Parallel Parsing for Unification Grammars

Andrew Haas
BBN Labs Inc.
10 Moulton Street
Cambridge, Massachusetts

Abstract

The parsing problem for arbitrary unification grammars is unsolvable We present a class of unification grammars for whcih the parsing problem is solvable and a parallel parsing algorithm for this class of grammars

1. Introduction

Unification grammars have the power of a Turing machine, and one can easily prove this by showing that a unification grammar can simulate any Prolog program It follows that the problem of finding all possible parses of a sentence in a given unification grammar is unsolvable The best we can do is an algorithm that sometimes finds a set of parses and sometimes goes into an infinite loop The top-down, left-to-right parser used with definite clause grammars is of this kind - if the grammar contains left recursion the parser may run forever If we want to write a parallel parsing algorithm for unification grammar, we first need to find a subset of unification grammar for which the parsing problem is solvable Indeed this is the hard part of the problem We shall see that once we have a parsing algorithm, finding the parallelism is straightforward

Part 1 reviews the algorithm of Cocke. Kasami and Younger for parsing context-free grammars in Chomsky normal form This algorithm is beautifully simple and easily extends to a parsing algorithm for unification grammars in Chomsky normal form Therefore the parsing problem is solvable for unification grammars in Chomsky normal form Unfortunately this subset of unification grammar is too restricted to describe human language Part 2 therefore considers an extension of Chomsky Normal Form which allows chain rules - rules having one non-terminal symbol on the right side We generalize the CKY algorithm to handle context-free grammars with chain rules, and extend this algorithm to unification grammars. The extension works only if one places a restriction on the use of chain rules in a unification grammar, and that restriction is one main point of the paper Once we have the parsing algorithm for unification grammars with chain rules, we can easily extend it to unification grammars with any number of symbols on the right side of a rule Finally we consider the possibilities of parallelism in the new parsing algorithm

2. Parsing in Choaaky Normal Form

A context-free grammar in Chomsky normal form contains two kinds of rules. Terminal rules have a single terminal on the right side, branching rules have exactly two non-terminal symbols on the right side Since no rule has an empty right side, no symbol can generate the empty string We use the capital letters A.B.C as variables ranging over non-terminal symbols To describe the substrings of an input sentence we number the spaces between words - 0 is the space before the first word and n is the space after the n-th word If I < j. input[i j] is the string of words between space i and space j The CKY algorithm builds a matrix M such that

```
M[i j] - | A | A »>• input[i j] |
```

Strictly speaking this is a recognizer not a parser, but it is easily extended to a parser, and the same is true for the other algorithms in this paper

If SI and S2 are sets of non-terminal symbols define the product of SI and S2. Si * S2, by

```
S1 • S2 -
  |A | (txist B in SI. C in S2  (A -> B C) is o rul«)|
```

The following lemma shows why this product is useful

Basic Parsing Lemma Suppose that i <. k and for all j such that i <; j < k, M[i j] = J A | A =>• input[i j] { and M[j k] = | A | A =>• input[j k] j Then for all A. A =>» inputji k] iff A is in M[i j] • M[j k] for some j with i < j < k

All proofs are omitted for lfack of space

If i < j < k, M[i j] and M[j k] are shorter than M[i k] Thus the Basic Parsing Lemma allows us to find all possible parses of input[i k], given all possible parses for strings shorter than input[i k] This is the key to writing parsing algorithms that are guaranteed to halt

The CKY algorithm has two parts First the algorithm finds all possible parses for strings of length 1, using the terminal rules Next it considers strings longer than 1 in order of length, it finds all possible parses for input[i k] by applying the Basic Parsing Lemma for all choices of j

Here is a version of the CKY algorithm for a sentence of length N

```
Algorithm 1.

for i from e to (N - 1) do
    M[i  i+1] :-
      | A | (A -> input[i  i+1]) it o rul« |;
for L :- 2 to N do
  for oil i.k such that  i -f L • k do
    M[i  k]  •- union      M[• j] • M[j  k]
                i < j < k
```

Theorem 1 (Correctness of CKY) When the CKY algorithm halts, M[i k] = |A | A =>' inputfi k]{

We turn to unification grammar Consider a first-order language with a simple type system each variable and function letter is assigned a type, and each argument position of each function letter is assigned a

type A tern is well-formed if every argument of each function letter has the correct type for its position We use the capital letters X.Y.Z.W. sometimes with prime marks, as variables ranging over well-formed terras of a unification grammar A unification grammar is a finite set of rules of the form (X -> YI Yn). where X is a well-formed term and YI through Yn are either terms or terminal symbols If G is a unification grammar, the ground grammar G derived from G is the set of well-formed ground instances of rules in G To define the language generated by a ground grammar we use the standard definition of the language generated by a context-free grammar, assuming that the start symbol is always S. It is possible that the ground grammar is infinite, and therefore is not a context-free grammar. This does not create a problem, because the definition of the language generated by a context-free grammar does not rely on the fimteness of the grammar in any way. Finally we define the language generated by a unification grammar G to be the language generated by the ground grammar derived from G.

Consider a trivial example Let the type Person contain the variable p and the constants 1st, 2nd and 3rd Let the type Number contain the variable n and the constants Singular and Plural Suppose NP and VP are function letters with two arguments, a person and a number Then the rule

```
(S) -> (NP :p :n) (VP :p :n)
```

says that a sentence may consist of a noun phrase and a verb phrase that agree in person and number This rule has 6 ground instances, as follows

```
(S) -> (NP 1st Singular) (VP 1st Singular)
(S) -> (NP 2nd Singular) (VP 2nd Singular)
(S) -> (NP 3rd Singular) (VP 3rd Singular)
(S) -> (NP 1st Plural) (VP 1st Plural)
(S) -> (NP 2nd Plural) (VP 2nd Plural)
(S) -> (NP 3rd Plural) (VP 3rd Plural)
```

In this case the ground grammar is finite, so this rule is only an abbreviation for a context-free grammar It appears that most of the syntax of natural language is context-free, therefore unification grammars with finite ground grammars have almost the formal power needed to describe natural language Still there are constructions that cannot be described by context-free grammar The crossed serial dependencies in Dutch and Swiss German are examples (Bresnan 1982) We can describe these constructions using a unification grammar whose ground grammar is infinite As an illustration of the technique we give a grammar for the language (atn)(btn)(ctn). which no context-free grammar can generate. The function letters A, B and C each take a single argument of type Integer. The variable n. the function letter s (for "successor") and the constant 0 are of type Integer, s takes a single argument of type Integer The rules are

```
(S) -> (A :n) (B :n) (C :n)
(A 0) ->
(A (s .n)) -> a (A :n)
(B 0) ->
(B (s .n)) -> b (B :n)
(C 0) ->
(C (s .n)) -> c (C .n)
```

(A n) represents a string of n as, (B n) a string of n b's, and (C n) a string of n C'J The first rule says that a sentence consists of n as, n b's, and n c's

It is important to understand that our parsing algorithm does not construct the ground grammar, nor does it construct the set of ground non-terminals that generate input[i k] Instead it represents sets of ground non-terminals indirectly. If SI is a set of terms in a unification grammar, let (ground SI; be the set of ground instances of terms in Si The parser sets M[i k] to a set of terms so that (ground M[i k]) « | A | A =>• input[i k] |.

A unification grammar in Chomsky normal form has two kinds of rules Terminal rules have a single terminal symbol on the right side, branching rules have two well-formed terms on the right side. Unification is defined for pairs of terms as well as individual terms - the most general unifier of [X Y] and [X Y] is the most general substitution that unifies X with X' and Y with Y . As usual it is neccesary to rename variables before unifying. For this purpose assume that for any term X and integer n, (rename X n) is an alphabetic variant of X, and if m is not equal to n then (rename X n) and (rename Y m) have no variables in common Note that renaming a term or rule does not change its set of ground instances. For this reason we use the term "rule" for any alphabetic variant of a rule in G

If SI and S2 are sets of terms in a unification grammar, the product of SI and S2, Si •' S2, is defined as follows.

```
S1 •' S2 =
{ (s X) | (exist Y' in (rename S1 1),
                Z' in (rename S2 2),
                (X -> Y Z) in (rename G 3),
          s is the most general unifier of
          the pairs [Y Z] and [Y' Z']
}
```

Loosely speaking, this product is like the product used in the CKY algorithm except that instead of testing symbols for equality, it makes the symbols equal by substitution - if possible

Unification Parsing Lemma Suppose that I < k and for all j such that i < j < k, (ground M[i j]) = | A | A =>♦ input[i j] j, and (ground M[j k]) = J A I A =>• input[j k] j. Then A =>♦ input[i k] iff for some j such that I < j < k. A is in (ground M[i j] •' M[j k])

The parser for unification grammars in Chomsky normal form is then as follows

```
for i from 0 to (N - 1) do
    M[i i+1] :=
      { X | (X -> input[i i+1]) is a rule },
for L := 2 to N do
  for all i,k such that i + L = k do
    M[i k] := union  M[i j] •' M[j k]
              i < l < k
```

Theorem. When the unification parser halts, (ground M[k]) ■ |A | A -> mputfi k]j.

Unification grammars in Chomsky normal form are more powerful than context-free grammars, they can generate languages that no context-free grammar can generate, and they can capture generalizations that cannot be captured with context-free grammars Despite this difference in the power of the two formalisms, one can parse these unification grammars using a very straightforward generalization of the CKY algorithm. This is a good example of the power and simplicity of unification.

3. Parting with Chain Rules

Unfortunately one cannot describe natural language syntax with a unification grammar in Chomsky normal form. For example, any intransitive verb can form a sentence by itself - "Run!" and "Stop!" are such sentences. If we restrict ourselves to grammars in Chomsky normal form we must have a rule for each of these sentences - (S -> run), (S -> stop), and so on. The rule we really want is something like (S -> (Vp (2nd) .n)) - a verb phrase that agrees with a second

person subject can form a sentence by itself As a second example, many theories hold that in "Who did Mary like[9]" there is a trace after "like" - a noun phrase with no words under it (NP ->) is not allowed in Chomsky normal form We therefore consider the problem of parsing a unification grammar that allows chain rules - rules with exactly one non-terminal on the right side The solution of this problem includes the essential ideas needed to parse grammars with any number of symbols on the right sides of their rules

Let us first consider the context-free case How can we parse a grammar that allows chain rules, branching rules, and terminal rules[7] We stressed above that the Basic Parsing Lemma is useful because it allows us to find all possible parses of a substring if we are given all possible parses of shorter substrings This allows us to look at substrings in order of length, knowing that at each stage we have the information we need The chain rule (A -> B) tells us that if B =>* input[i k], so does A input[i k] is not shorter than itself, so we could repeat this process indefinitely without ever getting to a longer substring Then what guarantees termination[9]

In the case of context-free grammar, we can easily build a table of all pairs [A B] such that A =>♦ B This table is finite because the set of non-terminals is finite Define (close S) to be the set of all non-terminals A such that A -> B for some B in S Since A =>* A, (close S) contains S

Second Parsing Lemma Let G be a context-free grammar containing only terminal rules, branching rules, and chain rules Suppose that 1 <. k and for all j such that i < j < k. M[i j] =-) A | A =>* input[i j] { and M[j k] = { A | A =>. input[j k] j Then for all A, A =>* input[i k] iff A is in (close M[i j] * M[j k]) for some j with 1 < j < k

Using this observation we extend the CKY algorithm to handle grammars with chain rules

```
Algor i thro 2.
Context-free porter with chain toble

for i froro 0 to (N - 1) do
      M[i I+1] •« (close input[i i+1]).
for L  > 2 to N do
  for all i.k such thot i + L ■ k do
    M[i k] :-  union (close M[i j] * M[j k])
               i < j < k
```

Theorem 3 Suppose the grammar G contains only terminal rules, branching rules and chain rules After algorithm 2 halts, M[i k] =)A | A =>. input[i k](

If one tries to generalize this algorithm to unification grammars with chain rules, a serious problem appears The argument above relies crucially on the finite number of non-terminals in the context-free grammar A unification grammar may generate an infinite ground grammar, so the chain table for a unification grammar might be infinite For example, suppose (f x) -> (f (f .x)) is a rule Then the sequence (f x) -> (f (f x)) -> (f (f (f x))) is an infinite derivation using only chain rules We propose a straightforward solution, we require that for every unification grammar G there is an integer n such that every chain derivation in G is shorter than n

Does this restriction stop us from describing natural languages[9] We claim the answer is no This claim is based on experience - we have written a grammar that is not trivial and contains no chain longer than 4 If this restriction holds, a simple algorithm will generate a chain table for a unification grammar That is. it will generate a finite set C of pairs of terms such that A =>* B in the ground grammar iff [A B] is in (ground C) The method is to construct a series of sets C(k) such that (ground C(k)) ^) [A B] I A =>• B in exactly k steps | Clearly C(I) is the set of pairs [X Y) such that (X -> Y) is a rule If we can construct C(k+1) from C(k), we can build the chain table by constructing C(k) for successive values of k until we find a C(k) that is empty If the grammar contains unbounded chains this algorithm will run forever, and it is up to the grammar writer to fix this In practice this is not likely to be a problem In any case it is better than having the parser go into an infinite loop - which can happen with the depth-first, left-to-right parser used with definite clause grammars

In order to construct C(k) from C(k+1) we define a new product Let SI and S2 be sets of pairs of terms Define

```
S1 •• S2 -
 I (■ [* Z]) | (exist [X Y] in (renome S1  1).
                    [Y* Z] in (renome S2 2).
                s is the most generol unifier
                of Y and Y')
```

I

Lemma (ground SI *• S2) = {[A B] | (exist C [A C] is in (ground SI) and [C B] is in (ground S2)){

Lemma If (ground C(k)) = J[A B]| A =>* B in exactly k steps!, then C(k) •♦ |[X Y] | (X -> Y) is a rulej = }[A B] | A =>• B in exactly k + 1 steps|

In order to extend our second context-free parser to a unification parser, we now define

```
(close* S) ■
  \ (s X) | [X Y] is in (renome Choin 1),
    Y* is in (rename S 2), and s is the mgu of Y and Y' |.
```

We then rewrite the parser by adding "close", just as we did for the CKY algorithm

```
Algorithm 4

for i from 0 to (N - 1) do
 M[i  i+1]  -
  (close* \ X | (X -> input[i  i+1])
                is a rule |).
for L :- 2 to N do
 for all i.k such that i + L - k do
   M[i  k]  -
   union (close* M[i  j] •' M[j  k])
   i < j < k
```

Once again, the proof of soundness and completeness requires only a proof for the corresponding context-free algorithm along with the basic properties of unification

The parser that was actually implemented allows any number of symbols on the right side of a rule. In order to handle rules with an empty right side, we must make another restriction similar to the first one for every grammar G there is an integer n such that every derivation of the empty string in G is shorter than n. In order to handle rules with more than two symbols on the right side, we use dotted rules as described in (Graham 1984) Indeed our parser was inspired by the context-free parser of Graham, Harrison and Ruzzo The implemented parser also uses left context to reject some of the hypotheses that are generated bottom-up, the technique is similar to Shiebers notion of restriction (Shieber 1985)

4. Parallel Parting

We claim that in practice, the time for a unification or a substitution is dependent on the grammar but not on the sentence If the ground grammar is finite, there is a finite bound on the largest term that can be constructed by the parser - it is no larger than the

largest term in the ground grammar Sometimes a grammar will include features that take on an infinite set of values, but these features are few and their values do not get very large for ordinary sentences So to a good approximation, the size of the terms constructed during a parse is independent of the sentence. Therefore the time taken for a unification or a substitution is independent of the sentence

Therefore let U be the maximum time for a unification or substitution Assume that an unlimited number of processors are available and the time to set up tasks for parallel execution is negligible In computing a product SI * S2, we can consider each combination of an X' in SI. a Y' in S2 and a rule (X -> Y Z) in parallel This allows us to compute the product in time 3U, regardless of the size of SI or S2 or the number of branching rules in the grammar. By similar reasoning we can compute (close SI) in time 2U, regardless of the size of SI Then we can compute M[i I+I] for all i in time 2U Suppose we have computed M[i k] for all substrings shorter than L To compute M[i k] for a substring of length L, we can compute M[i j] * M[j k] in parallel for all j such that i < j < k, and then compute the closure This gives a total time of 5U for each length from 1 to the length of the sentence, thus the parser should be able to run in time linear in the length of the sentence

In practice it is essential to remove from M[i k] any terms that are substitution instances of other terms in M[i k] In parsing cerain constructions (for example, sequences of noun modifiers or prepositional phrases), this makes the difference between a matrix of size polynomial in the input, and a matrix of size exponential in the input A simple parallel algorithm will do this in time 2U, the number of processors needed is the square of the number of elements in M[i k] Note that if X is the first term in M[i k] to be computed, it might be a substitution instance of the last term to be computed Thus one cannot be certain that X belongs in M[i k] until all the potential elements of M[i k] have been computed This limits the parallelism in the algorithm There is no such limitation in the case of a context-free parser because an occurence of a non-terminal A in M[i k] is redundant only if there is another occurence of A in M[i k] When the algorithm finds the first occurence of A, it can keep that one and throw away all later occurences

This algorithm achieves speed by wasting processors In many cases we can use left context to show that even though A =>• mput[i k], there is no derivation of the whole sentence in which A is the interpretation for *U*[\ k] For example, suppose our grammar includes a rule (Trace ->) A simple bottom-up parser will conclude that Trace =>• input[i 1] for each 1, but left context will eliminate most of these hypotheses An algorithm that uses left context may therefore manage with fewer processors, but it will not run in linear time in the worst case It must work from left to right, after reading the n-th word it computes M[i n] for all 1 < n If i < j it must compute M[i n] before M[j n] because M[j n] is a proper substring of *U*[\ n] Then the time to read the n-th word is proportional to n, and the total time is 0(nt2)

Of course this worst-case time may not be realized for natural grammars and natural sentences This raises the question of testing the implemented parser. Our current grammar concentrates on clause-level phenomena. There is a large set of subcategorization frames for verbs, and the grammar describes raising, control, passive, subject-aux inversion and wh-movement Noun phrases, adjective phrases and prepositional phrases are described only as much as needed to illustrate the clause-level phenomena. The program was first tested with a Butterfly simulator running on a VAX, it was then moved to a real Butterfly and ran the first time. Tests have shown that the program is reliable but slow, we have not yet attempted to speed it up or to measure the gain from parallelism.

5. Conclusion

We propose to restrict unification grammars by requiring that for each grammar G there is a constant N such that every derivation of a string of length 0 or 1 is shorter than N. Essentially this says that the size of a parse tree is bounded by the size of the sentence; one cannot build arbitrarily large structures that are not realized in the surface string. Given this requirement one can parse unification grammars by an algorithm related to the CKY algorithm. In principle such an algorithm should be able to parse in linear time on an ideal parallel machine. In practice it is probably desirable to use left context, raising the worst case time to 0(nt2) In any case it should be possible to parse in time independent of the size of the grammar Natural sentences are seldom over 50 words, while natural grammars are likely to contain many hundreds of rules, so this is an encouraging result.

references

Bresnan, Joan, Kaplan. Ronald *U.,* Peters, Stanley, and Zaenen, Annie. Crossed Serial Dependencies in Dutch. Linguistic Inquiry, volume 13, number 4, pp 613-635

Graham. Susan L., Harrison, Michael A . and Ruzzo, Walter L An Improved Context-Free Recognizer Transactions on Programming Languages and Systems, volume 2, number 3, pp 415-461.

Shieber, Stuart. Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics, pp 145-152