Developing an EBL Bypass for a Large-Scale Natural Language Query Interface to Relational Data Bases

by

Christer Samuelsson and Manny Rayner
Developing an EBL Bypass for a Large-Scale Natural Language Query Interface to Relational Data Bases*

Christer Samuelsson and Manny Rayner

Swedish Institute of Computer Science
Box 1263, S-164 28 KISTA
Sweden

christer@sics.se, manny@sics.se
SICS Research Report R91/R91001

Abstract

The syntactic analysis component of a large-scale natural language query interface to relational data bases was greatly sped up by applying explanation-based learning (EBL), a machine learning technique.

The idea is that one for most input queries can bypass the ordinary parser, instead using a set of learned rules. When no rule applies, one must pay the price of a small overhead. The set of learned rules is extracted automatically from sample queries posed by a user, thus tuning the system for that particular user.

Several non-trivial problems, arising from the characteristics of the target system, were solved during the project. Measurements on a small test corpus indicated that the speed-ups when a learned rule could be used are on average a factor 30 and that the overhead, when no rule applied, is less than 3 percent.

* This is an extended version of a paper titled "Using Explanation-Based Learning to Increase Performance in a Large-Scale NL Query System" by the same authors, that was presented at the third DARPA Workshop on Speech and Natural Language, Hidden Valley, 1990.
1. Introduction

The target system is a large-scale natural language query interface to relational databases developed at IBM Nordic Laboratories, Lidingö. Processing times in natural language systems tend to be quite high, higher than many users may want to accept. Therefore, we were asked to try to speed up the system's analysis component by using explanation-based learning (EBL), a machine learning technique that we had previously applied to the CHAT-80 system [Rayner 88], [Rayner & Samuelsson 89]. In the large-scale natural language query interface to relational data bases, we applied the EBL technique to syntactic analysis only, since this phase was by far the most time consuming. Our work developing an EBL Bypass for the large-scale natural language query interface to relational data bases was carried out during first of November 1989 to last of June 1990.

The idea behind the EBL bypass is that one, for most input queries, can bypass the ordinary parser, instead using a set of learned rules, and thus vastly speed up the processing. This is done paying the price of a small overhead when no rule proves applicable. The set of learned rules is extracted automatically from sample queries posed by a user. By extracting the rules from real user interaction, the set of rules is tailored so that it can deal with the user's most common query types. The hope is that a comparatively small set of query types will account for the vast majority of the queries actually posed. Whether this is in practise true or not remains yet to be ascertained.

Technically, the project was a success; we managed to circumvent several non-trivial problems arising from the fact that the target system is a large-scale system intended for serious applications, and achieved a speed up of one and a half order of magnitude at an overhead of less than 3 percent. Unfortunately we were unable to carry out a thorough investigation of how the coverage and speed-ups depend on the size of the set of learned rules for reasonably large sets. We intend to address this important issue in subsequent reports.

The rest of the report is laid out as follows: In section 2 we give a picture of what EBL means in the natural language domain, in section 3 we describe the architecture of the EBL module developed, in section 4 we report the results from preliminary timing studies and in section 5 we summarize our experiences and point to further research areas. Appendix 1 holds the generalizer code together with explanatory notes and appendix 2 tells the tail of translating the EBL code from SICStus to VM-Prolog.

2. What EBL is

Explanation-based learning is a machine-learning technique, closely connected to other techniques like macro-operator learning, chunking, and partial evaluation; a phrase we have found useful for describing the method to logic programmers is example-guided partial evaluation; a phrase that may appeal to a natural-language person is grammar rule chunking. The basic ideas of the method are well-described in an overview article which appeared in Artificial Intelligence [Minton et al 89], to which we refer the reader who wants to understand the general principles; here, we will only describe briefly what EBL means in the context of natural-language processing by giving an example of one-level generalization using a toy NL system.
2.1 Example of a derivation

Consider the toy grammar and lexicon in diagram 1:

\[
\begin{align*}
S & \rightarrow nP (Agr, VP^S), \ VP (Agr, VP). \\
NP (3-s, NP) & \rightarrow pN (NP). \\
nP (Agr, NP) & \rightarrow det (N(L_N^NP)), \ n (Agr, N(L)). \\
VP (Agr, X^S) & \rightarrow tv (Agr, X^VP), \ NP (VP^S). \\
VP (Agr, VP) & \rightarrow iv (Agr, VP). \\
PN (PN^VP)^VP & \rightarrow [PN], \ \{\text{lex}(PN, pn)\}. \\
det (X^S1)^X^S2^Quant (Det, X, S1, S2) & \rightarrow \\
&Det, \ \{\text{lex}(Det, det)\}. \\
n (Agr, X^L_N^[N, X]) & \rightarrow [N], \ \{\text{lex}(N, n (Agr))\}. \\
tv (Agr, X^Y^L_TV, X, Y) & \rightarrow [TV], \ \{\text{lex}(TV, tv (Agr))\}. \\
iv (Agr, X^L_IV, X) & \rightarrow [IV], \ \{\text{lex}(IV, iv (Agr))\}. \\
\end{align*}
\]

\text{lex} (john, pn).
\text{lex} (a, det).
\text{lex} (cat, n (3-s)).
\text{lex} (loves, tv (3-s)).
\text{lex} (sleeps, iv (3-s)).

Diagram 1, toy grammar and lexicon

This grammar can parse a few trivial sentences like John sleeps or John loves a cat, and associate with each a corresponding expression in first-order logic; we will now show what happens when the second sentence is generalized with respect to the lexicon.

We first construct the normal derivation tree for the original sentence, or to be more exact for the proposition

\[
s (\text{quant} (a, A, [\text{cat}, A], [\text{loves}, \text{john}, A]), [\text{john}, \text{loves}, a, cat], []). 
\]

(The first argument - the logical form - is to be read as "An A such that [cat, A] is also such that [loves, john, A].") The derivation tree will be as shown in diagram 2 where each node has been marked with the number of the clause resolved on at that point.

Diagram 2, derivation of "John loves a cat"
Now we want to generalize away the lexical information present. To do this, we perform the same series of resolution steps, but this time omitting all resolutions with unit clauses of the type \( \text{lex}(*,*) \). This will yield us the conditional derivation tree in diagram 3, where the assumptions have been written in bold-face.

\[
\begin{align*}
\text{lex}(A, \text{pn}) & \quad \text{lex}(B, \text{tv}(3-s)) & \quad \text{lex}(C, \text{det}(\text{Agr})) & \quad \text{lex}(D, n(\text{Agr})) ; \\
(6) \quad \text{pn}((A^{\text{VP}})^{\text{VP}}, [A \mid R], R) & \quad (9) \quad \text{tv}(3-s, X^{Y}[B, X, Y], [B \mid R], R) & \quad (7) \quad \text{det}(\text{Agr}, (X^{S_1})^{(X^{S_2})}^{\text{quant}(C, X, S_1, S_2), [C \mid R], R}) & \quad (8) \quad n(\text{Agr}, Y^{[D, Y]}, [D \mid R], R) \\
(2) \quad \text{np}(3-s, (A^{\text{VP}})^{\text{VP}}, [A \mid R], R) & \quad & \quad (3) \quad \text{np}(\text{Agr}, (Y^S)\text{quant}(C, Y, [D, Y], S), [C, D \mid R], R) \\
(1) \quad s(\text{quant}(C, Y, [D, Y], [B, A, Y]), [A, B, C, D], []) & \quad (4) \quad \text{vp}(3-s, X^{\text{quant}(C, Y, [D, Y], [B, X, Y]), [B, C, D \mid R], R}) \\
\end{align*}
\]

Diagram 3, generalized derivation tree

The suspended calls to operational goals are then collected to form the learned rule in diagram 4. Since the new tree represents a valid derivation for any values of the (meta-)variables \( A, B, C, D \) and \( \text{Agr} \), it thus constitutes a proof of the learned rule.

\[
s(\text{quant}(C, Y, [D, Y], [B, A, Y]), [A, B, C, D], []) \leftarrow \\
\text{lex}(A, \text{pn}) \& \\
\text{lex}(B, \text{tv}(3-s)) \& \\
\text{lex}(C, \text{det}(\text{Agr})) \& \\
\text{lex}(D, n(\text{Agr})).
\]

Diagram 4, generalized derived rule

Thus we are now in possession of a learned rule which can handle not only \textit{John loves a cat}, but also many other sentences, for example \textit{Harry hates the winter}.

2.2 Indexing

Once we have derived a large set of learned rules, it will be too time consuming to simply try one after the other until we stumble upon the proper one. All computing time gained from using learned rules instead of normal processing, will be devoured by costly linear search among the learned rules, first shown in the planning domain by Steve Minton [Minton 88] and later shown in the natural-language domain [Rayner & Samuelsson 89]. We must index the learned rules to ensure quick access to them. The obvious way to filter out most learned rules as inapplicable, is to inspect the lexical categories of the words in the input sentence (or phrase); all sentences (or phrases) a particular learned rule can handle must have the very same sequence of lexical categories.

The key indexing method, described in detail in [Rayner & Samuelsson 89], summarizes the structure of the input phrase in an atomic key, which is supplied as the first argument of the predicate containing the learned rule, thus utilizing the first functor
indexing facility of most Prolog implementations. The method uses a generate and test matching algorithm and proved most appropriate for indexing one-level rules, since the length of the key is fixed and there is no need to attempt sub-string matching. The thing that can trip up this method is lexical ambiguity, since the number of possible keys grows combinatorially with the number of lexical ambiguities. For a normal sentence, though, this is not a serious problem. On the other hand, the method did not prove that efficient for two-level rules, where we treat noun phrases as primitive on top-level and use a set of sub-rules for constructing noun phrases. This problem arises from the fact that for every input sentence, there are a vast number (approximately \(N^2/2\), where \(N\) is the number of words in the sentence) of possible sub-phrases, each potentially matching a sub-rule (i.e. being a noun phrase) and thus requiring a key of its own. The method simply drowns in a flood of possible keys arising from lexical ambiguities and the number of possible sub-strings.

Instead of key indexing, we stored the learned rules in a decision tree, using a demand driven algorithm dispatching on lexical category. In the one-level case this works as follows:

Consider indexing the rule derived from *John loves a cat* in section 2.2:

\[
s(\text{quant}(C, Y, [D, Y]), [B, A, Y]), [A, B, C, D], []) :-
\]

\[
\text{lex}(A, \text{pn}),
\text{lex}(B, \text{tv}(\text{3-s})),
\text{lex}(C, \text{det}(\text{Agr})),
\text{lex}(D, n(\text{Agr})).
\]

Say that we wish to store it in the format

\[
\text{learned_rule} \text{(RuleNumber, InputSentence, LogicForm)} :-
\text{constraints}.
\]

We start at the start node *sent_root* in the decision-tree and add a path of tree arcs labelled with the lexical categories of the input sentence, if possible using already existing arcs. The last node will be, as are all leaves, a pointer to the learned rule. Thus, the rule above will result in the following entries:

\[
\text{tree_arc} \text{(sent_root, pn, node1).}
\]
\[
\text{tree_arc} \text{(node1, tv, node2).}
\]
\[
\text{tree_arc} \text{(node2, det, node3).}
\]
\[
\text{tree_arc} \text{(node3, n, rule42).}
\]
\[
\text{learned_rule} \text{(rule42, [A, B, C, D], quant(C, Y, [D, Y], [B, A, Y])) :-}
\]

\[
\text{lex}(A, \text{pn}),
\text{lex}(B, \text{tv}(\text{3-s})),
\text{lex}(C, \text{det}(\text{Agr})),
\text{lex}(D, n(\text{Agr})).
\]

To find a learned rule at run-time we do the following: First, we find the start node *sent_root* in the decision-tree, then we inspect the lexical category of the first input constituent. If the start node has an arc labelled with this category, we follow it to a child node, otherwise we give up. If we were successful, we then inspect the category of the next input word. If our current tree node has an arc labelled with this category, we follow it, and so we continue until all of the input words are consumed. For all but the last input constituent, we only consider arcs leading to internal nodes.

The end node should be a leaf, and thus contain a pointer to a learned rule. If it is not, we search those of its siblings with arcs labelled with the correct category for one who is. We then attempt to apply the rule thus found. If this rule does not apply, we search all remaining such siblings for leaves and try the corresponding rules in turn until either one is found that applies or no further rule reference remains, in the latter case we fail.

When indexing two-level rules we do the very same thing as in the one-level case, starting at the start node *sent_root* in the decision-tree for sentence (top-level) rules and starting at *np_root* for noun-phrase (sub-)rules. Two level generalization of the derivation
of a logic form for *John loves a cat* will give rise to in three different rules: One for each of the noun phrases "John" (rule23 below) and "a cat" (rule666 below), and one for the sentence rule NP tv NP (rule42 below). They will be indexed as follows:

```prolog
tree_arc(np_root, pn, rule23).
tree_arc(np_root, det, node1).
tree_arc(node1, n, rule666).
tree_arc(sent_root, np, node1).
tree_arc(node2, tv, node3).
tree_arc(node3, np, rule42).
learned_rule(rule23, [A], A) :-
  lex(A, pn).
learned_rule(rule666, [A, B], Y^S^quant(A, Y, [B, Y], S)) :-
  lex(A, det(Agr)),
  lex(B, n(Agr)).
learned_rule(rule42, [A, B, C], NPl(Y^B[NP1, Y])^NP2) :-
  np(np(3-s, B), NP1, A),
  lex(B, tv(3-s)),
  np(np(Agr, G), NP2, C).
```

In the two-level case, we begin the analysis of each word in the input sentence by searching the current tree node for an arc labelled np ("noun phrase"). If such an arc is found we search for potential noun phrases beginning with this word. If the current tree node does not have an arc labelled np or if the search for noun phrases fails, the search proceeds as in the one-level case.

The search for noun phrases is conducted by selecting np_root, the start node for noun phrases in the decision-tree, and then proceeding as in the one-level case described above, with the difference that we at each tree node consider arcs to leaves as well as to internal nodes. Thus we are not required to consume all input words before attempting to apply a learned rule. As soon as a rule is found that applies, we return with the noun phrase constructed by it and continue the search at top-level.

When we have returned to the top-level with a noun phrase, we move to a child node following the arc labelled np in the top-level tree and continue the search from the word following the noun phrase. We do not pursue the search for alternative noun phrases until forced to by back-tracking. To avoid repeating work, it proved useful to include a well-formed sub-string table for the derived noun phrases.

Though an order of magnitude slower than the one-level key indexing method, the decision-tree indexing scheme proved almost an order of magnitude faster than two-level key indexing, and had much better worst case behaviour.

3. Design of the EBL module

We will now describe the way the ideas in the previous section were realized for the large-scale natural language data base query interface to relational data bases.

3.1 Overall architecture

The EBL module can naturally be divided into its compile-time and run-time parts. The compile-time components fall into two categories; the grammar pre-processor, which converts the system's original grammar into a suitable pure Horn-clause representation and the learning component, which extracts learned rules from training examples. The grammar pre-processor is necessary, since the grammar, as it appears in the system, is not formulated as a pure Horn-clause program, whereas the learning component requires such a formulation of the grammar.
Diagram 5, the grammar pre-processor.

For convenience, we will sub-divide the learning component into three smaller components. These are the generalizer, which performs the actual extraction of learned rules, the simplifier, which attempts to reduce them in size by removing unnecessary calls and the rule generator, which indexes the learned rules to ensure quick access to them at run-time.

Diagram 6, the learning component.

The run-time component of the system is the pattern-matcher, which maps input queries to potentially applicable learned rules and then applies these rules in turn until some are found that succeeds, or until it is established that no applicable combination of learned rules exists.

We now examine each of the components in turn.
3.1 The grammar pre-processor

Since the generalizer in this implementation is a kind of Prolog meta-interpreter, the system's grammar must be reduced to a set of Horn-clauses. The gap between the original grammar and an equivalent "clean" version is substantial.

The two major problems stemming from the grammar formalism is its treatment of features and movement. The fundamental feature operation is not unification, but priority merge: movement is handled not by gap features, but rather by unrestricted rules, in which more than one non-terminal can occur on the left-hand side of the rule as well as the right. The first of these problems is solved by exiling all feature manipulation to run-time, preserving the order in which it is executed.

The grammar pre-processor performs the job of converting the grammar used by the system's parser into a pure DCG form, in which the first argument of each non-terminal contains a term encoding its derivation history; the motivation for this additional condition will be apparent in the next section. The only non-trivial part of the process, proved to be dealing with the unrestricted rules. This problem is solved by first representing the unrestricted rules in Pereira's Extrapolation Grammar (XG) format; using the XG compiler from [Pereira 83], it is then straightforward to turn the grammar into pure Horn-clauses. Conceptually, the XG compiler turns the unrestricted grammar into a DCG, where each non-terminal is given an extra pair of arguments (the "extraposition list"), to pass around the additional left-hand constituents. As an example, consider the following typical (and slightly edited) unrestricted rule, intended to cover free relatives like the one in "John mentioned a book yesterday which you should read":

\[
s(2, \text{prm}=1, \text{fpe}(2)) \ & \ \text{temp_advp}(1, \text{tzm}=1, \text{fpe}(1)) \rightarrow \\
\text{temp_advp}(\text{dng}=0) \ & \ s(\text{rel}=1)
\]

The rule reverses the sequence of temporal adverbial and relative clause, in effect transforming the sentence into "John mentioned a book which you should read yesterday". The "2" in the first argument position in the left-hand "s" indicates that its features are to be inherited from those in the second constituent on the right-hand side; "\text{prm}=1" means that the "\text{prm}" feature in the inherited set will if necessary be overridden and set to 1.
This rule is represented in the XG-formalism (again in a slightly edited form) as follows:

\[ s\left(s\text{rule12},T,S,\text{Features}_1,\text{Sem}_1,X_{\text{in}},\right. \\
\left. x\text{nogap},\text{nonterminal},\text{temp_advp}(T,\text{Features}_2,\text{Sem}_2),X_{\text{out}}\right) \rightarrow \\
\text{temp_advp}(T,\text{Features}_3,X_{\text{in}},X_{\text{next}}) \& \\
\{\text{get_feature}(\text{Features}_3,dng,0)\} \& \\
s\left(S,\text{Features}_4,X_{\text{next}},X_{\text{out}}\right) \& \\
\{\text{get_feature}(\text{Features}_4,rel,1) \& \\
\text{put_feature}(\text{Features}_3,\text{prm},1,\text{Features}_1) \& \\
\text{put_feature}(\text{Features}_4,\text{tmp},1,\text{Features}_2)\}. \]

The DCG produced can potentially contain left-recursive rules. However, we shall see in the next section that this causes no problems, since it is not used for normal, unrestricted parsing; the non-terminating branches in the search space can thus never be entered.

### 3.2 The generalizer

The basic idea is first to define the class of operational goals; by this, we mean the goals which will be allowed to appear on the right-hand-side of learned rules. Having done this, a successfully processed example is generalized by (notionally) constructing a derivation tree for it, and then chopping off all the branches rooted in operational goals; the leaves in the new, "generalized" derivation will be the conditions in the learned rule (and thus by construction operational), and the root will be a more general version of the goal corresponding to that in the example. In the simplest (one-level) version of the scheme, operational goals will coincide with lexical ones as in the toy grammar example of section 2.

A slight refinement is to allow non-lexical operational goals, in particular ones corresponding to NP's. The basic method can now be applied recursively, first to the proof tree corresponding to the entire example, and then to each tree rooted in an operational NP goal; in the latter case, the operationality criterion is once again lexical. This results in the acquisition of two sets of rules, corresponding to the two different operationality criteria: the top-level rules construct S's from NP's and lexical items, and the second-level ones construct NP's from lexical items alone.

The generalizer is basically a Prolog meta-interpretor, similar to the one described in [Van Harmelen & Bundy 88]. Generalization is from a computational perspective essentially the parsing of a query with a DCG; this means that care has to be taken to ensure that parsing efficiency is acceptably high, and even more importantly that infinite recursions are not caused by left-recursive grammar rules. Luckily, there is a simple and uniform way to solve this problem, by exploiting the fact that the first argument in each DCG rule, SynTree, has been set up to hold the derivation history. The query is first run through the normal, "dirty" grammar, to find the intended instantiation of the derivation argument; this is then used to guide DCG parser used by the generalizer, effectively making the "parsing" deterministic. The top-level is thus schematically:

\[ \text{learn_rule}(\text{Query},\text{Rule},\text{SubRules}) \leftarrow \]

\[ \ldots \]

\[ \text{lexical_analysis}(\text{Query},\text{Chart}) \& \]

\[ \text{normal_parser}(\text{Chart},\text{sent}(\text{SynTree},\text{Parse})) \& \]

\[ \text{generalize}(\text{sent}(\text{SynTree},\text{Parse}), \]

\[ \text{(sent}(\text{GenSynTree},\text{GeneralizedParse}) \leftarrow \text{Conds}) \]

\[ \text{SubRules}) \& \]

\[ \ldots \]

The generalization itself is performed by the top predicate generalize/3; the code of the generalizer is reproduced and documented in appendix 1.

### 3.3 The simplifier

The purpose of this module is to attempt to reduce the size of learned rules, in particular calls to feature-manipulation primitives; these make up most of the body of typical rules
with on average about 50 calls per rule. The basic mechanism is to take each feature-value, and trace its update history backwards through successive updates. We optimize by:

- Removing get_feature/4 calls which can already be seen at compile-time to succeed. Since learned rules are compositions of normal ones, this case occurs when one component rule calls get_feature for a feature that an earlier component has set by a call to put_feature.
- Removing duplicate copies, when the same get_feature call occurs more than once in the rule.
- Reordering the rule body so that the structure-building calls buildtree/3 and fpearg/1 are moved to end: this ensures that structure will only be built if the rule succeeds.

If features were only used for syntax, it would also be possible to perform a further kind of optimization for S-level rules; having traced each get_feature call back through the chain of put_feature calls ending in the feature set it accesses, we could then remove the put_feature calls altogether. This would represent a very considerable reduction in average rule-size. Unfortunately, this is not possible, since the semantic processing following the parsing stage does occasionally make use of the features.

The following pseudo-code characterizes the simplification algorithm:

**Phase 1**

1. Combine get_feature calls and put_feature calls accessing the same feature set into groups. Replace each group with a corresponding call to get_group or put_group.
2. Collect all calls to buildtree/3 and fpearg/1.

**Phase 2**

Go through the body of the rule, passing an alist of annotations; this is used to replace or simplify calls to get_group. The alist associates with each feature set a history of its derivation. This is one of:

- primitive(Constituent) - the feature set is the one associated with Constituent.
- update_from(Old_features, Update_set) - the feature set was derived from Old_features by the chain of updates Update_set.

For each literal \(L\) in the rule body, do one of the following.

i) If \(L\) is of the form put_group(Old, Updates, New), then add a suitable entry to the alist, constructed from \(L\) and the derivation history of Old.

ii) If \(L\) is of the form get_group(Feature_set, Access_list), replace it with a literal of the form get_group(Original, Access_list_1), where:

   a) Original is the base of the update chain that Feature_set belongs to.

   b) Access_list_1 is derived from Access_list as follows: for each element \(F=V\), if \(F=V_1\) is in the list of updates, unify \(V\) with \(V_1\) and throw away \(F=V\).

iii) If \(L\) is of any other form, keep it unaltered.

**Phase 3**

1. Remove duplicate calls.
2. Re-expand calls to get_group and put_group.
3. Add calls to buildtree/3 and fpearg/1 to the end of the rule body.

**3.4 The rule generator**

If the pattern-matcher is to be able to use the learned rules efficiently, they must be indexed in order to ensure quick access to them. This is what the rule generator does; it
down-loads the learned rules into a decision tree. The decision-tree indexing scheme is implemented almost exactly as described in section 2.2: The decision-tree used here is isomorphic to the trie-structure of the normal grammar of the large-scale natural language query interface to relational data bases. The `tree_arc/3` and `learned_rule/3` predicates of section 2.2 correspond in an obvious way to the

```
  lrt(+NodeNo,+CompType,-NextNode)
  lrt(+NodeNo,+CompType,-w(RuleNo))
```

and

```
  lrtaux(+RuleNo,+Chart,?Start,+SubComps,Comp) <- Conds.
```

predicates used in the learned rule-trie. There are equivalent predicates in the normal grammar.

As described in section 2.2, an `lrtaux/5` clause is added for each learned rule. The `lrt/3` predicates are derived by inspecting the list of conditions, `Conds`, of the rule and an entry is made for each lexical goal or two-level goal in it. The third argument of the last `lrt/3` entry for this learned rule is set to point at the corresponding `lrtaux/5`.

On inserting a rule into the rule-trie, one attempts to use the `lrt/3` clauses already present in it and additional clauses are only added when necessary. Also, if the rule itself is a duplicate of one previously inserted, no extra clauses are added at all.

### 3.5 The pattern-matcher

Since the learned rules acquired by the generalizer in effect comprise a specialized grammar, it would be possible to apply the normal parsing mechanism to them. However, this fails to exploit the grammar's unusually simple structure: the depth of a derivation-tree cannot exceed two, and NP is the only non-lexical category. Thinking about the problem in this way should make the pattern-matcher's construction easy to understand. The rules are compiled into a trie-structure, indexed by constituent category; this can either be "NP", or some lexical category. The pattern-matcher then locates potentially suitable rules by a kind of non-deterministic LR parsing method, driven by the trie-structure and otherwise optimized to exploit the peculiarities of the situation; a well-formed sub-string table is used to remember previously located NP's.

The following pseudo-code characterizes the algorithm. Positions in the input string are marked from `*start*` to `*end*`; `*trie-root*` denotes the root-node of the trie-structure; pointer marks the place we have reached in the input string, trie-node the current position in the rule trie, and `nps` the sequence of NP's so far located between 0 and pointer. We assume that lexical analysis has already been performed, so that we can discover by a suitable look-up operation whether or not there is an item of a given lexical category at a given location in the input string.

Pattern-matching algorithm

1. Set `pointer` to `*start*`. Set `trie-node` to `*trie-root*`.
2. Set `category` to the lexical category of the item at `pointer`.
3. Non-deterministically do one of:
   a) If there is a trie arc from `trie-node` to `next-node` triggering on `category` then set `trie-node` to `next-node`. Bump pointer and go back to 2.
   b) If there is a trie arc from `trie-node` to `next-node` triggering on "NP", and there is an NP from `pointer` to `next-pointer`, set `trie-node` to `next-node`, set `pointer` to `next-pointer`, push the found NP onto `nps`, and go back to 2.
   c) If `pointer = *end*`, and `trie-node` is a leaf of the trie marked with a rule, then try to apply it to the whole input string, if necessary looking up NP's in sequence from `nps`.  


The subroutine for finding NP's is similar, though slightly simpler; the variable and constant names correspond in the obvious way to those in the first algorithm.

To find an NP from pointer to next-pointer:

1. If the well-formed sub-string table records that NP's have been searched for at pointer, pick one non-deterministically and return, else
2. Set NP-pointer to pointer. Set NP-trie-node to *NP-trie-root*.
3. Set NP-category to the lexical category of the item at NP-pointer.
4. Non-deterministically do one of:
   a) Find a trie arc from NP-trie-node to NP-next-node triggering on NP-category. Set NP-trie-node to NP-next-node. Bump NP-pointer and go back to 3.
   b) If there is a reduction rule at NP-trie-node, attempt to apply it to the segment of the input string joining pointer to NP-pointer, and record the result in the well-formed sub-string table. Then return.
   c) If NP-pointer = pointer and there are no alternatives left, record in the well-formed sub-string table that NP's have been searched for at pointer, and return with failure.

4. Results

A proper evaluation of performance gain due to the EBL bypass is impossible without a large statistical sample of typical user interactions with the target system; at this stage of the project, such data is unfortunately not available. Our preliminary performance measurements have been based on a corpus of 31 queries of distinct syntactic type, in length varying between 3 and 14 words; the histogram in diagram 2 summarizes the distribution of the speed-up factor over this set. The speed-up factor was defined as the ratio of EBL look-up to parsing for sentences where an applicable rule existed. It averaged slightly over 30, and as shown in the diagram exceeded 10 on all queries. The average look-up overhead on sentences for which no applicable rule existed was less than 3%. One of the few disappointments of the project was however the poor performance of the simplifier, which was unable to achieve better than an average 20% reduction in rule size; this appeared mainly to be due to the necessity to keep all feature sets for possible later use in semantic interpretation.

**Distribution of Speed-Ups**

<table>
<thead>
<tr>
<th>No of Sentences</th>
<th>Speed-up Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10</td>
<td>8</td>
</tr>
<tr>
<td>10-20</td>
<td>6</td>
</tr>
<tr>
<td>20-30</td>
<td>4</td>
</tr>
<tr>
<td>30-40</td>
<td>4</td>
</tr>
<tr>
<td>40-50</td>
<td>4</td>
</tr>
<tr>
<td>50-60</td>
<td>2</td>
</tr>
<tr>
<td>60-70</td>
<td>2</td>
</tr>
</tbody>
</table>

*Diagram 2. Distribution of speed-ups due to EBL bypassing.*
The following transcript of a short session with the system illustrates the EBL module in action. Input sentences are shown in bold-face, and comments in italics. Note that the glosses for acquired rules are only very approximate, and omit nearly all features.

EBL bypass initialized, no rules.

**Does Iceland export fish?**

*Bypassing.*

*No match.*

*Adding a top level rule.
"S -> does NP TV NP?"*

*Adding 2 second level rules.*

"NP -> Name" and
"NP -> N:[mass=y]"

**Is the Vip Club a small organization?**

*Bypassing.*

*No match.*

*Adding a top level rule.
"S -> is NP NP?"*

*Adding 2 second level rules.*

"NP -> the Name" and
"NP -> DET ADJ N"

**Who is a member of the Vip Club?**

*Bypassing.*

*No match.*

*Adding a top level rule.
"S -> NP:[wh=y] is NP?"*

*Adding 2 second level rules.*

"NP -> PRO" and
"NP -> DET N P DET NAME"

**Is John a citizen of the United States?**

*Bypassing.*

*EBL look-up succeeded.*

The top-level rule used is "S -> is NP NP?", from the second example; the second-level rules are "NP -> Name" from the first example, and "NP -> DET N P DET NAME" from the third.

### 5. Summary and conclusions

On the basis of the experiences from this project, we think there are good reasons to take EBL seriously as a practical and generally applicable way of optimizing NL query systems; the speed-ups achieved were very considerable at a low overhead. Even more importantly, it was possible to apply the EBL method despite the A Large-Scale natural language data base query interface having several characteristics undesirable from this point of view; our a priori guess at the beginning of the project was that, if it worked here, it would work on most systems. We have already implemented a similar module for the Swedish Core Language Engine, a large unification grammar for Swedish, where it proved easy to cover both morphological, syntactic and semantic processing [Samuelsson & Rayner, forthcoming].

One thing that will be studied more is the dependence of access time on the number of learned rules when this number becomes large, that is around a thousand rules. It certainly
seems reasonable to hope that the pattern-matching algorithm presented here will give approximately logarithmic behaviour, but this is really an empirical question, since it depends on the distribution of the common query-types in terms of their lexical categories. Another important question is the extent to which it is possible to compress the generated rules. Since we are essentially trading space for time, this is likely to define the limits of the method, since we will eventually simply run out of space to store more learned rules, even if we can index them efficiently. The perhaps most important question, however, is if one can hope to cover a large portion of the queries actually posed by a user with a realistic set of learned rules. The first and third of these issues will be addressed in following reports from the series "Applying Explanation-Based Learning to Natural Language Processing" by Rayner and Samuelsson.

In conclusion, it seems to us that application of the EBL method to Natural Language offers a fruitful field for continued investigation of both a practical and theoretical nature.

Acknowledgements

This project would have been impossible without the assistance of many people at SICS and IBM Nordic Laboratories. We would in particular like to thank Ivan Bretan, Carl Brown, Jane Brown, Mats Carlsson, Pär Dahlin, Mikael Eriksson, Rune Gustavsson, Per Kristiansson, Sten Örväärn, Roberto Rongione and Mohammad Sanamrad for their help and support.

References


Appendix 1: The code of the generalizer

A call to generalize(+Goal,-Rule,-SubRules) will instantiate Rule to a rule, whose head is a literal which subsumes Goal, and whose body is a list of operational literals. SubRules will be instantiated to a list of rules, one for each literal in the body of Rule that corresponds to a second-level goal. The top predicate generalize/3 calls generalize/4, which in turn calls the two predicates generalize/7. The first of these is the core of the generalizer, and closely resembles the program from [Rayner 88]. The third argument (Level) determines whether the second-level goals are to be treated as primitive or whether they should be expanded as any other non-operational user goal. As can be seen, the last
four arguments together form two difference lists, one for the operational conditions on the main rule, and one for the learned sub-rules.

generalize(Goal,Rule,SubRules) <-
    generalize(Goal,top_level,Rule,SubRules).

generalize(Goal,Status,(G_goal <- Conds),SubRules) <-
    generalize(Goal,G_goal,Level,Conds,[],SubRules,[],[]).

generalize(true,true,*,C,C,S) <- cut.

generalize1((H & T),(G_H & G_T),Level,Conds0,Conds,
    SubRules0,SubRules) <- cut &
    generalize1(H,G_H,Level,Conds0,Conds1,SubRules0,SubRules1,SubRules) &
    generalize1(T,G_T,Level,Conds0,Conds,SubRules1,SubRules).

generalize1((H:T),(G_H,G_T),Level,Conds0,Conds,
    SubRules0,SubRules) <- cut &
    (generalize1(H,G_H,Level,Conds0,Conds,SubRules0,SubRules);
    generalize1(T,G_T,Level,Conds0,Conds,SubRules0,SubRules)).

generalize1(Goal,G_goal,top_level,[G_goal|Conds],Conds,
    [Rule|SubRules],SubRules) <-
    two_level_goal(Goal) & cut &
    call(Goal) &
    generalize(Goal,second_level,Rule,*).

generalize1(Goal,G_goal,*,[G_goal|Out],Out,Rules,Rules) <-
    built_in_goal(Goal) & cut &
    call(Goal).

generalize1(Goal,G_goal,*,[G_goal|Conds],Conds,
    [Rule|SubRules],SubRules) <-
    operational_goal(Goal) & cut &
    call(Goal).

generalize1(Goal,G_goal,Level,Conds0,Conds,SubRules0,SubRules) <-
    non_operational_user_goal(Goal) & cut &
    expand_goal(Goal,G_goal,Body,G_body),
    generalize1(Body,G_body,Level,Conds0,Conds,SubRules0,SubRules).

generalize1(Goal,*,*,*,*,*) <-
    ttnl & ttnl &
    display('*** Error: no way to deal with ') &
    display(Goal),ttnl & cut &
    fail.

expand_goal(Goal,G_goal,Body,G_body) <-
    functor(Goal,P,N) &
    copy_term(Goal,Copy) &
    clause(Copy,* Ref) &
    functor(Head,P,N) &
    clause(Head,Body,Ref) &
    copy_term([Head,Body],[G_head,G_body]) &
    Goal = Head & G_goal = G_head.

Appendix 2: Translation from Quintus to VM-Prolog

The EBL module was developed in SICStus under SUN/UNIX, using a version of the parser of the large-scale natural language query interface to relational data bases, a parser developed by Mats Carlsson and Mikael Eriksson and thus available at SICS [Eriksson 90]. The EBL module (and the parser) were then adapted to Quintus under PS/2, which required only a few trivial changes. Since the large-scale natural language query interface to relational data bases is developed and maintained in VM-Prolog, we were requested to translate the code to VM-prolog, regardless of the fact that it will later be re-transformed automatically to Quintus and actually run in Quintus. In this situation, we chose to emulate
the Quintus specific predicates in VM-Prolog, if there was no obvious one-to-one mapping between predicates in the two languages. The reasons for doing this were:

A) Avoiding messing up the original Quintus code, introducing errors and doing the very same replacements in many places.

B) Facilitate the re-transformation by keeping the Quintus flavour of the code, avoiding VM-Prolog specific predicates or at least limiting the use of them to special files that could be disregarded on re-translation.

In practise, the translation involved editing the code files and writing replacement predicates. In the editor, we performing obvious replacements such as substituting

```prolog
:-
with
<-

and introducing parentheses in else constructions:

(a, b ; c, d)
became

(a & b) ; (c & d).
```

Most replacement predicates were easy to write, for example:

```prolog
q_write(Stream, Term) :- write(Term, Stream).
```

The most time-consuming predicates to translate were those handling IO, where we simulated the stream manipulating predicates of Quintus using the logical file names of VM-Prolog.