Theoretical Computer Science 34 (1984) 227-240
North-Holland

ENUMERATION OF SUCCESS PATTERNS IN LOGIC PROGRAMS

Taisuke SATO
Electrotechnical Laboratory, Sakura-Mura, Ibaraki-Ken, Japan

Hisao TAMAKI
Ibaraki University, Hitachi-Shi, Ibaraki-Ken, Japan

1. Introduction

Since the proposal of logic programming by Horn clauses [5] and Prolog [7] has been gaining popularity because of the unified treatment of declarative semantics and procedural semantics. It has been successfully applied to natural language processing [3], data base queary [4] and others. Pattern directed invocation and built-in backtracking mechanism of Prolog are suited for symbolic manipulation.

A Prolog program consists of logical formulas called Horn clauses. There are two types of Horn clauses—definite clause and goal clause. The definite clause has the form $A_0 \leftarrow A_1 \ldots A_m$ $(m \geq 0)$ where $A_1 \& \ldots \& A_m$ implies A_0. A_0 is the head and the literal sequence $A_1 \ldots A_m$ is the body. Each A_i is a goal. $m = 0$ is the unit clause, which asserts that A_0 is true. When $m > 0$, the clause works as a procedure to compute the relation A_0 by the goals $A_1 \ldots A_m$. A Prolog program, which consists of definite clauses, can be seen as an axiom. For example,

$$S_0 = \{\mathrm{add}(0, x, x) \leftarrow, \mathrm{add}(s(x), y, s(z)) \leftarrow \mathrm{add}(x, y, z)\}$$

is an axiomatic addition program where "s" is the successor function. It defines the addition-relation over the term domain (Herbrand universe). The goal clause has the form $\leftarrow A_1(x) \ldots A_m(x)$ $(m \geq 1)$, which is supplied by the user as a top goal. Here, Prolog interpreter has to find the value of x which satisfies $A_1, \ldots, A_m$ simultaneously. For example, $\leftarrow \mathrm{add}(s(0), s(s(0)), x)$ is a goal clause to compute $1+2$. The Prolog interpreter computes the value of x by invoking clauses in S_0 and returns $s(s(s(0)))$ as an answer. Procedure invocation and parameter passing are carried out by the matching pattern (unification) of a goal with a clause head. Computation by Prolog can be seen as a (refutation) proof for the program to achieve the top goal.

The Prolog program is basically nondeterministic and the interpreter (compiler) lacks the ability to detect the determinacy in a program. Hence, it always prepares for backtrackings even if careful inspection would show the determinacy of the program. Preparations for backtracks are time- and memory-consuming tasks and

cause slow computations. The detection of the determinacies in a program and the elimination of preparations for backtracks definitely save time and memory.

In this paper we will describe a method to analyze Prolog programs based on an abstract item set construction. This method gives run time instantiation patterns of the clauses in a program. It reveals:

(1) possible calling patterns for a clause,

(2) necessary conditions for successful computation of a clause.

The first information enables us to detect the possible variable values of a clause at calling time which will be very useful to a Prolog compiler or a Prolog program transformation system [6].

The second information is more important because it helps to avoid backtrackings. If an invoked clause does not meet the condition, backtracks will inevitably occur in the subsequent computation. Therefore, eliminating such an invocation will avoid backtrackings in a Prolog program and will lead to detection of the determinacy of the program. This is exemplified by a parsing program in Section 5.

Our analysis consists of the combination of term abstraction and item set construction for a Prolog program and a top goal. Term abstraction identifies two terms which will reduce an infinite set of items to a finite set from some abstract point of view. Our item set construction resembles the one in LR(k) parsing theory except that abstracted items are stored. The resulting item set for a program to achieve the top goal covers all clause instantiation patterns that may appear during computations.

First we define the term depth abstraction and the item set construction for a Prolog program S and a top goal $\leftarrow\delta$. Next we reveal the relationship of the item set for S and $\leftarrow\delta$ to the state of Prolog interpreter and/or to the proof tree for $\leftarrow\delta$ by S. Then an example shows how to detect the hidden determinacies in a nondeterministic program using the information obtained from the item set for the program. Finally, we discuss another instance of abstract item set construction called mode abstraction which enables us to give appropriate mode declarations of a program automatically. Mode declarations disambiguate the input/output roles of arguments of a predicate in a program so that they can be used by a compiler for optimization. Readers are assumed to be familiar with clausal logic [2]. In what follows, $\alpha, \beta, \ldots$ stand for a sequence of literals and $\theta, \lambda, \ldots$ for substitutions.

2. Term-depth abstraction

Term-depth abstraction converts a term t to the term s whose instantiation is t. It is applied to both literals and clauses.

Definition 1 (*level* and *subterm*).

(a) For a given term t, t has level 0. t is called a level 0 subterm of t.

(b) If the subterm $f(t_1, \ldots, t_n)$ of t has level k (≥ 0), then each t_i has level $k+1$ and is called a level $k+1$ subterm of t.

Definition 2 (*depth k abstraction*). For a given term t and an integer k, replace every level k subterm of t by a newly created variable. The resulting term, denoted by $[t]k$, is called the depth k abstraction of the term t or simply k-term of t. Obviously the original t is an instantiation of $[t]k$ for every integer k. For an expression E (term, literal, clause, item (see Definition 3)), replace every term t in the argument position of E by $[t]k$. Then the resulting expression, denoted by $[E]k$, is called the depth k abstraction of E.

Example. Let a term $t = f(g(x, a), y, b)$ and $u_1, v_1, v_2, v_3, w_1, w_2$ be new variables other than x, y. 0-term of t is u_1. 1-term of t is $f(v_1, v_2, v_3)$. 2-term of t is $f(g(w_1, w_2), y, b)$. k-term of t ($k \geq 3$) is $f(g(x, a), y, b)$ which is the same as t.

3. Item set

Here we define an item set for a program to achieve a top goal. $(E)\theta$ is the result of a substitution θ to an expression E.

Definition 3 (*item*). An item is a Horn clause with a dot in the body. An item of the form $A \leftarrow .\alpha$ is called an initial item. An item of the form $A \leftarrow \alpha.$ is called a closed item. We say an item is a k-item if every subterm occurring in the item has level at most k.

Definition 4 (*variant*). If an expression E (term, literal, clause, item) is different from an expression F only in the variable names, E is called a variant of F, and vice versa. If E and F are variants of each other, we write $E = F$ (modulo renaming). This also applies to sets of expressions.

To construct the item set for a program S and a goal $\leftarrow \delta$, first we set up the initial item set for S and $\leftarrow$

Definition 5 (*initial item set*). For a program S, an integer k and a goal $\leftarrow \delta$, an item set

$$\text{Init}(S, \delta) = \{\leftarrow .\delta\} \text{ or } \{A \leftarrow . \mid A \leftarrow \text{ is a unit clause in } S\}$$

is called the initial item set for S and $\leftarrow \delta$.

Second we add items to Init(S, δ) by taking the downward closure and the upward closure of the preceding item set alternately.

Definition 6 (*downward k-closure*). For a finite item set I, a program S and an integer k, we define the downward k-closure, denoted by D-closure(I, S, k), as the minimum item set J such that:

(1) J includes I (modulo renaming).

(2) Suppose there is an item $A \leftarrow \alpha.X\beta$ in J and a definite clause $B \leftarrow \gamma$ in S whose head B is unifiable with X. Let the mgu (most general unifier) of B and X be θ. Then a variant C of $[(B \leftarrow .\gamma)\theta]k$ is also included in J.

Definition 7 (*upward k-closure*). For a finite item set I and integer k, we define the upward k-closure, denoted by U-closure(I, k), as the minimum item set J such that:

(1) J includes I (modulo renaming).

(2) Suppose there are an item $A \leftarrow \alpha.X\beta$ and a closed item $B \leftarrow \gamma.$ in J whose head B is unifiable with X. Let the mgu of B and X be θ. Then a variant C of $[(A \leftarrow \alpha X.\beta)\theta]k$ is also included in J.

Definition 8 (*Son item* and *descendent item*). A closed item $B \leftarrow \gamma.$ referred to in Definition 7 is called a son item of C ($=[(A \leftarrow \alpha X.\beta)\theta]k$). Every closed item that has already been defined as a son item of $A \leftarrow \alpha.X\beta$ is also a son item of C. We define descendent item as the transitive closure of son item.

The existence of D-closure(I, S, k) and U-closure(I, k) can be easily verified (as to closure construction algorithm, see [1]). Since k-items are finite for a given integer k, D-closure(I, S, k) is finite (modulo renaming). Similarly, U-closure(I, k) is finite.

Starting with the initial item set $S_0 = \mathrm{Init}(S, \delta)$ for a program S, a goal $\leftarrow \delta$ and an integer k, we construct a series $S_1, S_2, \ldots$ of item sets by taking alternately the downward closure and the upward closure of the preceding item set. At some mth stage of the construction the downward closure and the upward closure of S_m is the same as S_{m-1} because the possible k-items generated from $S \cup \{\leftarrow \delta\}$ are finite.

Definition 9 (*closure*). For a program S, a goal $\leftarrow \delta$ and an integer k, define a series of item sets $I_0, I_1, \ldots$ and the item set for S and $\leftarrow \delta$, denoted by $I(S, \delta, k)$, as follows:

$$I_0 = \mathrm{Init}(S, \delta),$$

$$I_{i+1} = \text{U-closure}(\text{D-closure}(I_i, S, k), k) \quad \text{for } i \geq 0.$$

$$I(S, \delta, k) = I_0 \cup I_1 \cup I_2 \cup \cdots.$$

Proposition 10. *Let S be a program, $\leftarrow \delta$ a goal, k an integer. And let $I = I(S, \delta, k)$ be the k-item set for S and $\leftarrow \delta$; then:*

(1) *I is finite.*

(2) $I = \text{D-closure}(I, k)$.

(3) $I = \text{U-closure}(I, k)$.

Proof. (1) is obvious. Note that I is the least upper bound of an ascending chain $I_0 \leq I_1 \leq I_2 \leq \cdots$ and both D-closure and U-closure are monotonous and continuous as a function of an item set. Therefore,

$$I \leq \text{D-closure}(I, S, k) \leq \text{U-closure}(\text{D-closure}(I, S, k), k) = I.$$

(2) and (3) can be derived from these relations easily. □

Example. Let a program $S_1 = \{A(x) \leftarrow B(x)D(x),\ B(y) \leftarrow C(y),\ C(a),\ C(b),\ D(b),$ x, y: variable, a, b: constant} and let the top goal to achieve be $\leftarrow A(z)$ (z: variable). For simplicity we ignore the term depth abstraction, i.e., the term abstraction depth k is infinite.

$$I_0 = \text{Init}(S_1, A(z))$$
$$= \{\leftarrow .A(z),\ C(a) \leftarrow .C(b) \leftarrow .D(b) \leftarrow .\},$$
$$I_0' = \text{D-closure}(I_0, S, k)$$
$$= I_0 \cup \{A(x) \leftarrow .B(x)D(x),\ B(y) \leftarrow .C(y)\},$$
$$I_1 = \text{U-closure}(I_0', k)$$
$$= I_0' \cup \{B(a) \leftarrow C(a).,\ B(b) \leftarrow C(b).,\ A(a) \leftarrow B(a).D(a),\ A(b) \leftarrow B(b).D(b)\},$$
$$I_1' = \text{D-closure}(I_1, S, k) = I_1,$$
$$I_2 = \text{U-closure}(I_1', k) = I_1 \cup \{A(b) \leftarrow B(b)D(b).,\ \leftarrow A(b).\},$$
$$I_2' = \text{U-closure}(\text{D-closure}(I_2, S, k), k).$$

Therefore, the item set for S to achieve $\leftarrow A(z)$ is as follows:

$$I(S, A(z), k) =$$
$$= \{\leftarrow .A(z),\ \leftarrow A(b).,$$
$$C(a) \leftarrow .,\ C(b) \leftarrow .,\ D(b) \leftarrow .,$$
$$A(x) \leftarrow .B(x)D(x),\ A(a) \leftarrow B(a).D(a),$$
$$A(b) \leftarrow B(b).D(b),\ A(b) \leftarrow B(b)D(b).,$$
$$B(y) \leftarrow .C(y),\ B(a) \leftarrow C(a).,\ B(b) \leftarrow C(b).\}.$$

An item constructed by instantiating a clause and putting a dot in the body is called an item generated from the clause. Thus:

Items generated from $A(x) \leftarrow B(x)D(x)$
$$= \{A(x) \leftarrow .B(x)D(x),\ A(a) \leftarrow B(a).D(a),$$
$$A(b) \leftarrow B(b).D(b),\ A(b) \leftarrow B(b)D(b).\},$$
Items generated from $B(y) \leftarrow C(y)$
$$= \{B(y) \leftarrow .C(y),\ B(a) \leftarrow C(a).,\ B(b) \leftarrow C(b).\},$$

and so on.

4. Relationship between $I(S, \delta, k)$ and the computation to achieve $\leftarrow\delta$

In this section we investigate the relation $I(S, \delta, k)$ and the computation process of $\leftarrow\delta$ by Prolog interpreter. We introduce a model of Prolog interpreter. The intermediate state of the interpreter can be represented by the pair, $\langle\alpha$ (goals), θ (substitutions)$\rangle$. The actual value of the goals is $(\alpha)\theta$. For example, if the state is $\langle A(x, y)B(y, z), \{x\backslash a, z\backslash f(y)\}\rangle$, the actual goals are $A(a, y)B(y, f(y))$. In the following, $|A|$ (A may be empty) means that A is an ancestor goal for the current goals. It is usually called an A-literal in the context of resolution [2].

Suppose an initial goal is $\leftarrow\delta$ and a program is $S = \langle C_1, \ldots, C_n\rangle$. The initial state is $\langle\delta\,||, \varepsilon\rangle$ where ε is a null substitution.

The state transition can be defined in two ways:

(1) *expansion* (procedure call).

Let the current state be $\langle A\alpha, \theta\rangle$. The interpreter always attacks the left-most goal. To solve or compute the current goal $(A)\theta$, the interpreter selects a clause $B_0 \leftarrow \beta$ in S, whose head B_0 is unifiable with $(A)\theta$ (renaming is assumed implicitly). Let the mgu be λ. Then next state is $\langle\beta\,|\,B_0|\alpha, \theta * \lambda\rangle$.

(2) *truncation* (procedure return).

Let the current state be $\langle|A|\alpha, \theta\rangle$. Since $|A|$ in the left-most position means that A has already been solved, the interpreter truncates $|A|$. Then next state is $\langle\alpha, \theta\rangle$.

This interpreter model does not take consideration of unsuccessful computations, i.e., backtracks. But it suffices for showing the relation between a Prolog program and the item set.

If the state becomes $\langle||, \theta\rangle$, then the computation ends successfully. The answer substitution is θ and the goal proved is $(\delta)\theta$. The relation between $I(S, \delta, k)$ and the computation process for $\leftarrow\delta$ by an interpreter is revealed by Proposition 11.

Proposition 11. *Construct the item set $I(S, \delta, k)$ for program S to achieve $\leftarrow\delta$. Assume that clause $B_0 \leftarrow \beta_1\beta_2$ (this may be $\leftarrow\delta$ incidentally) is invoked and the state of the interpreter becomes $\langle\beta_2|\,B_0|\,\alpha, \theta\rangle$ during the computation to achieve $\leftarrow\delta$. Then:*

(1) there is an item $B_0' \leftarrow \beta_1' \cdot \beta_2'$ in $I(S, \delta, k)$ generated from $B_0 \leftarrow \beta_1\beta_2$, and

(2) There is a substitution λ for the variables in $B_0' \leftarrow \beta_1'\beta_2'$ such that $(B_0)\theta = (B_0')\lambda$ and $(\beta_2)\theta = (\beta_2')\lambda$.

The proof of this proposition is based on induction on the number of steps to the current state. (Details are omitted.)

Corollary 12. *For a program S, a goal $\leftarrow\delta$ and an integer k, construct the item set $I(S, \delta, k)$ for S and $\leftarrow\delta$. If a clause $A \leftarrow \alpha$ is called in the computation for solving $\leftarrow\delta$, then there exists an initial item $A' \leftarrow \alpha'$ from $A \leftarrow \alpha$ in $I(S, \delta, k)$ and the unified head of the clause $A \leftarrow \alpha$ is an instantiation of A'.*

Corollary 12 is useful in two ways. First, if none of the initial items from $A \leftarrow \alpha$ exist in $I(S, \delta, k)$, then $A \leftarrow \alpha$ is never called in the computation for solving $\leftarrow\delta$.

Thus it enables us to detect redundant clauses to eliminate from the program S. Second it teaches us possible calling patterns of a clause in a computation for $\leftarrow\delta$. In other words, it informs us of the possible values of variables in the head of a clause after unification. This would be of great use to a Prolog compiler or a Prolog program transformation system [6].

Now we turn our attention to the instantiation patterns of a clause in the successful computation, i.e., global success patterns of the clause. We already know by Proposition 11 that if a clause $A\leftarrow\alpha$ is called in the computation for a top goal $\leftarrow\delta$ and the subgoal α is successfully solved, then there is a closed item $A'\leftarrow\alpha'.$ in $I(S,\delta,k)$ of which instantiation $(A'\leftarrow\alpha')\lambda$ is $(A\leftarrow\alpha)\theta$ where θ is the substitution obtained up to the time when the subgoal α was solved. Therefore, the set of closed items from $A\leftarrow\alpha$ teaches us the instantiation patterns of $A\leftarrow\alpha$ at the time when its subgoal has been successfully solved, i.e., the local success patterns of $A\leftarrow\alpha$.

However, it is possible that local successes become useless because of the subsequent backtracks. To illustrate this, consider the example following Proposition 10, where program $S_1=\{A(x)\leftarrow B(x)D(x),\ B(y)\leftarrow C(y),\ C(a),\ C(b),\ D(b)\}$. The top goal $\leftarrow A(z)$ being given, $A(x)\leftarrow B(x)C(x)$ is invoked and the current goal will become $B(x)$. Then $B(x)$ calls $B(y)\leftarrow C(y)$ and the current goal will become $C(y)$. $C(y)$ can be successfully solved by calling $C(a)$ or $C(b)$ with resulting substitution $\{y\backslash a\}$ or $\{y\backslash b\}$ respectively. Therefore, the local success patterns of $B(y)\leftarrow C(y)$ for the top goal $\leftarrow A(z)$ will become $\{B(a)\leftarrow C(a),\ B(b)\leftarrow C(b)\}$ as is suggested by the closed item set $\{B(a)\leftarrow C(a).,\ B(b)\leftarrow C(b).\}$ generated from $B(y)\leftarrow C(y)$.

Suppose that the subgoal $C(y)$ is solved with substitution $\{y\backslash a\}$. In this case, the call of $B(y)\leftarrow C(y)$ returns with $\{y\backslash a\}$. This solves $B(x)$ in $A(x)\leftarrow B(x)D(x)$ with substitution $\{x\backslash a\}$ and the next goal becomes $D(x)\{x\backslash a\}=D(a)$. $D(a)$, however, cannot be solved because S_1 does not include a clause whose head is $D(a)$. At this point the backtrack starts. It will undo the successful return of $B(y)\leftarrow C(y)$. Thus, the local success of $(B(y)\leftarrow C(y))\{y\backslash a\}=B(a)\leftarrow C(a)$ will not help to achieve the top goal $\leftarrow A(z)$. $B(a)\leftarrow C(a)$ is not a global success pattern of $B(y)\leftarrow C(y)$ but an unsuccessful pattern of $B(y)\leftarrow C(y)$.

As this example shows, even if there is a closed item $A'\leftarrow\alpha'.$ generated from $A\leftarrow\alpha$ in the item set $I(S,\delta,k)$, it is not necessarily a global success pattern of $A\leftarrow\alpha$ to achieve $\leftarrow\delta$. $I(S,\delta,k)$ often includes closed items that correspond to unsuccessful computations.

To rule out closed items which correspond to unsuccessful computations as many as possible, we enumerate the closed items generated from the clauses which may have taken part in the successful computations to achieve the top goal $\leftarrow\delta$. This is done by starting with the closed item $\leftarrow\delta'.$ generated from $\leftarrow\delta$, and tracing downward the descendant closed items of $\leftarrow\delta'.$. At the same time, instantiations are propagated from parent items to their closed sons. Any global success pattern of a clause for achieving $\leftarrow\delta$ is proved to be an instance of some enumerated and instantiation-propagated item. Non-enumerated items correspond to unsuccessful computations (see Proposition 14). Enumerated items are called success items. Based on $I(S,\delta,k)$, the success item set I-suc(S,δ,k) is given by the following definition.

Definition 13 (*success item set*). For a program S, a goal $\leftarrow \delta$ and an integer k, construct the item set $I(S, \delta, k)$ for S and $\leftarrow \delta$ as follows:

(1) Let I-suc $= \{\leftarrow \delta. \mid$ a closed item from the goal clause $\leftarrow \delta\}$.

(2) If $A_0 \leftarrow A_1, \ldots, A_m.$ is in I-suc, then choose its son item $B \leftarrow \beta.$ from $I(S, \delta, k)$. Let λ be a substitution such that $A_i = (B)\lambda$ for some i.

(3) Add $(B \leftarrow \beta.)\lambda$ to I-suc.

(4) Repeat steps (2) and (3) until no closed items are added to I-suc.

(5) The resulting I-suc set is the success item set I-suc(S, δ, k) for S and $\leftarrow \delta$.

Proposition 14. *For a program S, a goal $\leftarrow \delta$ and an integer k, construct the success item set I-suc(S, δ, k) for S and $\leftarrow \delta$. If a clause $A \leftarrow \alpha$ is used in the successful computation for $\leftarrow \delta$ with an answer substitution θ, then there are a closed item $A' \leftarrow \alpha'$. generated from $A \leftarrow \alpha$ in I-suc(S, δ, k) and a substitution λ such that $(A \leftarrow \alpha)\theta = (A' \leftarrow \alpha')\lambda$ holds.*

The proof of this proposition is omitted.

Proposition 14 is the main point of this paper. Imagine that a clause $A \leftarrow \alpha$ is invoked in a computation to achieve $\leftarrow \delta$. When $A \leftarrow \alpha$ is instantiated by the unification, it is not known whether it can have a global success or not. According to Proposition 14, however, the necessary condition for global success mentioned below must be satisfied.

Necessary condition for global success. *Assume that a clause $A \leftarrow \alpha$ is invoked and instantiated to $A' \leftarrow \alpha'$ by the invocation. Then there is a success item $A'' \leftarrow \alpha''$ in I-suc(S, δ, k) generated from $A \leftarrow \alpha$ such that $A' \leftarrow \alpha'$ is an instance of $A'' \leftarrow \alpha''$.*

In other words, once the instantiation pattern of an invoked clause violates the above condition, any computation including the invocation will be cancelled by backtracks. Therefore, checking this condition at calling time avoids fruitless computations and realizes better behavior of a program. This condition also gives a chance to detect the determinacies in a program. Assume that a program S has several clauses for the relation "p", i.e., clauses whose head predicate name is "p". So there will be non-determinacies with respect to the selection of possible callees (clauses whose head predicate name is "p") when a caller of the form $p(\ldots)$ appears in the computation for a top goal $\leftarrow \delta$. But if in I-suc(S, δ, k) every success item generated from one callee and every success item generated from another callee are non-unifiable, we can conclude that there is at most one callee that has a global success. Thus, a non-unifiable check in I-suc(S, δ, k) reveals the determinacy with respect to the relation "p".

From the above discussions and from the fact that instantiation patterns in a success item set become informative in proportion to the term abstraction depth k, it is reasonable to expect that we can eliminate almost all backtracks and detect

determinacies, if any, in a program by choosing a sufficiently large k. On the other hand, since the computational cost of a success item set grows exponentially with respect to k, a large k makes the computation needed for the item set construction impossible. Deciding the term abstraction depth seems critical and should depend on the program and the top goal to achieve.

5. Success values of variables and the detection of non-determinacies in a program

In this section we will illustrate how to apply the necessary condition for global success to eliminating backtracks in Prolog programs. We take a parsing program for a small regular language as an example. It is a nondeterministic top-down parser without left recursive rules. Based on the information obtained from the success item set for the program, we show that it can be transformed to a deterministic parser.

```
%GRAMMAR-1 for the regular expressions by {a, b, emp}
%⟨exp⟩     ::=⟨term⟩ ⟨exp1⟩
%⟨exp1⟩    ::=+⟨term⟩ ⟨exp1⟩| λ
%⟨term⟩    ::=⟨factor⟩ ⟨term1⟩
%⟨term1⟩   ::=⟨factor⟩ ⟨term1⟩ | λ
%⟨factor⟩  ::=(⟨exp⟩) ⟨factor1⟩ | a ⟨factor1⟩ |
%             b⟨factor1⟩|emp ⟨factor1⟩
%⟨factor1⟩ ::=*⟨factor1⟩|λ
```

In this grammar 'λ' denotes an empty symbol. Below is a DEC-10 Prolog [7] program S_1 for GRAMMAR-1.

```
%------------ Prolog program S1 ------------------------
 %1%  exp(X1, Z1) :-term(X1, Y1), exp1(Y1, Z1).
 %2%  exp1(['+'|X2]Z2) :-term(X2, Y2), exp1(Y2, Z2).
 %3%  exp1(X3, X3).
 %4%  term(X4, Z4) :-factor(X4, Y4), term1(Y4, Z4).
 %5%  term1(X5, Z5) :-factor(X5, Y5), term1(Y5, Z5).
 %6%  term1(X6, X6).
 %7%  factor(['('|X7], Z7) :-exp(X7, [')'|Y7]), factor1(Y7, Z7).
 %8%  factor([a|X8], Y8) :-factor1(X8, Y8).
 %9%  factor([b|X9], Y9) :-factor1(X9, Y9).
%10%  factor([emp|X10], Y10) :-factor1(X10, Y10).
%11%  factor1(['*'|X11], Y11) :-factor1(X11, Y11).
%12%  factor1(X12, X12).
```

Each predicate name(exp, exp1, term, etc.) corresponds to a nonterminal symbol. A string with an upper case letter at its head (X_1, Y_1, Z_1, etc.) is a variable. Other strings are constants. "$[a|b]$" and "$[a, b, c\ldots]$" denote cons(a, b) and list(a, b, $c\ldots$) in LISP respectively. ":-" is an implication symbol.

Every predicate $p(X, Y)$ in S_1 has two arguments both of which are lists of words (terminals). $p([w_1, \ldots, w_i, w_{i+1}, \ldots, w_n], [w_{i+1}, \ldots, w_n])$ has declarative meaning as: The category $\langle p \rangle$ derives $[w_1, \ldots, w_i]$, the part of the entire sentence $[w_1, \ldots, w_n]$ and the rest of sentence to parse is $[w_{i+1}, \ldots, w_n]$.

The program S_1 may cause a number of backtracks due to ungrammatical input sentences and nondeterministic computations in S_1. For example, because $['*', a]$ is not a grammatical sentence in GRAMMAR-1, :−exp(['*', a], [ ]) invokes clauses 1-4, then clauses 7-10 and backtrack occurs. Although $[a, '+', a]$ is a grammatical sentence, a goal :−exp([a, '+', a], [ ]) causes a backtrack too. In the parsing of $[a, '+', a]$, there appears a subgoal term1(['+', a], [ ]) which calls clause 5. Then backtrack results.

We will show how to eliminate these backtracks, either by an input sentence grammatical or not, using the modified version of the necessary condition for global success which is reformulated in terms of the instantiation patterns of variables in a clause called success values of variables.

Definition 15 (*success value*). From a given program S, top goal $\leftarrow\delta$ to achieve and integer k, construct the success item set I-suc(S, δ, k). The term t is a success value of the variable v in a clause $A \leftarrow \alpha$ to achieve $\leftarrow\delta$ if v takes t as its value in some success item $A' \leftarrow \alpha'$. generated from $A \leftarrow \alpha$ in I-suc(S, δ, k).

It follows from the necessary condition for global success stated in Section 4 that if the value of a variable v in an invoked clause is not an instance of any success value of v to achieve the top goal, the invoked clause will not have a global success. Thus we can eliminate backtracks by checking success values.

In order to realize backtrack elimination using success values, first we have to select an appropriate top goal. As any parsing computation by S_1 is included in the computations to achieve a general goal :−exp(X_0, [ ]) where X_0 is a variable, we choose this as the top goal for the success item set construction. Second, we have to decide the depth of term abstraction. We choose 2 as the depth since our interest is the first element of a list in the argument position. Then we construct the item set $I(S_1, \text{exp}(X_0, [\,]), 2)$ for S_1 to achieve the top goal :−exp(X_0, [ ]) with term abstraction depth 2. This includes 201 items (the item set construction was carried out by a LISP program).

Next we construct the success item set I-suc(S_1, exp(X_0, [ ]), 2) for S_1 to achieve :−exp(X_0, [ ]) from $I(S_1, \text{exp}(X_0, [\,]), 2)$. This includes 133 items. Then we extract the success values of head variables (variables occurring in a clause head) from I-suc(S_1, exp(X_0, [ ]), 2).

In order to extract the success values for X_1 of clause 1, for example, exp(X_1, Z_1), the head of clause 1, is unified with the head patterns of the success items generated from clause 1. Since the head patterns of success items from clause 1 are {exp([a], [ ]), exp([a|X], [ ]), exp([a|X], [')'|*]), ..., exp(['(', X|Y], [')', X|Y]), exp(['(', X|Y], [')', X|Y]), exp(['(', X|Y], [ ]) (15 different patterns)}, the success

values for X_1 are $\{a, b, \text{emp}, '('\}$. Discussions so far show us that if the value of X_1 is not one of $\{a, b, \text{emp}, '('\}$, the computation containing clause 1 never succeeds globally. As the reader may notice, the set $\{a, b, \text{emp}, '('\}$ is the First(1) symbol set for the category ⟨exp⟩. In this fashion we can extract the success values for X_3, X_4, X_5, X_6, X_{12} (see Fig. 1).

What do these success values imply in practice? Let us return to the previous example. Suppose that we are going to parse $a(n)$ (illegal) sentence $['*', a]$. The first call is :−exp($['*', a]$, [ ]) with $X_1 = '*'$. But since $'*'$ is not a success value for X_1 (see Fig. 1), we can immediately return with failure without further computations that would be discarded by backtrackings. Suppose that we are going to parse a (legal) sentence $[a, '+', a]$. The call term1($['+', a]$, [ ]) occurs. There are two possible calls (nondeterminacy!), clause 5 and clause 6. The success values in Fig. 1 suggest that we should choose clause 6 because $'+'$ is a success value for X_6, not for X_5.

Note that if the first argument of some caller is not ground (contains no free variables), the detection of the determinacy of program S_1 by those success values may not work well (imagine what happens if a call term1(Y_4, [ ]) occurs where Y_4 is a free variable: both callees, clause-5 and clause-6, will pass the check by success values and be successfully invoked). Therefore, we must confirm whether the condition that the first argument of any caller is ground is satisfied or not. Fortunately, this condition is shown to hold (automatic verification of this condition concerns another embodiment of the concept of "abstract item set for logic programs" which will be discussed in Section 6). Thus, by the success values and the clause head patterns in S_1 we can conclude that S_1 is deterministic in spite of the nondeterministic appearance.

If we convert the example program to an explicitly deterministic one using the success values given in Fig. 1, it becomes an LL(1) parser. This is not accidental.

Variable	Success values
X_1	a, b, emp, $'('$
X_3	$')'$, [ ]
X_4	a, b, emp, $'('$
X_5	a, b, emp, $'('$
X_6	$'+'$, $')'$, [ ]
X_{12}	a, b, emp, $'('$, $')'$, $'+'$, [ ]

Fig. 1. Success values for head variables in S_1.

When a grammar is LL(k) and a parsing program like S_1 for the grammar is given, we can always detect the LL(k)-ness of the program by extracting success values from the success item set, and convert it to the deterministic program. Even if a grammar is not LL(k), it is evident that we can optimize a parsing program by examining success values.

We have applied similar optimization techniques to a bottom-up parser for GRAMMAR-1. The resultant program is an LR(1)-like deterministic parser as expected. But in general we cannot expect the conversion to an exact LR(k) parser because, in order to realize an LR(k) parser, the Prolog interpreter would need to invoke several clauses at a time, which corresponds to a GO TO action.

6. Automatic mode declaration via mode items

Our analysis for logic programs relies on the item set construction and term abstraction. Another kind of abstraction, instead of term depth abstraction, is expected to bring forth a new kind of item set construction. For example, looking at the arguments of a predicate from data type point of view would produce an item set which contains information about data types of the arguments. Here we will discuss briefly another embodiment of the concept of "abstract item set" which concerns automatic mode declarations for logic programs.

In logic programs, every argument in a predicate works as both input and output. For example, S_0 in Section 1 can be used as a subtract program by exchanging the roles of arguments though it is designed to be used as an addition program. Such ambiguities about input/output distinction force a compiler to generate superfluous codes to cope with them, and result in slow computations. Therefore, it is advantageous to give information about the input/output roles of the arguments in a predicate to the compiler by some mechanism, namely, mode declarations. Of course they depend on the programmer's intention as how to use use predicates and especially how input/output roles are assigned to the argument of a top goal.

We have developed a method to generate mode declarations automatically when "argument mode" is assigned to each argument in the top goal predicate. (Argument mode means the classification of instantiation states, i.e., mode abstraction of an argument term t defined by:

+ if t is instantiated to a ground term,

− if t is a variable and uninstantiated,

? if t is instantiated to an unknown term.)

This method is based on observations that by mode abstraction there can be only a finite number of argument modes for one clause and only some of them are allowed to actually occur at run time. If we can enumerate all of the run time argument modes, it is easy to give a mode declaration appropriate to a predicate

"p" by extracting the argument modes of callers of the form "$p(\ldots)$".

Therefore, the point is how to sort out exactly the actual argument modes from the possible ones when the argument modes of a top goal are given. In order to make the enumeration as accurate as possible, we construct a set of "mode items" in which each may represent a state of some invoked clause and contain its variable modes (defined similarly to term modes) at that state. A typical mode item would be

$$\langle \mathrm{add}(s(x), y, s(z)) \leftarrow .\mathrm{add}(x, y, z), [\langle x, +\rangle, \langle y, +\rangle, \langle z, -\rangle]\rangle.$$

This item stands for one of the possible variable modes of a clause $\mathrm{add}(s(x), y, s(z)) \leftarrow \mathrm{add}(x, y, z)$ just before the call of $\mathrm{add}(x, y, z)$. Since x and y are ground and z is a variable (indicated by the left component of the item), the argument modes of the predicate "add" at this call are judged to be $\mathrm{add}(+, +, -)$. This item shows that a mode item keeps the mode types of instantiated variables instead of the variable instantiations.

An algorithm to construct a set of mode items for a program and a top goal is very similar to the one for an item set construction by term depth abstraction defined in the previous sections. It starts with an initial item, say $\langle \leftarrow .\mathrm{add}(n_1, n_2, n_3), [\langle n_1, +\rangle, \langle n_2, +\rangle, n_3, -\rangle]\rangle$ where n_1, n_2 and n_3 are variables, which corresponds to the top goal. Then it takes the downward closure and the upward closure of the preceding mode item set alternately until no mode item can be added. During the closure construction process, argument modes are made to propagate by matching the argument modes of a caller predicate and a callee predicate via variable modes in these two.

A set of mode items for a program and a top goal do not only bring the mode declarations for them but also sometimes help us to detect the determinacy of the program driven by the goal. For example, in order to detect the determinacy of the parsing program S_1 in Section 5 we are required to confirm that the first argument of a predicate is always ground when it becomes a caller in a parsing process. This is checked affirmatively by constructing a set of mode items for S_1 to achieve the top goal $:-\exp(x, y)$ where x is ground and y is a variable (the construction was carried out by a tentative automatic mode declaration program written in DEC-10 Prolog).

7. Conclusion

We have proposed a method to detect the determinacies in a logic program by constructing an item set for the program to achieve the top goal. The item set construction with term depth abstraction reduces infinite clause instantiation patterns to finite. From the item set we can extract the success values of variables in a program. They enable us to detect the determinacies implicit in the program with the help of the item set generated by term mode abstraction. This is shown by a

parsing program example. We hope that our method provides one step closer to an intelligent logic program compiler and a logic program understanding system.

Acknowledgment

The authors are grateful to Mr. Tanaka, Chief of Machine Inference Section of Electrotechnical Laboratory and other members of the section for helpful discussion.

References

[1] A.V. Aho and J.D. Ullman, *Principles of Compiler Design* (Addison-Wesley, Reading, MA, 1977).
[2] C.C. Chang and R.C.T. Lee, *Symbolic Logic and Mechanical Theorem Proving* (Academic Press, New York, 1973).
[3] A. Colmerauer, *Metamorphosis Grammars*", Lecture Notes in Computer Science 63 (Springer, Berlin, 1978).
[4] H. Gallaire and J. Minker (eds.), *Logic and Data Bases* (Plenum Press, New York, 1978).
[5] R.A. Kowalski, Predicate logic as programming language, in: J.L. Rosenfeld, ed., *Proc. IFIP-74 Congress* (North-Holland, Amsterdam, 1974).
[6] H. Tamaki and T. Sato, A transformation system for logic programs which preserves equivalency, ICOT TR-018, 1983.
[7] D. Warren, L.M. Pereira and F. Pereira, User's guide to DEC system-10 Prolog, Occasional Paper 15, Dept. of AI, Edinburgh Univ., 1979.