2^{i1} \tilde{a}_1 a_1 a_2^{jm} a_1 \cdots a_1 a_2^{j1} a_1$.)

This also implies that $h(b_1 \cdots b_{k+1})$ is a prefix of a word in ϕD_2. Since $A_1 \Rightarrow_L^{k+1} b_1 \cdots b_{k+1} A_{j1} \cdots A_{jm} A_{i2} \cdots A_{ir}$, the 'only if' part of the proof is proved.

Conversely, suppose that we have $h(b_1 \cdots b_{k+1}) = \phi y_1 \cdots y_{k+1}$ is a prefix of a word in ϕD_2 and $red(\phi y_1 \cdots y_{k+1}) = \phi a_1 a_2^{ir} a_1 \cdots a_1 a_2^{i1} a_1$. From the construction of h, we have a partition:

$red(\phi y_1 \cdots y_k) = \phi a_1 a_2^{ir} a_1 \cdots a_1 a_2^{ip} a_1$,

$red(y_{k+1}) = h(b_{k+1}) = \tilde{a}_1 \tilde{a}_2^{t} \tilde{a}_1 a_1 a_2^{is} a_1 \cdots a_1 a_2^{i1} a_1$, where

there exists $A_t \rightarrow b_{k+1} A_{i1} \cdots A_{is}$ in P'.

But, since $red(\phi y_1 \cdots y_{k+1})$ is a word of the form $\phi a_1 a_2^{ir} a_1 \cdots a_1 a_2^{i1} a_1$ there must be some cancellation between the two, which implies that ip = t. By the induction hypothesis,

$$A_1 \Rightarrow_L^{k} b_1 \cdots b_k A_t \cdots A_{ir},$$

and applying $A_t \rightarrow b_{k+1} A_{i1} \cdots A_{is}$, we have

$$A_1 \Rightarrow_L^{k+1} b_1 \cdots b_{k+1} A_{i1} \cdots A_{is} \cdots A_{ir}.$$

This completes the proof. □