記号処理 37－1
（1986. 6. 20）

バックトラック機能を持つ処理系におけるバックトラックの自動削除

伊藤 貴康　松山 隆司　山崎 憲一
（東北大学 工学部）

バックトラックは、非決定的な問題解決における逐次的探索法として様々な分野で利用されている。PrologやSnobolは、自動バックトラック機能を処理系の大きな特徴としている。しかし、自動バックトラックには、時として計算効率の大幅な低下を招くといった問題がある。
本稿では、入力データやプログラムから抽出された"特徴"を利用することにより、無駄なバックトラックを自動的に削除するという考え方を提案し、Prologにおける有効性を示す。我々の基本的な考え方は、
1）与えられた入力データから、データタイプやデータの長さといった特徴を抽出する
2）抽出された特徴を用いて、非決定的な選択点における選択枝の数を減らす
というものである。すなわち、特徴を用いてその選択枝が失敗につながるかどうかを調べ、無駄なバックトラックが生じないようにするというもので、いわゆるIntelligent Backtrackingとは異なる。

Automatic Elimination of Backtracking in Programming Systems with Backtracking

Takayasu ITO, Takashi MATSUYAMA and Ken-ichi YAMAZAKI

Department of Information Engineering
Tohoku University

Backtracking is a popular sequential control mechanism to solve various search problems involving nondeterminism. Automatic backtracking was introduced into Prolog and Snobol as one of the major features of their programming systems. Although the automatic backtracking is useful, it sometimes incurs serious degradation of computational efficiency in time and memory. This paper describes an idea of eliminating redundant backtrackings by using "features" extracted from input data and programs, and discusses its application to Prolog. Our basic idea is as follows. For a given input data we extract features such as its data types and quantitative properties (e.g. length of data, etc), and then we use these features to reduce the number of possibilities at many decision points in nondeterministic searching. Our method reduces backtrackings by eliminating useless possibilities leading to failures by using features extracted from input data; thus our method differs from (so-called) intelligent backtracking.

1. INTRODUCTION

Backtracking has been a popular control mechanism to solve various search problems in many fields of information processing. In the areas of symbolic processing and artificial intelligence, we often encounter problems for which we are required to write search programs based on backtracking. Thus, in these areas, there exist several programming systems with automatic backtracking, like Snobol, Planner, and Prolog. A programming system with automatic backtracking allows a programmer to write easily programs which involve searching and/or backtracking in the domains of elementary data structures.

Although "automatic backtracking" is useful, it sometimes incurs serious degradation of computational efficiency in time and memory. "Automatic backtracking" is usually implemented by the depth-first search procedure, so that many redundant backtrackings will be conducted in searching solution-paths in the problem space.

This paper presents and discusses an idea of eliminating redundant backtrackings by using "features" extracted from input data and programs.

Before moving on detailed discussions of our idea, we briefly review and discuss some typical approaches to the problem of reducing redundant backtrackings.

(1)Introduction of programming constructs to control program execution

Cut operator in Prolog is an example of this sort. As is well known, however, the use of cut requires careful consideration about the execution flow of programs, and its excessive use causes loss of their readability.

(2)Utilization of partial results obtained during search

In many heuristic problem solving tasks, the problem decomposition scheme is incorporated. It divides recursively a complex problem into a group of small subproblems and conquers the subproblems easy to solve. In this scheme, many subproblems of the same kind are generated during the decomposition, so that the "blind" sequential search process based on backtracking may solve the same problem many times. Keeping the solutions of such subproblems, we can use them during the search to increase the efficiency of the search process. BUP of [1] employs this sort of approach.

(3)Intelligent(dependency directed)backtracking

Usually, when a search has failed, the control is returned back to the nearest (latest) decision point where alternatives are found. With this simple backtracking scheme, however, many redundant searches may be conducted. This is because the real cause of the failure often resides not in the latest decision but in the one done fairly before. In intelligent backtracking[2][3], the cause of a failure is analyzed to find the decision point where that cause has been introduced, and the control is directly returned back to that decision point.

Our basic idea is as follows. For a given input data we extract features such as its data types and quantitative properties (e.g. length of data), and then we use these features to reduce the number of possibilities at many decision points in nondeterministic searching. Our method reduces backtrackings by eliminating useless possibilities leading to failures by using features extracted from input data. Thus our method differs from intelligent backtracking. (This idea was first introduced in [4] and its general framework was discussed in [7].)

Section 2 describes the general framework of our automatic elimination of backtracking with illustrative examples. Section 3 discusses the automatic feature determination from Prolog programs. Section 4 demonstrates the effectiveness of our idea with a Prolog interpreter designed based on our scheme.

2. GENERAL FRAMEWORK FOR AUTOMATIC BACKTRACK ELIMINATION

In this chapter, we will describe the general framework of our automatic elimination of backtracking. Section 2.1 describes our basic idea, taking a top-down string recognizer as an example. Section 2.2 describes functional modules in a general system for the automatic backtrack elimination. In section 2.3 we discuss several problems of the automatic backtrack elimination in the domain of parsing strings of context free grammars (in short, CFG).

2.1 Fundamental Idea

In order to give a clear understanding of our problem and idea, we first discuss elimination of backtracking in a top-down string recognizer; automatic backtracking is usually used in the top-down depth-first search procedure.

Given a grammar G and an input string x, a standard top-down recognizer determines whether $x \in L(G)$ or $x \notin L(G)$ by analyzing x based on G, where L(G) denotes the language generated by G.

Consider the following simple problem:

$$I+I+I \in L(G1) \ ?$$

where E is the initial symbol and the rules of grammar G1 are defined in BNF as follows:

```
E  ::= T | E+T
T  ::= P | T*P
P  ::= I | (E)
```

Even for this simple problem, a top-down recognizer based on backtracking generates a fairly large search tree until it finds out the answer. But if we know that the input string "I+I+I" contains two special symbols "+" (and no "*", "(", or ")"), then we can eliminate all of the backtrackings. That is, starting from E, apply the rule E ::= E+T twice to generate two special symbol "+", which results in E+T+T. Then use T ::=P and P ::=I, because no "*", "(", or")" is included in the input string.

This fact suggests that the use of features (i.e. special symbols in the above example) in input data enables us to eliminate all or most of backtrackings in the top-down string recognizer.

2.2 General Framework of Automatic Elimination of Backtracking

Fig. 1 illustrates the general configuration of a system for the automatic elimination of backtracking. The task of the system is to analyze input data based on a given rule set (program).

Before the analysis of data, the rule analyzer analyzes the rule set to determine what features are useful to eliminate redundant backtrackings (feature determination). Although useful features vary depending on problem domains, we can, in principle, determine them by analyzing the rule set. The features determined by the rule analyzer are stored as rule features (Fig. 1). (Section 3.1

describes roles of the rule analyzer in detail.)

Given an input data to be analyzed, first the input analyzer examines it to extract features included in the data (input features). Then, the execution controller analyzes the data by using the rule set. During the analysis, the controller uses the input and rule features as well as the execution history to eliminate redundant backtrackings. The execution history denotes partial solutions obtained during the analysis. We will not discuss the use of the execution history in this paper.

The rule organizer analyzes and accumulates pairs of input data and output results and reorganizes the rule set. This is a kind of learning to find useful rules. Here we will not discuss this, either.

2.3 Backtrack Elimination in Top-Down CFG Parsers

There are several problems to be answered in the above scheme:

(a)What are features and how to determine them?

(b)To what extent can backtrackings be eliminated by using features? (In what problems can we eliminate backtrackings completely?)

(c)How much can the computational efficiency be increased by the automatic backtrack elimination? (Note that certain overhead is included in the automatic backtrack elimination: to extract features from the input data and to refer to them during the analysis. Note also that the computation time for the rule analyzer to determine features need not be considered as overhead, because it is performed only once before analyzing data.)

In this section we will discuss these problems in the domain of top-down CFG parsers.

As shown in section 2.1, special terminal and non-terminal symbols would be useful features in the domain of string parsing. Since all of such symbols are defined in the grammar, we can determine which symbols are useful features by analyzing the grammar. Using these symbols as features, an input string is represented by a sequence of features.

Fig. 2 illustrates a model of parser which uses a sequence of input features to select appropriate rules to apply. Note that "select appropriate rules" means "inhibit the application of those rules leading to failure and, as a result, eliminate redundant backtrackings".

The feature table stores the control information for the rule selection. The execution controller refers to the feature table to select rules, using both input feature and internal state as access keys. For example, we can represent the feature table by an array: its row and column indices are specified by the input feature and the internal state, respectively, and each element of the array stores applicable rules under a given pair of input feature and internal state.

This scheme is the same as the model of deterministic parsers for LL(1) and LR grammars[5]. That is, our scheme of backtrack elimination works completely (i.e. eliminate all backtrackings) in parsing strings of these classes of grammars. Since the construction of the feature table can be done before the parsing, the run-time overhead is just the table lookup. This enables fast and efficient parsing without backtracking.

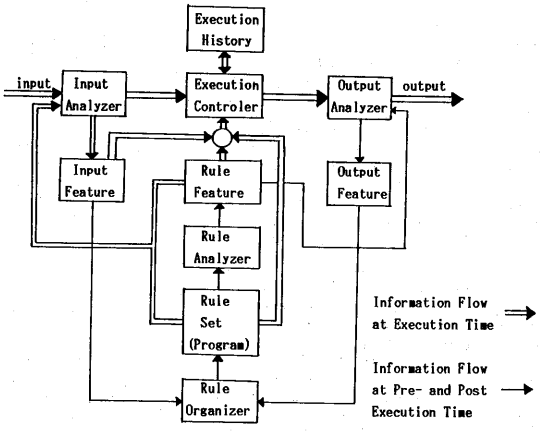


Fig. 1 General framework for the automatic backtrack elimination

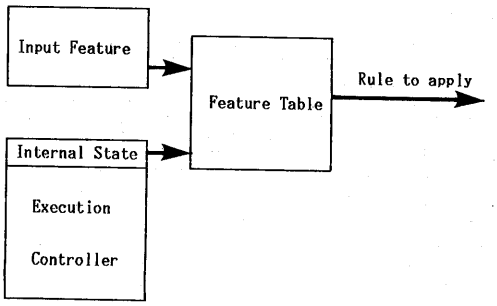


Fig. 2 Model of parser

Single terminal / nonterminal symbols are too primitive features to eliminate backtrackings in the case of general CFG. Thus we have to extract more complex features from input data. In the problem of parsing string, it would be reasonable to regard a sequence of primitive features as a higher order feature, and a sequence of higher order features as a still higher order feature. What sequences should be regarded as features can be determined by a given grammar.

For example, consider the problem of parsing English sentences by the grammar shown in Fig. 3. We first assign grammatical category(s), such as N and V, to each word in the input sentence. This process is the extraction of first order features from the input data. Then each pair of consecutive first order features is grouped into a second order feature based on the rules. For example, a sequence, V NP, can be grouped into VP by using the rule VP -> V,NP. The feature extraction can be continued until n-th order feature is extracted for an input sentence consisting of n words. Structures of higher order features are represented hierarchically in terms of lower order features (Fig. 3).

This process of extracting higher order features is the same as well-known CYK algorithm[5]. The triangular table in CYK algorithm corresponds to our input features. Thus, our method can eliminate all backtrackings even in the case of general CFG. (Note that our scheme of feature extraction can stop when k-th order features are extracted (k < n).)

In this scheme most of the computation time is

used to extract features from input data, and in the extreme case, where the process is performed until extracting all n-th order features, no computation time is required by the execution controller (Fig. 1) to determine whether or not a given input string $\in$ L(G). That is, all substantial computation is performed by the input analyzer. As is well known in pattern recognition, this phenomenon shows the trade-off between feature extraction and recognition processes; the use of complex features reduces the computation necessary for the recognition, while their extraction requires lots of computation.

The two case analyses described above shows that our scheme includes deterministic parsers for LL(1) and LR grammars and CYK parsing algorithm, where all backtrackings can be eliminated. Note that our scheme will be more powerful than these ordinary parsing algorithms; we can recognize a string belonging to a class of context sensitive languages such as $\{a^n b^n c^n \mid n>1\}$ by using quantitative features such as the length of character sequences. That is, features we can use are not limited to symbol sequences, but structured and quantitative features can be used in complex problems.

3. AUTOMATIC FEATURE DETERMINATION

This chapter discusses automatic determination of features useful for the backtrack elimination. Section 3.1 gives detailed discussions on roles of the rule analyzer. In section 3.2 we describe a method of automatic feature determination from Prolog programs.

3.1 Roles of The Rule Analyzer

From a practical point of view, the followings would be typical useful features to eliminate backtracking:

(1)special (terminal) symbols
(2)data types (non-terminal symbols)
(3)special sequences of symbols
(4)length of string
(5)number of occurrences of each symbol in input data
(6)structural information (e.g. levels of parentheses)

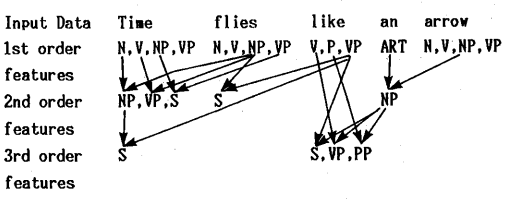
Since all symbols and their combinations to be used as features are defined by the rule set (program), we can determine all possible candidates for useful features by analyzing the rule set. The first role of the rule analyzer (Fig. 1) is to determines useful features by analyzing the rule set.

In order to eliminate redundant backtrackings, the execution controller requires the control information about which rules should be used in what situations. In Fig. 2, this information is represented by the feature table and situations are described in terms of input features and internal states. The second task of the rule analyzer is to add such control information to the rule set.

Analyzing the rule set, the rule analyzer generates a set of "rule selection rules" (the feature table in Fig. 2) in the following form:

If (situation),
THEN apply specific rule(s) in the rule set.
(i.e. discard useless rules leading to failure)

where situations are described in terms of features. Thus, features are used to describe conditions to apply rules as well as to characterize input data.



(a)Extracting higher order features

S -> NP,VP
VP -> VP,PP
VP -> V,NP
VP -> V
NP -> NP,PP
NP -> N,NP
PP -> P,NP

N -> Time | flies | arrow
V -> Time | flies |like | arrow
P -> like
ART -> an
NP -> ART,NP
NP -> N

(b)Grammar

Fig. 3 Extraction of higher order features

(a)P([],*x,*y):- ... P([*u|*v],*w,*z):- ...
(b)P(a,*x,[]) :- ... P(b,*y,[*u|*v]) :- ...
(c)P(*x,*y) :- ... P(*u,*u) :- ...
Ci Cj

Fig. 4 Feature determination from head predicates

3.2 Automatic Feature Determination from Prolog Programs

Backtrackings in Prolog are caused by inappropriate selection of clauses; since there are multiple clauses with the same head predicate, a Prolog interpreter (execution controller in our scheme) selects one of them according to the predefined order among the clauses. It is this selection of clauses that we want to make "smart" by using features; features are used to select appropriate clauses (i.e. apply promising ones first, and discard useless ones leading to failure).

3.2.1 Feature Determination from Head Predicates

Let {C1,C2,...,Cn} denote a set of clauses with the same head predicate P. The simplest way to determine features is to apply unification for every pair of head predicates of Ci and Cj ($i \neq j$).

(a)If the unification fails, then the cause of the failure is the feature to discriminate the application of Ci and Cj. For example, in the case of Fig. 4(a), the cause of the failure is the difference in the data type of the first argument. Thus we generate the following "clause selection rules".

IF the first argument of predicate P is NIL,
THEN select Ci
IF the data type of the first argument of P is LIST, THEN select Cj.

The interpreter uses these clause selection rules to select appropriate rule to apply.

Since clause selection rules directly examine argument(s) useful for the clause selection, it will take less computation time to select appropriate clauses than the ordinary unification. For example, suppose predicate P has N arguments and k-th argument of the head predicates of Ci and Cj have features (i.e. could not be unified). Then, given a goal predicate P(x1,x2,...,xN) to be resolved, the

rule selection rules first examine its k-th argument to select applicable clause(s). On the other hand, the ordinary unification checks the arguments one by one from x1.

Note that the unification to extract features should be continued until the last argument even if a certain intermediate argument fails to be unified; multiple arguments may have features useful for the clause selection. For example, since the first and third arguments of the head predicates in Fig. 4(b) cannot be unified, clause selection rules examine both of these arguments of a given goal predicate.

(b)If the unification of two head predicates succeeds, then it is very hard to find useful features without modifying the semantics of the program. This is because Prolog programs are usually written so that they may be correct when they are executed according to the sequential top-to-bottom examination of clauses.

For example, two head predicates in Fig. 4(c) can be unified while the lower (right) one has a feature that the first and second arguments are the same. If a current goal predicate has the same value in these two arguments, then it would be reasonable to apply clause Cj first. This, however, is only a heuristic, because the application of the upper (left) clause may succeed. Thus it would be safe not to modify the order of the application of clauses when their head predicates are unifiable.

In short, the backtrack elimination using features determined from head predicates can be considered as an extension of the popular indexing mechanism for fact clauses. That is, a set of clause selection rules can be considered as an indexing structure for selecting appropriate clauses.

3.2.2 Feature Determination by Program Analysis

The feature determination from head predicates is not so powerful to eliminate backtrackings; the features we can use are superficial, and only so-called shallow backtrackings can be eliminated. Thus, we have to analyze the entire program to determine features useful for substantial elimination of backtrackings.

In what follows, we will describe a method of feature determination from Prolog programs, taking Fig. 5 as an example. Fig. 5(b) shows a Prolog program for the top-down string recognition, where the original grammar rules are given in Fig. 5(a). In this program, strings are represented by lists, and variables are marked with *

Fig. 5(c) illustrates the network structure to represent mutual relations among the clauses (1)-(8) (except the query clause). Nodes E, T etc. denote names of head predicates, which we call head predicate nodes. Clauses with the same head predicate are represented by bold arcs attached to these nodes. The number associated with each bold arc corresponds to the clause number in Fig. 5(b).

Variables associated with solid arcs such as *sE and *eE denote formal arguments of head predicates. Variables in each set braces such as {*sE, *s1} denote those which are unified when a clause is applied. Each solid arc connects a pair of a variable in a clause and a formal argument of a head predicate, which are unified when the body of the clause is activated. Each broken arc connects a functor and its arguments.

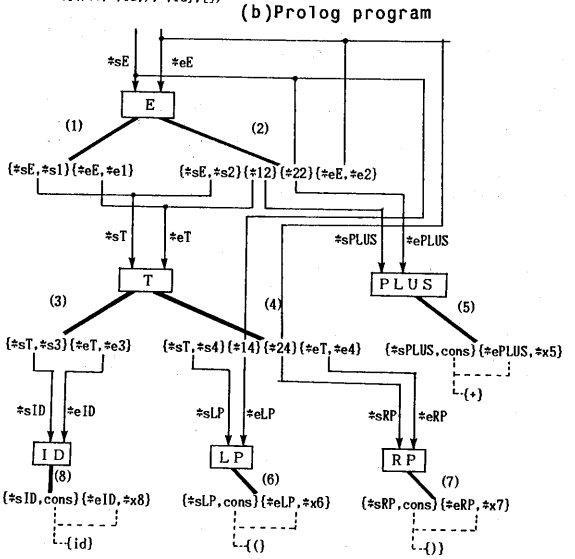
This network represents all dependency relations between clauses and variables. The (informal) algorithm for the feature determination using this network is as follows:

E::= T | T + E T::= id | (E)

(a)Grammar

```
(1) E(*s1,*e1):-T(*s1,*e1)
(2) E(*s2,*e2):-T(*s2,*12),PLUS(*12,*22),E(*22,*e2)
(3) T(*s3,*e3):-ID(*s3,*e3)
(4) T(*s4,*e4):-LP(*s4,*14),E(*14,*24),RP(*24,*e4)
(5) PLUS([+|*x5],*x5).
(6) LP([([|*x6],*x6).
(7) RP([)|*x7],*x7).
(8) ID([id|*x8],*x8).
(9) ?-E([(,id,+,id,),+,id],[])
```

(b)Prolog program



(c)Network structure for the feature determination

Fig. 5 Feature determination from a Prolog program

[STEP 1]
Consider constants and functor names as fundamental features. Starting from those clauses with such features, propagate the features backward to those clauses which may call them. For example, the first argument of predicate ID in clause (8) has a feature, *sID = cons(id,*eID) (= [id|*eID]). This feature is propagated to clause (3) through the solid arc attached to head predicate node ID. Then clause (3) is given a feature *sT = [id|*eT]. Similar feature propagations are performed from clauses (6), (7), and (5) to (4), (4), and (2) respectively. Note that only those features associated with formal arguments of head predicates can be propagated.

[STEP 2]
Examine such head predicate nodes that have multiple bold arcs (clauses), and then determine features to discriminate the clauses by comparing propagated features. The feature determination is done by the same method as described in 3.2.2. If features are found (i.e. a pair of head predicates with propagated features cannot be unified), then generate and store clause selection rules in corresponding head predicate nodes.

For example, after the feature propagation, head predicate T has the feature *sT = [id|*eT] for clause (3), and *sT = [(|*14] for clause (4). Note that here again we only have to examine features associated with the formal arguments of head predicate T: the feature, *24 = [)|*eT], which have been propagated from clause (7) to (4), need not be used for the feature determination. Since the unification of two *sT's of clauses (3) and (4) fails, we

can obtain the following clause selection rules:

IF the first argument of T is [id|*eT], THEN apply clause (3).
IF the first argument of T is [(|_], THEN apply clause (4)

where *eT and _ denote the second argument of head predicate T and an unnamed variable, respectively. These rules are stored in head predicate node T in the network. The execution controller (Prolog interpreter) refers to them when its current goal predicate to be resolved is T. In this sense, this network corresponds to the feature table in Fig. 2.

[STEP 3]
Since the one-step feature propagation is not enough to obtain useful features and clause selection rules, we have to iterate the above processes.

The feature propagation involves several complicated processes:

(i)Multiple different features may be propagated through a head predicate node.

For example, as described above, two features are propagated to the first argument of T: *sT = [id|*eT] from clause (3) and *sT = [(|_] from clause (4). These two features should be combined and propagated to clauses (1) and (2). We represent such combined feature by *sT = [id|*eT]/[(|_], where / denotes "or".

(ii)Multiple features may be propagated to a variable.

For example, variable *12 in clause (2) has two solid arcs from T and PLUS. Feature *12 = [+|*22] is propagated from PLUS, while no feature from T. Since a solid arc connects a pair of argument and variable to be unified, this feature should also be propagated to the second argument of head predicate T, *eT. Then, the feature is propagated downward through head predicate node T, and features are recomputed for clauses (6), (7), and (8): *sID = [id,+|_] for (8) and *sRP = [],+|_]. These new features, in turn, are propagated upward to clause (3) and (4). As the result of this propagation of the new features, the feature of clause (3) becomes *sT = [id,+|_]. Note that the feature of clause (4) is not changed. Note also that since the feature of clause (3), *sT = [id,+|_], is obtained by combining clauses (2) and (3), it is not propagated to clause (1) from node T: the feature propagated to clause (1) from (3) is the old one: *sT = [id|*eT]. Finally, the features of clause (1) and (2) become *sE = [id|*eE]/[(|_] and *sE = [id,+|_]/[(|_], respectively.

[STEP 4]
In general, the network has many loops, so that the feature propagation does not terminate. Thus we have to stop it after a certain number of iterations of the feature propagation.

To investigate the effectiveness of the features determined by this method, we compared the search trees in the following three cases:

(A)The normal execution without backtrack elimination

(B)Backtrackings are eliminated by using the clause selection rules associated with node T.

(C)Although there is no distinguishing feature between clauses (1) and (2) (i.e. two features of *sE's shown above can be unified), we used the following (heuristic) clause selection rule in addition to (B):

IF the first argument of the predicate E is [id,+|_], THEN select clause (2).

Fig. 6 illustrates the search trees in the above three cases given the goal clause (9) in Fig. 5(b). The trees represent the depth-first search processes with backtracking. Each arc in the trees denotes an applied clause and is labeled with its clause number. Although not all backtrackings were eliminated, even simple features could eliminate many redundant backtrackings.

REMARKS:
The process of the feature determination described above can be regarded as a method of partial evaluation of Prolog programs. While the method described in [6] performs the top-down partial evaluation starting from a goal clause, our method is purely bottom-up starting from those clauses with primitive features (i.e. constants and functors). Moreover, ours extracts the explicit control information (i.e. clause selection rules) by analyzing a program.

4. PERFORMANCE EVALUATION WITH A PROLOG INTERPRETER

4.1 TU-Prolog

In this section we will demonstrate the effectiveness of our idea of backtrack elimination with a Prolog interpreter, TU-Prolog. TU-Prolog is implemented on MC68000 based UNIX workstation (U-station E15). Its characteristics are:

(a)Prolog interpreter written in C
(b)Binary list for the internal data representation
(c)Structure copying method
(d)DAP (Dynamic Program Analyzer) for monitoring the execution flow of Prolog programs.
(e)Automatic backtrack elimination facility

It is often useful to monitor the execution flow of Prolog programs. DAP in TU-Prolog displays the depth-first search tree (the execution flow of a program) on a graphic display. For example, all trees in Fig. 6 and Fig. 8 were drawn by DAP, where numbers associated with branches denote selected clauses. DAP enables a user to understand the execution flow of programs and to find useful features to eliminate redundant backtrackings.

4.2 Automatic backtrack elimination in TU-Prolog

A simple automatic backtrack elimination facility has been implemented in TU-Prolog. Since the major objective of this facility is to examine the effectiveness of our idea of backtrack elimination, the implemented facility is very limited. Firstly no automatic feature

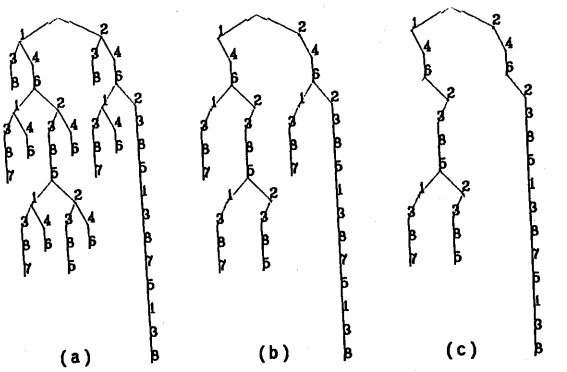


Fig. 6 Search trees

determination is implemented, so that a user has to specify features. Secondly we assume that the first argument of each head predicate is a list and that features are extracted from that list.

The process of the backtrack elimination consists of the following steps:

(1)Feature specification
 (1-1)Feature declaration
 (1-2)Declaration of clause selection rules

(2)Execution
 (2-1)Feature extraction from input data
 (2-2)Execution of the program based on the input features and the clause selection rules

Predicate "feature-class" is used to declare features. The following two kinds of features can be specified:

(i)type specification for constants
(ii)type specification for a pair of consecutive features.

For example, (feature-class + 2) defines symbol "+" as a feature of class 2, and (feature-class (2 3) 4) a pair of features of classes 2 and 3 as a feature of class 4. Note that since TU-Prolog uses the list structure, a clause P:-Q1,Q2,Q3 is represented by (P Q1 Q2 Q3). Besides these user specified features, the system regards symbols "(" and ")" as special features specifying levels of the structure of input data.

To define clause selection rules, we use
(select-clause head-predicate-name
 ((feature-class-1 clause-number)
  .
  (feature-class-i clause-number)
  .
  (feature-class-n clause-number)))
This works as follows. First associate the pairs of feature-class and clause-number with the head predicate specified by head-predicate-name. In selecting a clause to apply, examine those pairs associated with the current goal predicate. If current data under analysis has a feature of feature-class-i, then apply its corresponding clause. If no clause selection rule is associated with the head predicate or no feature is included in the data, the interpreter applies the clauses in the standard order.

Given a goal clause (input data), the interpreter first extracts features. Fig. 7 shows the internal data structure (feature table) to represent input features, where symbols "*" and "+" are declared as features. The feature table is the same list structure as the input data. Each element of the list records such features that
i)appear in the right portion of the input data from that element
ii)and are at the same level of parenthesis.
On execution, the interpreter refers to both this input feature table and the clause selection rules.

4.3 Performance Evaluation

Fig. 8(a) shows a Prolog program for syntactic analysis of English sentences. We used the following two features for this program:
(i)a sequence of VERB and PREPOSITION
 (e.g. ran through)
(ii)a sequence of NOUN and PREPOSITION
 (e.g. wolf with)
where VERB, NOUN, and PREPOSITION denote the types (feature classes) defined for constants "ran", "wolf", "through", and "with", respectively. The clause selection rules we used are
(a)IF there exists a feature of type (i),
 THEN select clause (2) (Fig. 8(a)).

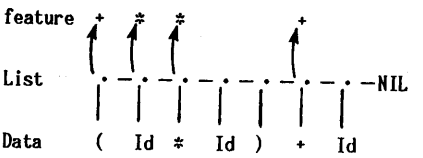
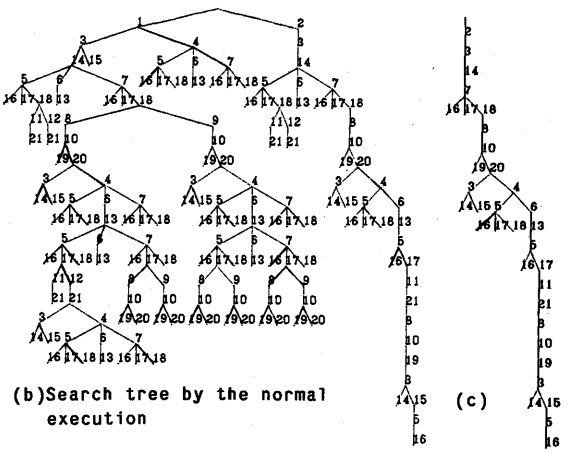


Fig. 7 Internal representation of input features

```
?- S([a,wolf,with,red,eys,ran,through,the,bush],[])

(1)  S(*s,*e):-NP(*s,*1),VP(*1,*e)
(2)  S(*s,*e):-NP(*s,*1),VP(*1,*2),PREPS(*2,*e)
(3)  NP(*s,*e):-DET(*s,*1),NP2(*1,*e)
(4)  NP(*s,*e):-NP2(*1,*e)
(5)  NP2(*s,*e):-N(*s,*e)
(6)  NP2(*s,*e):-ADJ(*s,*1),NP2(*1,*e)
(7)  NP2(*s,*e):-N(*s,*1),PREPS(*1,*e)
(8)  PREPS(*s,*e):-PP(*s,*e)
(9)  PREPS(*s,*e):-PP(*s,*1),PREPS(*1,*e)
(10) PP(*s,*e):-P(*s,*1),NP(*1,*e)
(11) VP(*s,*e):-V(*s,*e)
(12) VP(*s,*e):-V(*s,*1),NP(*1,*e)
(13) ADJ([red|*x],*x).
(14) DET([a|*x],*x).
(15) DET([the|*x],*x).
(16) N([bush|*x],*x).
(17) N([eyes|*x],*x).
(18) N([wolf|*x],*x).
(19) P([through|*x],*x).
(20) P([with|*x],*x).
(21) V([ran|*x],*x).
```

(a)A Prolog program for English sentence analysis



(b)Search tree by the normal execution

(c)

Fig. 8 Backtrack elimination in English sentence analysis

(b)IF there exists a feature of type (ii),
 THEN select clause (7) (Fig. 8(a)).

The experiment using these features showed great improvement in computation time: 96 msec by the standard execution with backtracking (Fig. 8(b)) was reduced to 30msec (Fig.8(c)). Note that the latter includes the computation time (overhead) for the feature extraction from the input data. The overhead in this example was 7msec. This means that the overhead for the feature extraction is not negligible.

Table 1 summarizes the performance of the backtrack elimination in the recognition of several arithmetic expressions by the program shown in Fig. 5(b), where we used constants "+" and "*" as features.

[Observation1]
The computation time by the normal execution varies very much. This means that many redundant backtrackings were conducted depending on input data.

[Obvervation2]
The computation time by the backtrack elimination is almost proportional to the length of input data. This implies that the features we used properly characterized input data and that almost all redundant backtrackings were eliminated.

[Observation3]
In the second example, the backtrack elimination took more time than the normal execution. This means no redundant backtrackings were eliminated and that the overhead for the feature extraction increased the computation time by the backtrack elimination. This is because "(" and ")" are not regarded as features to eliminate backtracking but are used only to specify the structure of input data. Of course we can declare these symbols as features. With such features, the computation time for the second example by the backtrack elimination became 16.3msec.

5. Concluding Remarks

The idea of automatic elimination of backtracking has been proposed and its effectiveness has been demonstrated in several Prolog programs. The followings are the summary of the discussions given in this paper and future problems to be studied.

[1]Our idea of automatic elimination of backtracking includes efficient deterministic parsing algorithms of CFG, where all backtrackings can be eliminated.

[2]Experimental results have shown that even simple features can eliminate considerable amount of redundant backtrackings. This is the essence of our idea of backtrack elimination.

[3]The complete algorithm of determining features by unifying head predicates should be devised; it is not trivial to find causes of a failure in the unification.

[4]The process of the feature propagation described in 3.2.2 cannot find such features as matched parentheses and quantitative features such as length of data. Since these features will be really effective in complex problems, the feature declaration facility as described in 4.2 would be necessary to incorporate the automatic backtrack elimination into programming systems.

[5]In [8], we proposed an extended unification in Prolog, where nondeterminism is introduced in the unification process. The automatic backtrack elimination will also be very effective to implement the extended unification.

This work was supported by the Ministry of Education, Japanese Government under Grant-in-Aid for Scientific Research (No.6011003 and 60420035)

Input data	Normal Execution	Backtrack Elimination
I+I+I+I+I	29.7msec	21.2msec
(((I)))	18.2msec	19.0msec
((I)+I)+I	66.3msec	21.8msec
(((I)))+I	79.8msec	22.7msec
(I+I+I+I+I)	36.7msec	26.2msec
(I+I)+(I+I+I)	56.3msec	29.8msec

Table 1 Experimental results

[REFERENCES]

[1]Y.Matsumoto et al, BUP:A bottom-up parser embedded in Prolog, New Generation Computing, Vol.1, No.2, pp.145-158, 1983.

[2]P.T.Cox and T.Pietrzykowski, Deduction plans:A basis for intelligent backtracking, IEEE Trans., Vol.PAMI-3, No.1, pp.52-65, 1981.

[3]S.Martwin and T.Pietrzykowski, Intelligent backtracking in plan-based deduction, IEEE Trans., Vol.PAMI-7, No.6, pp.682-692, 1985.

[4]T.Ito and A.Yosai, A functional programming language for string processing based on pattern matching, Tech. Rep. of Working Group on Symbol Manipulation, WGSYM24-8, 1983 (in Japanese).

[5]A.V.Aho and J.D.Ullman, Principles of compiler design, Addison-Wesley, 1977.

[6]A.Takeuchi et al, Partial evaluation and its application to meta programming, Tech. Rep. of Working Group on Software Foundation, WGSF13-2, 1985 (in Japanese).

[7]T.Ito, On automatic elimination of backtracking from languages with backtracking, 1984 (not published memo).

[8]K.Yamazaki, T.Matsuyama, and T.Ito, On extended unification in Prolog, Nat. Convention Rec. of IECE Japan, 1461, March 1986 (in Japanese).