九州大学学術情報リポジトリ
Kyushu University Institutional Repository

Explanation-Based Reuse of Prolog Programs

Koga, Yasuyuki
Research Institute of Fundamental Information Science, Kyushu University

Hirowatari, Eiju
Research Institute of Fundamental Information Science, Kyushu University

Arikawa, Setsuo
Research Institute of Fundamental Information Science, Kyushu University

http://hdl.handle.net/2324/3073

出版情報：RIFIS Technical Report. 86, 1994-04. Research Institute of Fundamental Information Science, Kyushu University
バージョン：
権利関係：

KYUSHU UNIVERSITY

Explanation-Based Reuse of Prolog Programs*

Yasuyuki Koga, Eiju Hirowatari ** and Setsuo Arikawa

Research Institute of Fundamental Information Science,
Kyushu University 33, Fukuoka 812, Japan
e-mail: {koga, eiju, arikawa}@rifis.kyushu-u.ac.jp

Abstract. This paper presents a method of extracting subprograms from background knowledge. Most studies on learning logic programs so far developed are mainly concerned with pure Prolog, so that we can not deal with programs with system predicates such as the cut symbol, *true*, *false*, and so on. Explanation-based generalization system builds an explanation and learns a concept definition as its generalization, provided an input program. However, it assumes the input program be pure Prolog program. This paper proposes explanation-based reuse (EBR, for short), which is an extension of the explanation-based generalization and a method of program reuse. In EBR, we can deal with some system predicates. In extracting subprograms, we need to extract correct subprograms based on not only an explanation but also the whole background knowledge for a goal concept. This paper also shows that such extracted subprograms by EBR are correct.

1 Introduction

In learning Prolog programs, background knowledge is frequently given before an input is given. It is not efficient to deal with the whole background knowledge in order to solve specific problems. In general, if a problem is solved by the whole Prolog program, then so is by its smaller subprogram. Hence, this paper investigates how to extract such subprograms. Then, we can regard extracting a subprogram from a Prolog program as a kind of machine learning from background knowledge.

As to such machine learning from background knowledge, the explanation-based generalization (EBG, for short) has been studied by many researchers from various viewpoints [1, 2, 4, 5, 6, 8, 10, 11, 13, 14]. EBG enables a learner to reformulate, operationalize or deduce what the learner already knows implicitly [3, 9]. EBG takes as inputs a domain theory, a training example, a goal concept and an operationality criterion. EBG has two stages, explanation and generalization. In the explanation stage, EBG constructs an explanation in terms of the domain theory that proves how the training example satisfies the goal concept. In the generalization stage, EBG determines a set of operationally sufficient conditions for the goal concept under which the explanation holds, and returns it as an

* This work is partly supported by Grants-in-Aid for JSPS fellows and Scientific Research on Priority Areas from the Ministry of Education, Science and Culture, Japan.

** JSPS Fellowship for Japanese Junior Scientists.

output. Thus, EBG can be regarded as a method of learning from a pure Prolog program which consists of a domain theory and a training example.

This paper introduces a method of extracting a subprogram from a given Prolog program and a given goal concept The subprogram is the collection of all rules in an explanation of EBG for a goal concept. Hence, our problem is how to extract a correct subprogram for a goal concept from a Prolog program. In general, a Prolog program contains the cut symbol !. However, the cut makes it impossible for the EBG method to correctly control the pruning, because EBG assumes an input program to be a pure Prolog program. Hence, This paper proposes explanation-based reuse (EBR, for short), based on the original EBG. EBR enables the learner to extract a correct subprogram for a goal concept from a background Prolog program.

We deal with Prolog programs which contain the system predicates such as !, *true*, *fail* and so on. Furthermore, by means of EBR, we can extract a correct subprogram for the given goal concept not only from an explanation but also from the whole background knowledge.

The remainder of this paper is organized as follows. In Section 2, we define a computation for Prolog programs with the cut symbol in a mathematical way. In Section 3, we prepare some concepts on EBG which deals with pure Prolog programs. In Section 4, we discuss how to extract a correct subprogram of a standard Prolog program for a goal concept. Then, we propose EBR and show some properties of EBR. In Section 5, we realize our EBR system as a Prolog program.

2 Formulation of Prolog Computation

In this section, we define some notions on first order logic and logic programming, and formulate computations in Prolog systems in a mathematical way. See [7, 12] for detailed definitions on first order logic, logic programming and Prolog programs.

An atomic formula is called an *atom* and an atom without variables is called a *ground atom*. In this paper, we regard some system predicates, for example !, *true*, *fail* and so on, as atoms. A predicate symbol of a 0-ary system predicate is identified with the system predicate itself. A *program clause* is a clause of the form

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

where $A, B_1, \cdots, B_n$ are atoms. A is called the *head* and $B_1, \ldots, B_n$ is called the *body* of the clause. A program clause with the body is called a *rule*. A program clause without the head is called a *goal*. Then, by $|B_1, \ldots, B_n|_!$ we denote the number of occurrences of the cut symbol ! in the body $B_1, \ldots, B_n$.

A program is a finite set of program clauses in the textual order, that is, in the order of the clauses as they are in the program. A program P_1 is a subprogram of a program P_2, denoted by $P_1 \subseteq P_2$, if P_1 is a subset of P_2 and the program clauses in P_1 are listed in the textual order of P_2.

In Prolog systems, we can usually reduce the search spaces by the cut. Then, we modify SLD-trees for programs and goals, and define the search trees and the pruned branches by the cut as follows:

Definition 1. Let P be a program and $\leftarrow G$ be a goal. Then, a *search tree* for $P \cup \{\leftarrow G\}$ is a tree satisfying the following conditions:

(a) Each node of the tree is a goal.
(b) The root node is $\leftarrow G$.
(c) Let $\leftarrow A_1, \ldots, A_k (k \geq 1)$ be a node in the tree. Suppose that the clauses

$$B_1 \leftarrow C_1^1, \ldots, C_1^{m_1},$$
$$\vdots$$
$$B_n \leftarrow C_n^1, \ldots, C_n^{m_n},$$

are input clauses in the textual order of P, where B_i and A_1 are unifiable with a most general unifier θ_i for each i $(1 \leq i \leq n)$. Then, the node has children

$$\leftarrow (C_1^1, \ldots, C_1^{m_1}, A_2, \ldots, A_k)\theta_1,$$
$$\vdots$$
$$\leftarrow (C_n^1, \ldots, C_n^{m_n}, A_2, \ldots, A_k)\theta_n,$$

in this order.
(d) Each node with the empty clause has no children.
(e) Let $\leftarrow A_1, \ldots, A_k (k \geq 1)$ be a node in the tree. If A_1 is a system predicate and a goal $\leftarrow A_1$ can be evaluated with a substitution θ, then the node has a child

$$\leftarrow (A_2, \ldots, A_k)\theta.$$

Let P be a program, $\leftarrow G$ be a goal, and T be a search tree for $P \cup \{\leftarrow G\}$. In order to simulate the cut effect, we assign a sequence of natural numbers to each goal in a search tree for $P \cup \{\leftarrow G\}$ in the following way:

(a) Assign a sequence (0) to the root goal of T.
(b) If a sequence $(n_1, \ldots, n_m)$ $(m \geq 1)$ is assigned to a goal $\leftarrow A_1, \ldots, A_k$ $(k \geq 1)$ in T, then assign a sequence $(n_1, \ldots, n_m, s)$ to a resolvent $\leftarrow (C_1, \ldots, C_l, A_2, \ldots, A_k)\theta$, where $B \leftarrow C_1, \ldots, C_l$ is an input clause such that A_1 and B are unifiable with a most general unifier θ, and $|C_1, \ldots, C_l|_! = s$.
(c) If a sequence $(n_1, \ldots, n_m)$ $(m \geq 1)$ is assigned to a goal $\leftarrow A_1, \ldots, A_k$ $(k \geq 1)$ in T and A_1 is the cut, then assign a sequence $(n_1, \ldots, n_i - 1, \ldots, n_m)$ to a resolvent $\leftarrow A_2, \ldots, A_k$, where n_i is the rightmost positive number in $(n_1, \ldots, n_m)$.

Let us call the goal which caused the clause containing a cut to be activated a *parent goal* of the cut. That is, the selected atom in the parent matches with the head of the clause whose body contains the cut. When "selected", the cut

simply "succeeds" immediately. However, if backtracking later returns to the cut, the system discontinues the searching in the subtree which has the parent goal at the root. The cut thus causes the remainder of that subtree to be pruned from the search tree.

A path from the root node to a leaf in the tree is called a *branch*. Then we propose the search tree with the sequence of natural number assigned to each goal for the cut effect.

Lemma 2. *Let P be a program, $\leftarrow G$ be a goal, and T be a search tree for $P \cup \{\leftarrow G\}$. Suppose that a sequence $(n_1, \ldots, n_m)$ is assigned to a goal $\leftarrow A_1, \ldots, A_k$ $(k \geq 1)$ in the branch of T, where n_i is the rightmost positive number in $(n_1, \ldots, n_m)$. If A_1 is a cut, then the parent goal of the cut is the goal in this branch to which $(n_1, \ldots, n_{i-1})$ is assigned.*

Let $\leftarrow G_0$ be a goal whose leftmost atom is a cut. Suppose that $\leftarrow G_0$ occurs in a branch B. By Lemma 1, there exists the parent goal $\leftarrow G_0'$ of the cut in B. Then, for all branches between $\leftarrow G_0$ and $\leftarrow G_0'$, the branches which occur in the right side of B are pruned. In general, the Prolog systems do not realize such execution. The pruned branches are called *senseless branches*. Except for senseless branches, a branch is a *success branch* if its leaf is an empty clause. Furthermore, a branch is a *fail branch* if its leaf is a non-empty clause, and a branch is an *infinite branch* if it has not a leaf. By the above definitions, the *computation* for $P \cup \{\leftarrow G\}$ is a process that searches for success branches by the depth-first search to the search tree for $P \cup \{\leftarrow G\}$. The computation for $P \cup \{\leftarrow G\}$ *succeeds* when the success branches is found in the computation $P \cup \{\leftarrow G\}$.

When the the search tree for $P \cup \{\leftarrow G\}$ has success branches and no infinite branches, G is called a *query* for P. A substitution given by the computation for $P \cup \{\leftarrow G\}$ is called an *answer* for G and P. By $(P, G) \vdash_{\mathrm{PC}} G\theta$, we denote that the computation for $P \cup \{\leftarrow G\}$ succeeds and the substitution θ is an answer for G and P.

3 Explanation-Based Generalization

First we recall some concepts on EBG according to [5]. In EBG which deals with programs, it is assumed that an input program is a pure Prolog program, that is, a finite set of definite clauses listed in the textual order.

A *domain theory* is a finite set of program clauses, denoted by D. A *training example* is a non-empty finite set of ground atoms, denoted by T. Both D and T are sets of program clauses. In this paper, a set $D \cup T$, denoted by P, is called a *program*. A *goal concept* is an atom G of the target for learning. Let Π_P be the set of all predicate symbols in P. Then, an *operationality criterion* is a subset O of Π_P. Let $\Pi(O)$ be the set of all atoms that have predicate symbols in O. In this paper, we identify an operationality criterion O with $\Pi(O)$. A triplet (P, G, O) is called an *input*, denoted by I, of EBG.

Definition 3. An *explanation tree* for an input $I = (P, G, O)$ is a finite tree satisfying the following conditions:

(a) each node of the tree is an element of M_P, where M_P is the least Herbrand model of P.
(b) the root node is an instance of G.
(c) if a node α in the tree has children $\beta_1, \cdots, \beta_n$ $(n \geq 1)$ in this order, then $\alpha \leftarrow \beta_1, \cdots, \beta_n$ is a ground instance of a rule in P, that is, we get child nodes by applying rule in P to a parent node.
(d) each node in an element of $\Pi(O)$ has no children.

We assign a number to each rule in the explanation tree in the following way:

(1) Assign a number to each node that is not an element of $\Pi(O)$ in depth-first order starting from 1.
(2) Let α be a node which has children $\beta_1, \ldots, \beta_n$ $(n \geq 1)$ in this order, and to which number j assigned. Then, assign the number j to a rule in P, denoted by C_j, which is a generalization of $\alpha \leftarrow \beta_1, \ldots, \beta_n$.

We use the rules with numbers in the following generalized derivation.

Definition 4. A *generalized derivation* for $I = (P, G, O)$ is an SLD-derivation via CR which consists of a finite sequence $\leftarrow G_0 (= \leftarrow G), \leftarrow G_1, \ldots, \leftarrow G_n$ of goals, a sequence $C'_1, \ldots, C'_n$ of variants of rules in P, and a sequence $\theta_1, \ldots, \theta_n$ of most general unifiers, where CR is a computation rule to select the leftmost atom not included in $\Pi(O)$.

For the rules $C_1, \ldots, C_n$ in the explanation tree, each C'_i $(1 \leq i \leq n)$ is a suitable variant of C_i in the sense that C'_i does not have any variables which already appear in the derivation up to G_{i-1}.

A generalized derivation depends on an explanation tree. An *explanation* is a construction of an explanation tree, and a *generalization* is a regression to a goal concept through the explanation tree using a generalized derivation.

Definition 5. *EBG* is a procedure that derives a *general definition* R of G from its input $I = (P, G, O)$ by an explanation and a generalization, and it is denoted by

$$I \xrightarrow{EBG} R,$$

where

$$R = \begin{cases} G & \text{if } n = 0 \,, \\ G\theta_1 \cdots \theta_n \leftarrow G_n & \text{otherwise} \,, \end{cases}$$

and $(\leftarrow G_n)$ is the goal and $(\theta_1, \ldots, \theta_n)$ is the sequence of most general unifiers in the generalized derivation of depth n.

Usually, an atom α in P is identified with a rule $\alpha \leftarrow true$ by regarding the system predicate *true* as an atom. Thus, in Theorem 1, we assume that an program contains the true symbol and an operationality criterion is $\{true\}$.

Theorem 6. *Let $I = (P, G, O)$ be an input such that the search tree for $P \cup \{\leftarrow G\}$ has no infinite branch. Let P' be a subprogram of P for G which consists of all input clauses in the explanation for I. Then, for any substitution θ,*

$$(P', G) \vdash_{\mathrm{PC}} G\theta \textit{ iff } (P, G) \vdash_{\mathrm{PC}} G\theta.$$

Proof. ($\Rightarrow$) Suppose $(P', G) \vdash_{\mathrm{PC}} G\theta$ for a substitution θ. For any subprogram P'' of P, each answer for G and P'' is an answer for G and P, because the search tree for $P \cup \{\leftarrow G\}$ has no infinite branch. Then, we can compute all answers for G and P' by using P instead of P'. Hence, $(P, G) \vdash_{\mathrm{PC}} G\theta$.

($\Leftarrow$) Suppose $(P, G) \vdash_{\mathrm{PC}} G\theta$ for a substitution θ. Then, we can compute all answers for G and P, because the search tree for $P \cup \{\leftarrow G\}$ has no infinite branch. Thus, we can construct the explanation tree for I. Then, P' includes enough clauses in P to compute all answers for G and P. Hence, $(P', G) \vdash_{\mathrm{PC}} G\theta$. □

Suppose that $I = (P, G, O)$ is an input and P' is the subprogram of P satisfying the condition in Theorem 1. Since P is identified with a pure Prolog, P' is the least subprogram of P to compute all answers for G and P. Hence, we should construct such a program P' obtained from the explanation for I, and use P' instead of P in order to compute an answer for G and P.

4 Explanation-Based Reuse

In this section, we investigate the method of extracting subprograms. In EBG, it is assumed that an input program is a pure Prolog program. However, this assumption is not adequate for knowledge acquisition from the standard Prolog programs. In this section, we regard an input program as a standard Prolog program which may contain the cut symbol and some other system predicates.

For a program P and a goal concept G, when we construct a subprogram P' of P we require that the answer for G and P' should be identical to the answer for G and P. From now on, we assume that G is a query for P.

Definition 7. Let P be a program, G be a goal concept, and P' be a subprogram of P. Then, P' is an *extracted program* of P for G if, for any substitution θ,

$$(P', G) \vdash_{\mathrm{PC}} G\theta \iff (P, G) \vdash_{\mathrm{PC}} G\theta.$$

An input is a triplet $I = (P, G, O)$ similar to EBG, where O is the set of all predicate symbol of system predicates. Then, we can construct a subprogram of P, which consists of all input clauses in the explanation for I.

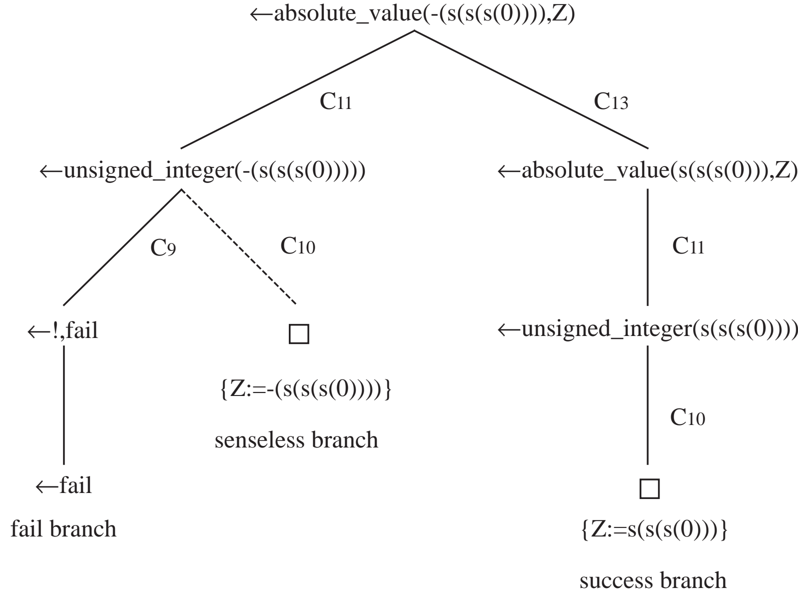

Fig. 1. Search tree for $P \cup \{\leftarrow G\}$ in Example 1.

Example 1. Let P be the following program:

$$P = \left\{ \begin{array}{l} C_1 : number_value(0, 0, _) :\!-\ !. \\ C_2 : number_value(X, Y, Z) :\!-\ i_number(X), !, \\ \qquad\quad absolute_value(X, Y), sign(X, Z). \\ C_3 : i_number(+(X)) :\!-\ i_number(X). \\ C_4 : i_number(-(X)) :\!-\ i_number(X). \\ C_5 : i_number(X) :\!-\ n_number(X). \\ C_6 : n_number(0). \\ C_7 : n_number(s(X)) :\!-\ n_number(X). \\ C_8 : unsigned_integer(+(X)) :\!-\ !, fail. \\ C_9 : unsigned_integer(-(X)) :\!-\ !, fail. \\ C_{10} : unsigned_integer(X). \\ C_{11} : absolute_value(X, X) :\!-\ unsigned_integer(X). \\ C_{12} : absolute_value(+(X), Y) :\!-\ absolute_value(X, Y). \\ C_{13} : absolute_value(-(X), Y) :\!-\ absolute_value(X, Y). \\ C_{14} : sign(X, +) :\!-\ unsigned_integer(X), !. \\ C_{15} : sign(+(X), Y) :\!-\ sign(X, Y). \\ C_{16} : sign(-(X), Y) :\!-\ sign(X, Z), change(Y, Z). \\ C_{17} : change(+, -). \\ C_{18} : change(-, +). \end{array} \right\},$$

and $G = absolute_value(-(s(s(s(0)))), Z)$ be a goal concept. Fig. 1 illustrates

the search tree for $P \cup \{\leftarrow G\}$.

Then, the following program is obtained from the explanation for I.

$$P_0 = \left\{ \begin{array}{l} C_{10} : unsigned_integer(X). \\ C_{11} : absolute_value(X,X) :- unsigned_integer(X). \\ C_{13} : absolute_value(-(X),Y) :- absolute_value(X,Y). \end{array} \right\} .$$

Fig. 2 illustrates the search tree for $P_0 \cup \{\leftarrow G\}$. This program is not an extracted program of P for G, because

$$(P_0, G) \vdash_{\mathrm{PC}} absolute_value(-(s(s(s(0)))), -(s(s(s(0))))),$$

but

$$(P, G) \not\vdash_{\mathrm{PC}} absolute_value(-(s(s(s(0)))), -(s(s(s(0))))).$$

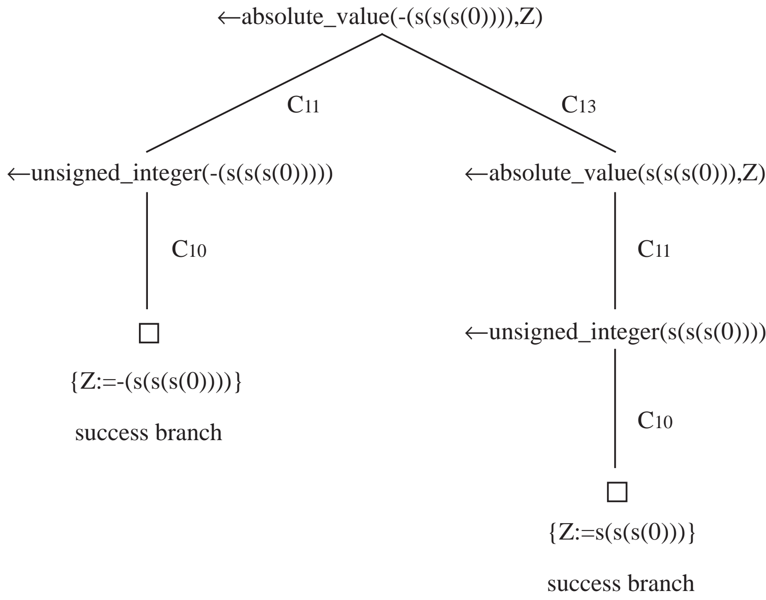

Fig. 2. Search tree for $P_0 \cup \{\leftarrow G\}$ in Example 1.

The above example means that we can not extract subprograms correctly only by the explanation, if an input program contains the cut symbol. Thus, we should pay attention to not only explanations, i.e., success branches, but also fail branches in order to obtain extracted programs of a program for a goal concept. Then, we propose explanation-based reuse as a method of constructing subprograms.

Definition 8. Let P be a program and G be a goal concept. Then, *explanation-based reuse of P satisfying G* (EBR of P satisfying G, for short) is a method of constructing subprograms, which is the collection of all input clauses in all success branches and fail branches in a search tree for $P \cup \{\leftarrow G\}$.

Example 2. Let P be the program and G be the goal concept in Example 1. Then, the following program is obtained by EBR of P satisfying G.

$$P_G = \left\{ \begin{array}{l} C_9 : unsigned_integer(-(X)) :\!-\ !, fail. \\ C_{10} : unsigned_integer(X). \\ C_{11} : absolute_value(X, X) :\!-\ unsigned_integer(X). \\ C_{13} : absolute_value(-(X), Y) :\!-\ absolute_value(X, Y). \end{array} \right\} .$$

This subprogram is the extracted program of P for G.

Theorem 9. *Let P be a program, G be a goal concept, and P_G be the subprogram obtained by EBR of P satisfying G. Then, for any substitution θ,*

$$(P_G, G) \vdash_{\mathrm{PC}} G\theta \ \textit{iff} \ (P, G) \vdash_{\mathrm{PC}} G\theta.$$

Proof. Let T and T_G be search trees for $P \cup \{\leftarrow G\}$ and $P_G \cup \{\leftarrow G\}$, respectively. By EBR of P satisfying G, P_G consists of all program clauses needed to construct T. Thus, T is identical to T_G. Hence, for any substitution θ, $(P_G, G) \vdash_{\mathrm{PC}} G\theta$ iff $(P, G) \vdash_{\mathrm{PC}} G\theta$. □

This theorem asserts that the subprogram obtained by EBR of P satisfying G is an extracted program of P for G. By Theorem 2, we can easily prove the following corollary:

Corollary 10. *Let P be a program, G be a goal concept, and P_G be the subprogram obtained by EBR of P satisfying G. For any subprogram P_0 of P such that $P_G \subseteq P_0 \subseteq P$ and any substitution θ,*

$$(P_0, G) \vdash_{\mathrm{PC}} G\theta \ \textit{iff} \ (P_G, G) \vdash_{\mathrm{PC}} G\theta.$$

Suppose that P is a program and G is a goal concept. Then, EBR returns the least extracted subprogram of P for G to compute all answers for G and P.

Just as we have done for one goal concept, we can naturally give the definition for many goal concepts.

Definition 11. Let P be a program, $G_1, \ldots, G_n$ be goal concepts, and P_{G_i} be the subprogram obtained by EBR of P satisfying G_i. Then, a subprogram $P_{\{G_1,\ldots,G_n\}}$ obtained by EBR of P satisfying $G_1, \ldots, G_n$ is the union of subprograms $P_{G_1}, \ldots, P_{G_n}$, where the program clauses in $P_{\{G_1,\ldots,G_n\}}$ are listed in the textual order of P. It is defined by $P_{\{G_1,\ldots,G_n\}} = \cup_i P_{G_i}$.

The following theorem asserts that we can obtain an extracted program of a program for goal concepts $G_1, \ldots, G_n$.

Theorem 12. *Let P be a program, $G_1 \dots G_n$ be goal concepts, and $P_{\{G_1,\dots,G_n\}}$ be a subprogram obtained by EBR of P satisfying $G_1, \dots, G_n$. Then, for any substitution θ,*

$$(P_{\{G_1,\dots,G_n\}}, G_i) \vdash_{\mathrm{PC}} G_i\theta \text{ iff } (P, G_i) \vdash_{\mathrm{PC}} G_i\theta \quad (1 \leq i \leq n).$$

Proof. Let P_{G_i} be the subprogram obtained by EBR of P satisfying G_i. Then, $P_{G_i} \subseteq P_{\{G_1,\dots,G_n\}} \subseteq P$. By Theorem 2 and Corollary 1, for any substitution θ, $(P_{\{G_1,\dots,G_n\}}, G_i) \vdash_{\mathrm{PC}} G_i\theta$ iff $(P, G_i) \vdash_{\mathrm{PC}} G_i\theta$. □

If a goal concept is a query, then the subprogram obtained by EBR of P satisfying G is the extracted program of P for G by Theorem 2. In general, a goal concept is not always a query. Hence, for a goal concept which is not a query, we extend EBR as follows: By presenting a sequence of queries g for P which are instances of G to the system one by one, we can eventually obtain a subprogram P_G such that the answer for g and P_G is identical to the answer for g and P. This method is called an *extended EBR* of P satisfying G. An extended EBR of P satisfying G identifies such a subprogram P_G in the limit by presenting a sequence of queries.

5 Prolog Implementation

The EBR system we have introduced takes as an input a Prolog program P and a goal concept G, and returns an extracted program by EBR of P satisfying G. As we have discussed in Section 4, the EBR system has the following three stages:

(a) Assign a number to each clause in P in the textual order starting from 1.
(b) Collect the numbers which are assigned to input clauses in all success and fail branches in the search tree for $P \cup \{\leftarrow G\}$.
(c) Extract a subprogram of P in the textual order, in which all clauses are assigned numbers in the set determined in the stage (b).

In the stage (a), the system replaces the i-th clause $A \leftarrow B_1, \dots B_n$ ($n \geq 0$) from the top of a input Prolog program with a clause

$$A \leftarrow counter(i), B_1, \dots B_n$$

for each i, where *counter* is a predicate symbol which does not appear in the input program. Then, the system stores it in a Prolog database and uses it as background knowledge in the stage (b).

The stage (b) is a main part of the EBR system. To see the stage (b) more concretely, we show the main Prolog program.

```
/*Explanation-Based Reuse from Prolog Program*/

ebr(Goal,List) :- assert(count([])),Goal,fail.
ebr(Goal,List) :- !,count(X),uniq(X,List).

counter(X) :- count(L),Y=[X|L],retract(count(L)),assert(count(Y)).
```

The predicate `ebr` takes as its first argument a goal concept and returns as its second argument the set of all numbers assigned to clauses in a program extracted by the EBR system.

Consider Example 1 again. For the input Prolog program P, in the stage (a), we transform P to the following program.

```
number_value(0,0,_):-counter(1),!.
number_value(X,Y,Z):-counter(2),i_number(X),!,
                     absolute_value(X,Y),sign(X,Z).
i_number(+(X)):-counter(3),i_number(X).
i_number(-(X)):-counter(4),i_number(X).
i_number(X):-counter(5),n_number(X).
n_number(0):-counter(6).
n_number(s(X)):-counter(7),n_number(X).
unsigned_integer(+(X)):-counter(8),!,fail.
unsigned_integer(-(X)):-counter(9),!,fail.
unsigned_integer(X):-counter(10).
absolute_value(X,X):-counter(11),unsigned_integer(X).
absolute_value(+(X),Y):-counter(12),absolute_value(X,Y).
absolute_value(-(X),Y):-counter(13),absolute_value(X,Y).
sign(X,+):-counter(14),unsigned_integer(X),!.
sign(+(X),Y):-counter(15),sign(X,Y).
sign(-(X),Y):-counter(16),sign(X,Z),change(Y,Z).
change(+,-):-counter(17).
change(-,+):-counter(18).
```

Let $G = \texttt{absolute_value}(-(\texttt{s}(\texttt{s}(\texttt{s}(0)))), \texttt{Z})$ be a goal concept.
The question to the EBR system is

```
?- ebr(absolute_value(-(s(s(s(0)))),Z),List).
```

The answers from the EBR system is

```
Z = Z,
List = [9,10,11,13]
```

Then, the answer from the total system is

```
unsigned_integer(-(X)):-!,fail.
unsigned_integer(X).
absolute_value(X,X):-unsigned_integer(X).
absolute_value(-(X),Y):-absolute_value(X,Y).
```

References

1. Ali, K. M.: Augmenting domain theory for explanation-based generalization. In Proceedings of the Sixth International Workshop on Machine Learning (1989) 40–42
2. Boström, H.: Improving example-guided unfolding. In Proceedings of European Conference on Machine Learning (1993) 124–135

3. Ellman, T.: Explanation-based learning: A survey of programs and perspectives. ACM Computing Surveys **21** (1989) 163–221
4. Etzioni, O.: Acquiring search-control knowledge via static analysis. Artificial Intelligence **62** (1993) 255–301
5. Hirowatari, E., Arikawa, S.: Incorporating explanation-based generalization with analogical reasoning. Bulletin of Informatics and Cybernetics **26** (1994) 13–33
6. Kedar-Cabelli, S.T., McCarty, L.T.: Explanation-based generalization as resolution theorem proving. In Proceedings of the Fourth International Workshop on Machine Learning (1987) 383–389
7. Lloyd, J. W.: Foundation of logic programming (second edition). Springer-Verlag (1987)
8. Mahadevan, S., Mitchell, T.M., Mostow, J., Steinberg, L., Tadepalli, P.V.: An apprentice-based approach to knowledge acquisition. Artificial Intelligence **64** (1993) 1–52
9. Mitchell, T.M., Keller, R. M., Kedar-Cabelli, S. T.: Explanation-based generalization: a unifying view. Machine Learning **1** (1986) 47–80
10. Numao, M., Maruoka, T., Shimura, M.: Speed-up learning by extracting partial structures of explanations. Journal of Japanese Society for Artificial Intelligence (in Japanese) **7** (1992) 1018–1026
11. Puget, J.-F.: Explicit representation of concept negation. Machine Learning **14** (1994) 233–247
12. Sterling, L., Shapiro, E.: The art of Prolog. The MIT Press (1986).
13. van Harmelen, F., Bundy, A.: Explanation-based generalization = partial evaluation. Artificial Intelligence **36** (1988) 401–412
14. Yamada, S.: Computing the utility of EBL in a logic programming environment. Journal of Japanese Society for Artificial Intelligence (in Japanese) **7** (1992) 309–319