Arbeitspapiere des SFB 340
Bericht Nr. 137, Juli 1999

A PARSING ALGORITHM TO REDUCE COPYING IN PROLOG

GERALD PENN

Abstract. This paper presents a new parsing algorithm that reduces the amount of copying required in a memoisation-based Prolog parser for context-free grammars with empty categories to two copies per non-empty edge.

Contents

1. Introduction

Most if not all efficient all-paths parsers for natural language are memoisation-based because of the inherent ambiguity that exists in large-scale natural language grammars. Within WAM-based Prolog, memoisation is a fairly costly operation because, in addition to the cost of copying an edge into the memoisation table, there is the additional cost of copying an edge out of the table onto the heap in order to be used as a premise in further deductions (rule applications), whereas one might suppose that in imperative programming languages, those table entries could be used *in situ*, with a minimal amount of trailing or timestamp updates in the case of categories with subsumption. In practice, textbook bottom-up Prolog parsers copy even more: once for every attempt to match an edge to a daughter category, based on a matching endpoint node, which is usually the first-argument on which the memoisation predicate is indexed.

ISSN 0947-6954/99

Depending on the grammar and the empirical distribution of matching mother/lexical and daughter descriptions, this number could approach $(n-1)^2$ copies for an edge added early to the chart, where n is the length of the input to be parsed.

For context-free grammars themselves, the category information that must be copied is normally quite small in size. For feature-structure-based grammars and other grammars with large categories, however, it becomes quite significant. The ALE system (Carpenter and Penn 1996) attempts to reduce this by using an algorithm due to Carpenter that traverses the string breadth-first, right-to-left, but matches daughters in a rule depth-first, left-to-right in a failure-driven loop, which eliminates the need for active edges and keeps the sizes of the heap and stack small. It still copies a candidate edge every time it tries to match it to a daughter description, however.

While the dream of an as-yet undiscovered algorithm with zero copying overhead may continue, one is presented below that reduces this worst case to two copies per non-empty edge, regardless of the length of the input string or when the edge was added to the chart. Since textbook algorithms require at least two copies per edge as well, assertion and potentially matching the next lexical edge to the left/right, this algorithm achieves the minimum number of copies attainable by them.

2. Empirical Efficiency

Whether this algorithm will outperform textbook methods is largely empirical, however, because:

1. one of the two copies is kept on the heap itself and not released until the end of the parse. For large parses over large data structures, that can increase the size of the heap significantly, and will result in a greater number of cache misses and page swaps.
2. the new algorithm also requires an off-line partial evaluation of the grammar rules that increases the number of rules that must be iterated through at run-time during depth-first closure. This can result in redundant operations being performed among rules and their partially evaluated instances to match daughter categories, unless those rules and their partial evaluations are folded together with disjunctions to share as much compiled code as possible.

My analysis of this algorithm's performance over the HPSG grammar distributed with ALE relative to ALE's previous parser reveals that (2) without (1) (and without code sharing) results in a uniform increase in parse times of approximately 3.7%. Switching then to the new algorithm (1) with no additional indexing results in a uniform decrease in parse times of approximately 1.3%, for a total increase of approximately 2.4% relative to the old parser.[1] It is fairly likely then that better sharing of code between different compiled rules and better indexing of categories in the chart will yield a considerable overall improvement in parsing times relative to the older method on similar grammars.

3. Theoretical Benefits

The principal benefits of this algorithm are that:

1. it reduces copying, as mentioned above.
2. it does not suffer from a problem that textbook algorithms suffer from when running under non-ISO-compatible Prologs, including SICStus Prolog. On such Prologs, asserted empty category edges that can match leftmost daughter descriptions of rules are not able to combine with the outputs of those rules.
3. keeping a copy of the chart on the heap also allows for more sophisticated indexing strategies that would otherwise be overwhelmed by the cost of copying an edge before matching.

4. The Algorithm

In this section, it will be assumed that the CFG in question obeys the following property:

Definition 1. *A context-free grammar, G, is* empty-first-daughter-closed *(*EFD-closed*) iff, for every production rule, $N_0 \longrightarrow N_1 \ldots N_n$ in G, there are no empty productions (empty categories) from non-terminal N_1.*

The next section will show how to transform any CFG into an EFD-closed CFG.

This algorithm, like ALE's algorithm, proceeds breadth-first, right-to-left through the string, at each step applying the grammar rules depth-first, matching daughter categories left-to-right. The first step is then to reverse the input string, and compute its length (performed by `reverse_count/5`) and initialise the chart:

[1] The original ALE parser had a bug that made it incomplete with respect to combining empty categories to form new empty categories. These figures are relative to the old parser after that bug was repaired. The parsing algorithm presented here does not suffer from this bug.

```
rec(Ws,FS) :-
  retractall(edge(_,_,_)),
  reverse_count(Ws,[],WsRev,0,Length),
  CLength is Length - 1,
  functor(Chart,chart,CLength),
  build(WsRev,Length,Chart),
  clause(edge(0,Length,FS),true).
```

Two copies of the chart are used in this presentation. One is represented by a term `chart(E1, ... ,EL)`, where the i^{th} argument holds the list of edges whose left node is i. Edges at the beginning of the chart (left node 0) do not need to be stored in this copy, nor do edges beginning at the end of the chart (specifically, empty categories with left node and right node `Length`). The other copy is kept in a dynamic predicate, `edge/3`, as a textbook chart parser would.

Neither copy of the chart stores empty categories. These are assumed to be available in a separate predicate, `empty_cat/1`. Since the grammar is EFD-closed, no grammar rule can produce a new empty category. Lexical items are assumed to be available in the predicate `lex/2`.

The predicate, `build/3`, actually builds the chart:

```
build([W|Ws],R,Chart):-
  RMinus1 is R - 1,
  (lex(W,FS),
   add_edge(RMinus1,R,FS,Chart)
  ; ( RMinus1 =:= 0 -> true
    ; rebuild_edges(RMinus1,Edges),
      arg(RMinus1,Chart,Edges),
      build(Ws,RMinus1,Chart)
    )
  ).
build([],_,_).
```

The precondition upon each call to `build(Ws,R,Chart)` is that `Chart` contains the complete term-copy of the non-loop edges of the parsing chart from node `R` to the end, while `Ws` contains the (reversed) input string from node `R` to the beginning. Each pass through the first clause of `build/3` then decrements `Right`, and seeds the chart with every category for the lexical item that spans from `RMinus1` to `R`. The predicate, `add_edge/4` actually adds the lexical edge to the `edge/3`-copy of the chart, and then closes the chart depth-first under rule applications in a failure-driven loop. When it has finished, if there are still more lexical items to be added

(RMinus1 is not 0), then build/3 retracts all of the new edges from the edge/3-copy of the chart and adds them to the term-copy.

add_edge/4 matches non-initial edges from either the term-copy of the chart, thus eliminating the need for additional copying of non-empty edges, or from empty_cat/1. Whenever it *adds* an edge, however, it adds it to the edge/3-copy of the chart. This is necessary because add_edge/4 works in a failure-driven loop, and any edges added to the term-copy of the chart would be removed during backtracking:

```
add_edge(Left,Right,FS,Chart):-
  assert(edge(Left,Right,FS)),
  rule(FS,Left,Right,Chart).

rule(FS,L,R,Chart) :-
  (Mother ===> [FS|DtrsRest]),
  match_rest(DtrsRest,R,Chart,Mother,L).

match_rest([],R,Chart,Mother,L) :-
  add_edge(L,R,Mother,Chart).
match_rest([Dtr|DtrsRest],R,Chart,Mother,L) :-
  arg(R,Chart,Edges),
  member(edge(Dtr,NewR),Edges),
  match_rest(DtrsRest,NewR,Chart,Mother,L)
 ; empty_cat(Dtr),
   match_rest(DtrsRest,R,Chart,Mother,L).
```

Note that we never need to be concerned with updating the term-copy of the chart during the operation of add_edge/4 because EFD-closure guarantees that all non-initial daughters must have left nodes strictly greater than the Left passed as the first argument to add_edge/4.

Moving new edges from the edge/3-copy to the term-copy is straightfowardly achieved by rebuild_edges/2:

```
rebuild_edges(Left,Edges) :-
  retract(edge(Left,R,FS)),
  -> Edges = [edge(R,FS)|EdgesRest],
     rebuild_edges(Left,EdgesRest)
   ; Edges = [].
```

The two copies required by this algorithm are thus: 1) copying a new edge to the edge/3-chart by add_edge/4, and 2) copying new edges from the edge/3-chart to the term-chart by rebuild_edges/2.

5. EFD-closure

To convert a context-free grammar to one in which EFD-closure holds, we must partially evaluate those rules for which empty categories could be the first daughter over the available empty categories. If all daughters can be empty categories, then a rule may create new empty categories, over which rules must be partially evaluated, and so on. The closure algorithm is presented here in pseudo-code and assumes the existence of six auxiliary lists:

- `Es` — a list of empty categories over which partial evaluation is to occur,
- `Rs` — a list of rules to be partially evaluated,
- `NEs` — new empty categories, created by partial evaluation (when all daughters have matched empty categories),
- `NRs` — new rules, created by partial evaluation (consisting of a rule to the leftmost daughter of which an empty category has applied, with only its non-leftmost daughters),
- `EAs` — an accumulator of empty categories, and
- `RAs` — an accumulator of rules

```
Initialise Es to the empty categories of the grammar;
initialise Rs to the rules of the input grammar;
initialise the other four lists to [];

loop:
while Es =/= [] do
  for each E in Es do
    for each R in Rs do
      match E against the leftmost unmatched category
        description of R;
      if it does not match, continue;
      if the leftmost category was the rightmost (unary rule),
        then add the new empty category to NEs
      otherwise, add the new rule (with leftmost category
        marked as matched) to NRs;
    od
  od;
  EAs := Es + EAs;
  Rs := Rs + RAs; RAs := [];
  Es := NEs; NEs := [];
od;
```

```
if NRs = [],
 then end: EAs are the closed empty cats, Rs are the closed
           rules
 else
   Es := EAs; EAs := [];
   RAs := Rs; Rs := NRs; NRs := []
   go to loop
```

6. Termination Properties

The parsing algorithm itself always terminates because the leftmost daughter always consumes input. Off-line EFD-closure may not terminate when infinitely many new empty categories can be produced by the production rules. In this case, bottom-up parsing will not terminate at run-time either. The new algorithm is thus neither better nor worse than a textbook bottom-up parser with respect to termination.

References

Carpenter, B. and G. Penn (1996). Compiling typed attribute-value logic grammars. In H. Bunt and M. Tomita (Eds.), *Recent Advances in Parsing Technologies*. Kluwer.