Bottom-Up Chart Parsing as Constraint Propagation

Frank Morawietz

April 13, 1999

Abstract

In this paper, parsing as deduction and constraint programming are brought together to outline a procedure for automatically generating constraint-based incremental bottom-up chart parsers. Viewing the items of a conventional chart parser as constraints and the constraint base as a chart allows the direct use of the constraint resolution process to parse sentences.

1 Introduction

The parsing-as-deduction approach proposed in (Pereira and Warren; 1983) and the parsing schemata defined as special deduction systems in (Sikkel; 1997) suggest an immediate connection between parsing and constraint programming. In this paper, I view the items of a conventional chart parser as constraints. This allows the automatic derivation of an incremental bottom-up chart parser for an arbitrary context free (CF) grammar in a constraint framework.[1]

There are surprisingly few applications of constraint programming techniques in natural language processing despite an apparent closeness between constraint-based grammar formalisms and constraint systems as used in computer science. This may be due to the fact that computer scientists are interested in solving the combinatorial problem of finding simultaneous variable bindings on tightly constrained domains, whereas linguists are usually interested in the well-formedness of entire structures where variable bindings only serve as the means of identifying objects in that structure.

As usual, a CF grammar is a fourtuple $\mathcal{G} = \langle N, T, I, \Pi \rangle$ of nonterminals, terminals, an initial symbol and a set of productions. As a running example the following CF grammar recognizing the language $\{\varepsilon\} \cup \{a^n b^n \,|\, n \in N\}$ will be used, ε standing for the empty word. Note that I do not follow the standard convention of refering to nonterminals with uppercase letters since those are used to refer to variables.

$$\mathcal{G} = \langle \{s\}, \{a,b\}, \{s\}, \{s \longrightarrow \varepsilon \,|\, asb\} \rangle \qquad (1)$$

Bottom-up chart parsing is a well understood way of recognizing strings. Items used by chart parsers encode well formed substrings by indexing the category recognized, i.e., the terminal or nonterminal symbol labeling a node in the parse tree, with the positions spanned by that category. A chart simply is a set of items recording all recognized substrings. The fundamental rule of bottom-up chart parsing as used in this paper combines n items in the chart to make a new item, $1 \leq p \leq n$, $i_p, j_p \in N$, $s_p \in N \cup T$, $s \in N$:

$$\begin{array}{c} [i_1, j_1, s_1], \ldots, [i_n, j_n, s_n] \Longrightarrow [i_1, j_n, s] \\ \text{if there exists a rule } s \longrightarrow s_1 \ldots s_n \text{ in } \Pi \end{array} \qquad (2)$$

What remains to be defined to implement such a parser is a search strategy, i.e., in which order edges are added to the chart, and how to manage building of the parse tree or chart. We do not go into this here. An easily accessible survey can be found in (Gazdar and Mellish; 1989).

Constraint programming is an elegant, declarative way of solving problems with higher-level programming languages. It represents a generalization of logic programming in the sense that it allows the (logical) specification of a problem on an arbitrary domain and tries to solve it with a combination of inference and search. An introduction to constraint programming can be found in (Marriott and Stuckey; 1998).

There are several constraint programming environments available. The most recent and maybe the most flexible is the Constraint Handling Rules (CHR) package included in SICStus Prolog Intelligent Systems Laboratory (1995); Holzbaur and Frühwirth (1998). All of these systems provide mechanisms for defining and solving constraints.

[1] Although the approach is only presented for CF grammars, it is easily extensible to ID/LP parsing, unification-based parsing or head-driven parsing, see the conclusions.

They maintain a constraint base or store which is continually monitored for possible rule applications, i.e., whether there is enough information present in it to successfully use a rule to simplify constraints or to derive new ones. In case we are dealing with constraint systems over particular domains, the systems provide a mechanism – a set of rules – to solve constraints on that domain.

CHR allow the definition of rule sets for constraint solving with three types of rules:

- simplification rules (symbol <=>) which replace a number of constraints in the store with new constraints;
- propagation rules (symbol ==>) which add new constraints to the store in case a number of constraints is already present; and
- simpagation rules (symbol <=> in combination with a \ in the head of the rule) which replace only those constraints with new ones which are to the right of the backslash.

All types of rules can be named. The name is separated from the actual rule by an @. Furthermore, rules can have guards and pragma declarations. A guard (separated from the rest of the body by a |) is a condition which has to be met before the rule can be applied.[2] A pragma declaration affects the way the rule is compiled. The translation will only introduce the *passive* declaration so we ignore the others. Declaring a constraint in the head passive entails that it will not trigger the rule on its own. An example for a CHR program can be found in Fig. 1 on page 4.

The strength of the CHR approach lies in the fact that it allows multiple heads in a rule, that it is flexible and that it is tightly and transparently integrated into the Prolog engine.

The proposal made in this paper is to directly use the built-in resolution between constraints to parse sentences. Variable bindings are of no importance, i.e., there is no constraint *solving*. The variables we use appear only in the rules and will be ground when they are instantiated with constraints from the constraint store. We translate a CF grammar rule by rule into constraint propagation rules and let the CHR library handle the necessary housekeeping of chart and agenda.

The procedure outlined seems especially interesting for constraint-based systems which use some sort of phrase-structure backbone for parsing, e.g. ALE Carpenter and Penn (1998), to derive a theoretically clean and homogeneous constraint environment for all processing aspects. Since it is easy to implement, it offers itself also for rapid prototyping and demonstration purposes.

The paper proceeds by outlining the translation from CF rules to constraint propagation rules, followed by a description of the implementation and test experiences and a short summary. Although we talk about parsing, what is presented in the paper only constitutes a recognizer unless one also considers a routine which either builds the parse directly in the items or extracts the parse from the chart.

2 The Translation

For the translation from a CF grammar into a constraint framework we have to distingguish two types of rules – those with from those without an empty right-hand side (RHS) – since empty RHSs constitute a problem for bottom-up chart parsing.

The only type of constraints the generated programs will have are called *edge constraints* and correspond to the items used in chart parsing. They have three arguments: the first two are integer valued if non-variable and denote the start and end position of the string covered by the item, the third one denotes the label of the edge, i.e., the terminal or nonterminal symbol recognized.[3]

We treat the easier case of the conversion first. For each rule in the CF grammar with a non-empty RHS we create a constraint propagation rule such that each daughter of the rule introduces an edge constraint in the head of the propagation rule with variable string positions but a fixed label. These edge constraints introduced by the daughters have to be connected via their string positions in the right order, i.e., if a is the leftmost daughter and s the adjacent one to the right, the resulting edge constraints have to use the identical variable in their right and left positional arguments respectively: $edge(A,B,a), edge(B,C,s)$. The new, propagated edge constraint spans the entire range of the positions of the daughters and is labeled with the (nonterminal) symbol of the left-hand-side (LHS) of the CF rule. In our example, the resulting propagation rule looks as follows:

$$\begin{aligned} edge(A,B,a), edge(B,C,s), edge(C,D,b) \\ \Longrightarrow edge(A,D,s) \end{aligned} \quad (3)$$

[2] A more careful distinction separates *ask* and *tell* guards. Ask guards have to be entailed by the constraint store whereas tell guards only have to be consistent with it Saraswat (1993).

[3] In case a parser is wanted, we could include an additional argument here which contains an explicit parse tree.

As usual for bottom-up parsing, the translation is a little bit more complicated for rules with empty RHSs. Basically, we create a propagation rule for each empty rule such that the head is an arbitrary edge, i.e., both positions and the label are arbitrary variables, and post new edge constraints with the LHS of the CF rule as label, using the positional variables and spanning no portion of the string. In our example the following rule results:

$$edge(A,B,Sym) \Longrightarrow edge(A,A,s), edge(B,B,s) \tag{4}$$

But obviously rules of this type lead to nontermination since they would propagate further constraints on their own output. We avoid this by including a guard which ensures that empty edges are only propagated for every possible string position once by testing whether the edge spans a string of length one and we avoid storing and using already existing edge constraints with an absorption rule which removes the newer of two identical entries from the constraint store.[4] An example will be presented in the following section on the implementation of the translation in Fig. 1. Although this is not an elegant solution, it seems that other alternatives such as analyzing the entire grammar or posting the empty constraints while traversing the input string are not appealing either since they give up the one-to-one correspondence between the rules of the CF grammar and the constraint program which is advantageous in debugging.

3 The Implementation

The conversion program takes as input a CF grammar and an initial category and outputs a CHR program which can be used for parsing. The converter has been written in SICStus Prolog Intelligent Systems Laboratory (1995) and compiles CF grammars into constraint programs using the constraint handling rules (CHR) package Frühwirth (1998) introduced in release 3.7 of SICStus Prolog.

The core of the program reads the CF-grammar rule by rule and transforms them according to the translation procedure defined in the previous section. We improve on the performance of the resulting CHR program by using pragma declarations in the rules to make them fire only when all necessary edge constraints have been posted, i.e., the rules are entered only on the appearance of the last head. Obviously, this depends on the order of the traversal of the input string, i.e., if we traverse the string left-to-right, the rule can only apply in case all daughters have been found which means that only the rightmost edge constraint in the head of the rule is not declared to be passive. The same holds for the absorption rule. Furthermore, we optimize the translation for empty rules by testing whether we are dealing with the leftmost edge. Only in that case we post edge constraints for both positional variables. Otherwise we only post one for the right one. Strictly speaking this isn't necessary for correctness since superfluous edge constraints will be absorbed but it improves efficiency.

Additionally, the parser generator program provides a header which loads the CHR library, defines the constraint handler and declares the constraints. At the end of the file we define the parsing predicate which posts the initial constraints according to the given string and reports how many successful parses were found. An example output is shown in Fig. 1.

The CHR code in the figure has three constraint rules. The first one is called `terminate` and absorbs any new occurrence of an already existing empty edge such that there cannot be any further propagation on it. The numbered rules correspond to the rules found in the CF grammar given in (1): the first one to the empty rule, the second one to the non-empty rule. In the second one all but the last edge constraint in the head are declared to be passive. The input to the `parse/1` predicate is given as a list of words which will be traversed in this case from left-to-right. For each element we post the appropriate edge constraint, getting as an additional output the length of the string.[5]

This posting automatically triggers the constraint resolution process. As soon as enough constraints are posted to match the head of a rule, the rule is applied, generating new constraints which might again trigger further rule applications. The functionality to apply the fundamental rule is provided by the CHR library by watching the constraint base, i.e., continually testing whether rules can be applied. Only when no constraint resolution steps can be done any more do we post the next constraint from the input. In our case this corresponds to an incremental left-corner strategy but without the use of any active edges such that the chart – the constraint base – remains small.

Consider for example that we want to parse

[4]The proper and more elegant treatment of this problem would test in the guard of the CHR rule whether the edge constraint to be posted is already entailed by the constraint store. Unfortunately, having a constraint in the guard of a clause in CHR causes that constraint to be posted which is definitely not what is needed.

[5]Note that calling a CHR constraint corresponds to posting it.

```
%%%%%%%%%%%%%%%%%%%% generated file, do not edit %%%%%%%%%%%%%%%%%%%%
%% File:              asb.chr                                       %%
%% created on:        08/01/1999 17:40:14                           %%
%% from source file:  asb                                           %%
%%                                                                  %%
%% with options                                                     %%
%%   unifications are:                Prolog term unifications      %%
%%   initial constraints are posted:  left-to-right                 %%
%%   debugging is switched:           off                           %%
%%   module encapsulation:            off                           %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- use_module(library(chr)).

handler asb.

constraints edge/3.
%% edge(X,Y,Z) means that there is an edge spanning from X to Y     %%
%% labeled Z.                                                       %%

terminate @ edge(B,B,S)#I \ edge(B,B,S) <=> true pragma passive(I).
rule1 @ edge(A,B,_Symbol)  ==>
         B is A + 1 |
         ( A == 1 ->
                 edge(A,A,s), edge(B,B,s)
         ;
                 edge(B,B,s)
         ).
rule2 @ edge(A,B,a)#I, edge(B,C,s)#J, edge(C,D,b) ==>
         edge(A,D,s)
           pragma passive(I), passive(J).

%%%%%%%%%%%%%%%%%%%%%%%%% Parsing predicates %%%%%%%%%%%%%%%%%%%%%%%%%

parse(L) :-
         post_constraints(L,1,Len),
         findall_constraints(edge(1,Len,s),R),
         length(R,ResNum),
         nl,nl,write('Found '),write(ResNum),write(' solutions.'),nl,nl.

post_constraints([],N,N).
post_constraints([H|T],N,Out) :-
         M is N+1,
         edge(N,M,H),
         post_constraints(T,M,Out).
```

Figure 1: The result of the translation of the grammar in Equation (1)

the string *aabb*.[6] We first post the edge constraint $edge(1,2,a)$, which triggers `rule1`, resulting in the two new edges $edge(1,1,s)$ and $edge(2,2,s)$. No more constraint resolution can be done such that the next element from the string is read and the constraint $edge(2,3,a)$ is posted. Jumping ahead, we find ourselves in the situation that the constraint store contains the following edges: $edge(1,2,a)$, $edge(1,1,s)$, $edge(2,2,s)$, $edge(2,3,a)$, $edge(3,3,s)$ and $edge(3,4,b)$. Now `rule2` is triggered and the new edge $edge(2,4,s)$ is propagated. This process continues until no more new edges can be created resulting in the chart given in Fig. 2.

```
edge(1,2,a), edge(1,1,s),
edge(2,2,s), edge(2,3,a),
edge(3,3,s), edge(3,4,b),
edge(4,4,s), edge(2,4,s),
edge(4,5,b), edge(5,5,s),
edge(1,5,s)
```

Figure 2: The final constraint base for `parse([a,a,b,b])`.

Since this contains an edge spanning the entire string which is labeled with the initial symbol – the edge $edge(1,5,s)$ – the input was successfully parsed. These edges are found and extracted into a list by the built-in predicate `findall_constraints/2` in the `parse/1` predicate. Finally we count the number of occurrences of these final edges in the store and report the number of solutions.

The parser generator is a rather trivial program consisting mainly of formatted output statements. Nevertheless, it has a number of switches for the CHR code generation. Trivially, one can switch the debugging option of the CHR program on or off and one can decide whether the CHR handler has to be encapsulated as a module. This is only interesting if the result is to be used in a bigger system. More interestingly, one can decide on the direction of the posting of the initial constraints, i.e., whether one wants a left-to-right or right-to-left traversal of the input. Furthermore, one can decide whether one wants to use Prolog's term unification to find the matching clauses in the constraint base or whether one wants to use an external unifier which then has to be provided in a separate file. Using such an external unifier has both advantages and drawbacks. We come back to both issues when we discuss some of possible extensions and applications.

[6] In this example we ignore the spurious empty edges which are absorbed by the `terminate` rule.

4 Some Experimental Results

We tested the resulting CHR programs by varying the available parameters. We used both the running example and a grammar closer to linguistic purposes (see Fig. 4). The tests were done on a DEC Alpha 3000/300 running SICStus Prolog 3.7.1. We will present absolute timings in seconds since this is useful for comparing them against each other. There is no real measurement implied for the overall speed of the programs.

With the artificial grammar, we tested strings of length 100, i.e., fifty *a*'s and *b*'s. The program given in Fig. 1 needed 11.415 seconds to build a chart containing 251 passive edges and to report that it found one solution.

This slowed down when we used explicit calls to an external unification predicate via the switch of the conversion program. For example the `rule2` of our example program then looks as given in Fig. 3 with `unify/2` simply defined as `unify(X,Y) :- X==Y.`

Now the parse of the same string took 132.541 seconds with the identical number of elements in the chart and of results. So it is eminently profitable to use the built-in head matching of the CHR package. On the other hand, if we want to be able to extend our approach from CF grammars to unification-based grammars we are faced with a choice: either we have to use this external unification procedure and pay the price of slowing things down, or we have to do some very clever manipulation of the structures to be unified such that the problem can be reduced to Prolog's term unification. We have to leave this question open for further research.

The second grammar, see Fig. 4, recognizes PP-attachments to both verb and noun phrases. Its lexicon has two lexical determiners, four adjectives, six nouns, two verbs and one preposition. A valid sentence, among others, is for example: *The ugly man hit the beautiful black dog with a stick.* Clearly the sentence is syntactically ambiguous even if one reading doesn't make much sense.

For a sentence containing nine words (*a man saw a black woman with a telescope*) the resulting parse time is 0.047 seconds with a chart having 36 entries containing two solutions. Again the same change to external unification resulted in a parsing time of 1.043 seconds. We can gain on this by inlining the unification into the guard (but not the head) to achieve a timing of 0.721 seconds. Still, this seems

```
rule2 @ edge(A,B,X)#I, edge(B,C,Y)#J, edge(C,D,Z) ==>
        unify(X,a),
        unify(Y,s),
        unify(Z,b) |
        edge(A,D,s)
     pragma passive(I), passive(J).
```

Figure 3: A rule using calls to external unification

```
np **> det,nb.
nb **> adj,nb.
nb **> nou,pp.
nb **> nou.
pp **> pre,np.
vp **> ver,np.
vp **> ver,np,pp.
s  **> np,vp.
```

Figure 4: The non-lexical rules of the PP-attachment grammar

like a rather long time compared to the original timing.

5 Conclusion

In the paper we presented an automatic generation of constraint programs from CF grammars which can then be used for parsing. The fully declarative programs result in small, incremental and flexible parsers. In a certain sense the approach complements the built-in Definite Clause Grammars as it provides bottom-up execution of CF grammars. The approach seems to be especially suited to rapid prototyping and demonstrations since it does not need an agenda or explicit tabling functions while it may be lacking in efficiency compared to highly specialized hand-crafted parsers. But the closeness and tight integration of the CHR library into the core of SICStus Prolog ensures the efficiency of the constraint processing. Furthermore, there is no need for active edges since we work bottom-up and thereby achieve the behaviour of a shift-reduce parser which waits until all the information is present before applying a rule. We can easily adapt it to alternative strategies by altering the parser generator such that the constraints are posted in a different order. Posting the constraints corresponding to the input in reverse ensures a rightmost strategy. The benefit being that we always know when a rule has to apply since all categories to its right have to have been created already.

A number of possible optimizations or extensions suggest themselves. By using the switch to call external unification programs we can easily extend the approach to work with unification-based grammars such that inclusion in larger constraint (logic) programs becomes well defined. Inclusion of ordering statements of the daughters in the guard would achieve an ID/LP distinction. It seems likely that speed-ups of the parsers can be achieved by doing a more sophisticated analysis of the grammar, e.g., one could compile a link relation which ensures that empty categories can only appear as sisters of certain other categories, not everywhere as has to be assumed at the moment, or by using top-down guidance of the parsing process, e.g., one could post target categories while traversing the input and have more rules for predictions.

References

Carpenter, B. and Penn, G. (1998). ALE: The attribute logic engine, version 3.1, *User manual*, Laboratory for Computational Linguistics, Philosophy Department, Carnegie Mellon University, Pittsburgh, PA 15213.

Frühwirth, T. (1998). Theory and practice of constraint handling rules, *Journal of Logic Programming* **37**(1–3): 95–138. Special Issue on Constraint Logic Programming.

Gazdar, G. and Mellish, C. S. (1989). *Natural Language Processing in Prolog*, Addison–Wesley.

Holzbaur, C. and Frühwirth, T. (1998). Compiling constraint handling rules, *ERCIM/COMPULOG Workshop on Constraints*, CWI, Amsterdam, The Netherlands.

Intelligent Systems Laboratory (1995). SICStus Prolog user's manual, *ISBN 91-630-3648-7*, Swedish Institute of Computer Science.

Marriott, K. and Stuckey, P. J. (1998). *Programming with Constraints*, MIT Press.

Pereira, F. C. N. and Warren, D. H. D. (1983). Parsing as deduction, *ACL Proceedings, 21st Annual Meeting*, Vol. 13.

Saraswat, V. A. (1993). *Concurrent Constraint Programming*, MIT Press, Cambridge, Massachusetts.

Sikkel, K. (1997). *Parsing Schemata: A Framework for Specification and Analysis of Parsing Algorithms*, ETACS Series: Texts in Theoretical Computer Science, Springer.