Chart Parsing and Constraint Programming

Frank Morawietz
Seminar für Sprachwissenschaft
Universität Tübingen
Wilhelmstr. 113
72074 Tübingen, Germany
frank@sfs.nphil.uni-tuebingen.de

Abstract

In this paper, parsing-as-deduction and constraint programming are brought together to outline a procedure for the specification of constraint-based chart parsers. Following the proposal in Shieber et al. (1995), we show how to directly realize the inference rules for deductive parsers as Constraint Handling Rules (Frühwirth, 1998) by viewing the items of a chart parser as constraints and the constraint base as a chart. This allows the direct use of constraint resolution to parse sentences.

1 Introduction

The parsing-as-deduction approach proposed in Pereira and Warren (1983) and extended in Shieber et al. (1995) and the parsing schemata defined in Sikkel (1997) are well established parsing paradigms in computational linguistics. Their main strengths are their flexibility and the level of abstraction concerning control information inherent in parsing algorithms. Furthermore, they are easily extensible to more complex formalisms, e.g., augmented phrase structure rules or the ID/LP format.

Constraint Programming (CP) has been used in computational linguistics in several areas, for example in (typed) feature-based systems (Smolka, 1995), or conditional constraints (Matiasek, 1994), or advanced compilation techniques (Götz and Meurers, 1997) or specialized constraint solvers (Manandhar, 1994). But none of these approaches uses constraint programming techniques to implement standard chart parsing algorithms directly in a constraint system.

In this paper, I will bring these two paradigms together by showing how to implement algorithms from the parsing-as-deduction scheme by viewing the parsing process as constraint propagation.

The core idea is that the items of a conventional chart parser are constraints on labeled links between the words and positions of an input string. Then the inference rules allow for the deduction of new constraints, again labeled and spanning parts of the input string, via constraint propagation. The resulting constraint store represents the chart which can be accessed to determine whether the parse was successful or to reconstruct a parse tree.

While this may seem a trivial observation, it is not just another way of implementing deductive parsing in yet another language. The approach allows for a rapid and very flexible but at the same time uniform method of implementation of all kinds of parsing algorithms (for constraint-based theories). The goal is not necessarily to build the fastest parser, but rather to build – for an arbitrary algorithm – a parser fast and perspicuously. For example, the advantage of our approach compared to the one proposed in Shieber et al. (1995) is that we do not have to design a special deduction engine and we do not have to handle chart and agenda explicitly. Furthermore, the process can be used in any constraint-based formalism which allows for constraint propagation and therefore can be easily integrated into existing applications.

The paper proceeds by reviewing the parsing-as-deduction approach and a particular way of implementing constraint systems, Constraint Handling Rules (CHR) as presented in Frühwirth (1998). Then it shows how to implement several parsing algorithms very naturally with constraint propagation rules before concluding with an outlook on how to extend the technique to more advanced applications.

1.1 Parsing as Deduction

Although I assume some familiarity with parsing-as-deduction, I will recall some basic definitions for convenience. The notations and the three basic algorithms are directly taken from Shieber et al. (1995).

As usual, strings w result from concatenation of symbols from some alphabet set Σ, i.e., $w \in \Sigma^*$. We refer to the decomposition of such a string into its alphabet symbols with indices. We fix this notation using $w = w_1 \dots w_n$. Further notational conventions are: $i, j \in \mathbb{N}$, n for the length of the string to be parsed, $A, B, C, \dots$ for arbitrary formulas or nonterminals, $a, b, c, \dots$ for terminals, ε for the empty string and $\alpha, \beta, \gamma, \dots$ for strings of terminals and nonterminals. Formulas used in parsing will also be called items or edges. A *grammatical deduction system* or, in Sikkel's terminology a *parsing schema*, is defined as a set of deduction schemes and a set of axioms. These are given with the help of *formula schemata* which contain (syntactic) meta-variables which are instantiated with concrete terms on application of the rules. A deduction scheme R has the general form

$$\frac{A_1 \ \dots \ A_n}{C} \ \langle \text{ side conditions on } A_1 \dots A_n, C \ \rangle$$

Table 1: Parsing algorithms as Grammatical Deduction Systems

	Bottom-Up	**Top-Down**	**Earley**
Items	$[j,\alpha\bullet]$	$[j,\bullet\beta]$	$[i,j,A,\alpha\bullet\beta]$
Axiom	$[0,\bullet]$	$[0,\bullet S]$	$[0,0,S',\bullet S]$
Goal	$[n,S\bullet]$	$[n,\bullet]$	$[0,n,S',S\bullet]$
Scan	$\dfrac{[j,\alpha\bullet]}{[j+1,\alpha w_{j+1}\bullet]}$	$\dfrac{[j,\bullet w_{j+1}\beta]}{[j+1,\bullet\beta]}$	$\dfrac{[i,j,A,\alpha\bullet w_{j+1}\beta]}{[i,j+1,A,\alpha w_{j+1}\bullet\beta]}$
Predict		$\dfrac{[j,\bullet B\beta]}{[j,\bullet\gamma\beta]}\langle B\longrightarrow\gamma\rangle$	$\dfrac{[i,j,A,\alpha\bullet B\beta]}{[j,j,B,\bullet\gamma]}\langle B\longrightarrow\gamma\rangle$
Complete	$\dfrac{[j,\alpha\gamma\bullet]}{[j,\alpha B\bullet]}\langle B\longrightarrow\gamma\rangle$		$\dfrac{[i,k,A,\alpha\bullet B\beta]\ [k,j,B,\gamma\bullet]}{[i,j,A,\alpha B\bullet\beta]}$

where the A_i and C are formula schemata. The A_i are called antecedents and C the consequence. Note that deduction schemes may refer to string positions, i.e., the indices within the input string, in their side conditions. Application of these schemata and derivations of formulas are then defined as in the Shieber et al. article. Intuitively, parsing uses the deductive rules – if their antecedents and the side conditions are met – to infer new items from the axioms and already generated items until no new ones can be derived. The parse was successful if a goal item was derived.

Therefore, all the parsing systems used in this paper are defined by specifying a class of items, a set of axioms, a set of inference rules and a subset of the items, the goals. For better readability, I follow Shieber et al. in using the familiar dotted items for the presentation. The three classical example algorithms we will use to illustrate our technique are given in Tab. 1. I assume familiarity with these algorithms.

Unless specified differently, we assume that we are given a context-free grammar $G = \langle N, \Sigma, S, P\rangle$ with nonterminals N, terminals Σ, start symbol S and set of productions P. For Earley's algorithm we also assume a new start symbol S' which is not in N. Each production is of the form $A \longrightarrow \alpha$ with $A \in N, \alpha \in (N \cup \Sigma)^*$. For examples I will use the simple PP-attachment grammar G given in Fig. 1 with the obvious sets of nonterminals and terminals, the start symbol S and productions P. It is left to the reader to calculate example derivations for the three algorithms for a sentence such as *John hit the dog with the stick*.

S	$\longrightarrow$	NP VP	V	$\longrightarrow$	*hit*
VP	$\longrightarrow$	V NP | V NP PP	PN	$\longrightarrow$	*John*
PP	$\longrightarrow$	P NP	N	$\longrightarrow$	*dog* | *stick*
NP	$\longrightarrow$	PN | Det N1	P	$\longrightarrow$	*with*
N1	$\longrightarrow$	N | N1 PP	Det	$\longrightarrow$	*the*

Figure 1: Example Grammar: PP-attachment

1.2 Constraint Handling Rules

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 (Frühwirth, 1998). These systems maintain a constraint base or store which is continually monitored for possible rule applications, i.e., whether there is enough information present to successfully use a rule to simplify constraints or to derive new constraints. Whereas usually one deals with a fixed constraint domain and a specialized solver, CHR is an extension of the Prolog language which allows for the specification of user-defined constraints and arbitrary solvers. The strength of the CHR approach lies in the fact that it allows for multiple (conjunctively interpreted) heads in rules, that it is flexible and that it is tightly and transparently integrated into the Prolog engine.

In CHR constraints are just distinguished sets of (atomic) formulas. CHR allow the definition of rule sets for constraint solving with three types of rules: Firstly simplification rules (<=>) which replace a number of constraints in the store with new constraints; secondly propagation rules (==>) which add new constraints to the store in case a number of constraints is already present; and thirdly "simpagation" rules (<=> 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. Rules can have guards. 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.

We cannot go into the details of the formal semantics of CHR here. The interested reader is referred to Frühwirth (1998). Since I will refer back to it let us just note that logically, simplification rules are equivalences and propagation rules are implications if their guard is satisfied. Simpagation rules are special cases of simplification rules. Soundness and completeness results for CHR are available (Abdennadher et al., 1996 Abdennadher, 1998).

2 Parsing as Constraint Propagation

The basic observation which turns parsing-as-deduction into constraint propagation is simple: items of a chart parser are just special formulas which are used in an inference process. Since constraints in constraint programming are nothing but atomic formulas and constraint handling rules nothing but inference rules, the connection is immediate.

In more detail, I will present in this section how to implement the three parsing algorithms given in Tab. 1 in CHR and discuss the advantages and drawbacks of this approach. Since CHR are integrated in SICStus Prolog, I will present constraints and rules in Prolog notation.

We use the following two types of constraints. The constraints corresponding to the items will be called `edge` constraints. They have two arguments in case of the two naive algorithms and five in the case of Earley's algorithm, i.e., `edge(X,N)` means in the case of the bottom-up algorithm that we have recognized a list of categories `X` up to position `N`, in the case of the top-down algorithm that we are looking for a list of categories `X` starting at position `N` and in the case of Earley's algorithm `edge(A,Alpha,Beta,I,J)` means that we found a substring from `I` to `J` by recognizing the list of categories `Alpha`, but we are still looking for a list of categories `Beta` to yield category `A`. The second constraint, `word(Pos,Cat-Word)`, is used in the scanning steps. It avoids using lexical entries in prediction/completion since in grammar rules we do not use *words* but their *categories*.

For simplicity, a grammar is given as Prolog facts: lexical items as `lex(Word,Category)` and grammar rules as `rule(RHS,LHS)` where `RHS` is a list of categories representing the right hand side and `LHS` is a single category representing the left hand side of the rule.

The algorithms are simple to implement by specifying the inference rules as constraint propagation rules, the axioms and the goal items as constraints. The inference rules are translated into CHR in the following way: The antecedents are transformed into constraints appearing in the head of the propagation rules, the side conditions into the guard and the consequence is posted in the body. A summarization of the resulting CHR programs is presented in Tab. 2.

We use Earley's algorithm for a closer look at the CHR propagation rules. In the scanning step, we can move the head of the list of categories we are looking for to those we already recognized in case we have an appropriately matching edge and word constraint in our constraint store. The result is posted as a new edge constraint

```
parse(InList):-
   axiom,
   post_const(InList, 0, Length),
   report(Length).

post_const([], Len, Len).
post_const([Word|Str], InLen, Len):-
   findall(Cat, lex(Word,Cat), Cats),
   post_words(Cats, InLen, Word),
   NewLen is InLen + 1,
   post_const(Str, NewLen, Len).
```

Figure 2: Utilities for CHR-based deductive parsing

with the positional index appropriately incremented.

The prediction step is more complex. There is only one head in a rule, namely an edge which is still looking for some category to be found. If one can find rules with a matching LHS, we collect all of them in a list and post the appropriate fresh edge constraints for each element of that list with the predicate `post_ea_edges/3` which posts edges of the following kind: `edge(LHS,[],RHS,J,J)`. The collection of all matching rules in a call to `setof/3` is necessary since CHR are a committed choice language. One cannot enumerate all solutions via backtracking. If there are no matching rules, i.e., the list of RHSs we found is empty, the call to `setof` in the guard will fail and therefore avoid vacuous predictions and nontermination of the predictor.

Lastly, the completion step is a pure propagation rule which translates literally. The two antecedents are in the head and the consequence in the body with appropriate instantiations of the positional variables and with the movement of the category recognized by the passive edge from the categories to be found to those found.

In the table there is one more type of rule, called an absorption rule. It discovers those cases where we posted an edge constraint which is already present in the chart and simply absorbs the newly created one.

Note that we do not have to specify how to insert edges into either chart or agenda. The chart and the agenda are represented by the constraint store and therefore built-in. Neither do we need a specialized deduction engine as was necessary for the implementation described in Shieber et al. In fact, the utilities needed are extremely simple, see Fig. 2.

All we have to do for parsing (`parse/1`) is to post the axiom[1] and on traversal of the input string to post the word constraints according to the lexicon of the given grammar. Then the constraint resolution process with the inference rules will automatically build a complete chart. The call to `report/1` will just determine whether there is an appropriate edge with the correct length in the chart and print that information to the screen.

Coming back to the issues of chart and agenda: the constraint store functions as chart and agenda at the same

[1] `axiom/0` just posts the edge(s) defined in Tab. 2.

Table 2: Parsing systems as CHR programs

	Bottom-Up	**Top-Down**	**Earley**
Items	`edge(X,N)`	`edge(X,N)`	`edge(A,Alpha,Beta,I,J)`
Axiom	`edge([],0)`	`edge([s],0)`	`edge(sprime,[],[s],0,0)`
Goal	`edge([s],Len)`	`edge([],Len)`	`edge(sprime,[s],[],0,Len)`

	Scan
Bottom-Up	`edge(Stack,N), word(N,Cat-_Word) ==>`
	`N1 is N+1,`
	`edge([Cat|Stack],N1).`
Top-Down	`edge([Cat|T],N), word(N,Cat-_Word) ==>`
	`N1 is N+1,`
	`edge(T,N1).`
Earley	`edge(A,Alpha,[Cat|Beta],I,J), word(J,Cat-_Word) ==>`
	`J1 is J+1,`
	`edge(A,[Cat|Alpha],Beta,I,J1).`

	Predict
Top-Down	`edge([LHS|T],N) ==>`
	`setof(RHS, rule(RHS,LHS), List) |`
	`post_td_edges(List,T,N).`
Earley	`edge(_A,_Alpha,[B|_Beta],_I,J) ==>`
	`setof(Gamma, rule(Gamma,B), List) |`
	`post_ea_edges(List,B,J).`

	Complete
Bottom-Up	`edge(Stack,N) ==>`
	`setof(Rest-LHS, split(Stack,Rest,LHS), List) |`
	`post_bu_edges(List,N).`
Earley	`edge(A,Alpha,[B|Beta],I,K), edge(B,Gamma,[],K,J) ==>`
	`edge(A,[B|Alpha],Beta,I,J).`

	Absorb
Bottom-Up	`edge(L,N) \ edge(L,N) <=> true.`
Top-Down	`edge(L,N) \ edge(L,N) <=> true.`
Earley	`edge(A,Alpha,Beta,I,J) \ edge(A,Alpha,Beta,I,J) <=> true.`

time since as soon as a constraint is added all rules are tried for applicability. If none apply, the edge will remain dormant until another constraint is added which triggers a rule together with it.[2] So, the parser works incrementally by recursively trying all possible inferences for each constraint added to the store before continuing with the posting of new constraints from the `post_const/3` predicate. The way this predicate works is to traverse the string from left-to-right. It is trivial to alter the predicate to post the constraints from right-to-left or any arbitrary order chosen. This can be used to easily test different parsing strategies.

The testing for applicability of new rules also has a

[2] Another way to "wake" a constraint is to instantiate any of its variables in which case it will be matched against the rules again. Since all our constraints are ground, this does not play a role here.

```
| ?- parse([john, hit, the, dog,
           with, the, stick]).

Input recognized.

word(0,pn-john),
word(1,v-hit),
word(2,det-the),
word(3,n-dog),
word(4,p-with),
word(5,det-the),
word(6,n-stick),
edge(sprime,[],[s],0,0),
edge(s,[],[np,vp],0,0),
edge(np,[],[det,nbar],0,0),
edge(np,[],[pn],0,0),
edge(np,[pn],[],0,1),
edge(s,[np],[vp],0,1),
edge(vp,[],[v,np],1,1),
edge(vp,[],[v,np,pp],1,1),
......
edge(s,[vp,np],[],0,7),
edge(sprime,[s],[],0,7)
```

Figure 3: A partial CHR generated chart

connection with the absorption rules. We absorb the newer edge since we can assume that all possible propagations have been done with the old identical edge constraint so that we can safely throw the other one away.

As an example for the resulting chart, part of the output of an Earley-parse for *John hit the dog with the stick* assuming the grammar from Fig. 1 is presented in Fig. 3.

The entire constraint store is printed to the screen after the constraint resolution process stops. The order of the constraints actually reflects the order of the construction of the edges, i.e., the chart constitutes a trace of the parse at the same time. Although the given string was ambiguous, only a single solution is visible in the chart. This is due to the fact that we only did recognition. No explicit parse was built which could have differentiated between the two solutions. It is an easy exercise to either write a predicate to extract all possible parses from the chart or to alter the edges in such a way that an explicit parse tree is built during parsing.

By using a built-in deduction engine, one gives up control of its efficiency. As it turns out, this CHR-based approach is slower than the specialized engine developed and provided by Shieber et al. by about a factor of 2, e.g., for a six word sentence and a simple grammar the parsing time increased from 0.01 seconds to 0.02 seconds on a LINUX PC (Dual Pentium II with 400MHz) running SICStus Prolog. This factor was preserved under 5 and 500 repetitions of the same parse. However, speed was not the main issue in developing this setup, but rather simplicity and ease of implementation.

To sum up this section, the advantages of the approach lie in its flexibility and its availability for rapid prototyping of different parsing algorithms. While we used the basic examples from the Shieber et al. article, one can also implement all the different deduction schemes from Sikkel (1997). This also includes advanced algorithms such as left-corner or head-corner parsing, the refined Earley-algorithm proposed by Graham et al. (1980), or (unification-based) ID/LP parsing as defined in Morawietz (1995), or any improved version of any of these. Furthermore, because of the logical semantics of CHR with their soundness and completeness, all correctness and soundness proofs for the algorithms can be directly applied to this constraint propagation proposal. The main disadvantage of the proposed approach certainly lies in its apparent lack of efficiency. One way to address this problem is discussed in the next section.

3 Extensions of the Basic Technique

There are two directions the extensions of the presented technique of CHR parsing might take. Firstly, one might consider parsing of more complicated grammars compared to the CF ones which were assumed so far. Following Shieber et al., one can consider unification-based grammars or tree adjoining grammars. Since I think that the previous sections showed that the Shieber et al. approach is transferable in general, the results they present are applicable here as well.[3] Instead, I want to consider parsing of minimalist grammars (Chomsky, 1995) as defined in recent work by Stabler (1997, 1999).[4]

3.1 Minimalist Parsing

We cannot cover the theory behind derivational minimalism as presented in Stabler's papers in any detail. Very briefly, lexical items are combined with each other by a binary operation *merge* which is triggered by the availability of an appropriate pair of clashing features, here noted as `cat(C)` for Stabler's categories `c` and `comp(C)` for `=c`. Furthermore, there is a unary operation *move* which, again on the availability of a pair of clashing features (e.g., `-case`, `+case`), triggers the extraction of a (possibly trivial) subtree and its merging in at the root node. On completion of these operations the clashing feature pairs are removed. The lexical items are of the form of linked sequences of trees. Accessibility of features is defined via a given order on the nodes in this chain of trees. A parse is acceptable if all features have been checked, apart from one category feature which spans the length of the string. The actual algorithm works naively bottom-up and, since the operations are at most binary, the algorithm is CYK-based.

[3] Obviously, using unification will introduce additional complexity, but no change of the basic method is required. If the unification can be reduced to Prolog unification, it can stay in the head of the rule(s). If it needs dedicated unification algorithms, they have to be called explicitly in the guard.

[4] The code for the original implementation underlying the paper was kindly provided by Ed Stabler. Apart from the implementation in CHR, all the rest is his work and his ideas.

Table 3: Part of a CHR-based minimalist parser

Items	`edge(I, J, Chain, Chains)`	
Axiom	`edge(I, I, Chain, Chains)`	$\langle$`[Chain|Chains]` $\longrightarrow \varepsilon\rangle$, `I` a variable
	`edge(I, I+1, Chain, Chains)`	$\langle$`[Chain|Chains]` $\longrightarrow w_{i+1}\rangle$ and there is no `-X` in `[Chain|Chains]`
	`edge(J, J, Chain, Chains)`	$\langle$`[Chain|Chains]` $\longrightarrow w_{i+1}\rangle$, `J` a variable and `-X`, `I` and `I+1` occur in `[Chain|Chains]`
Goal	`edge(0, Length, [cat(C)], [])`	

Merge
```
edge(I,J,[comp(X)|RestHead],Ch0), edge(K,L,[cat(X)|RestComp],Ch1) ==>
   check(RestHead,NewHead,I,J,K,L,A,B,RestComp,Ch0,Ch1,Ch) |
       edge(A,B,NewHead,Ch).
```

An initial edge or axiom in this minimalist parsing system cannot simply be assumed to cover only the part of the string where it was found since it could have been the result of a move. So the elements of the lexicon which will have to be moved (they contain a movement trigger `-X`) actually have the positional indices instantiated in the last of those features appearing. All other movement triggers and the position it will be base generated are assumed to be traces and therefore empty. Their positional markers are identical variables, i.e., they span no portion of the string and one does not know their value at the moment of the construction of the axioms. They have to be instantiated during the minimalist parse.

Consider the set of items as defined by the axioms, see Tab. 3. The general form of the items is such that we have the indices first, then we separate the chain of trees into the first one and the remaining ones for better access. As an example for the actual edges and to illustrate the discussion about the possibly variable string positions in the edges, consider the lexical item *it* (as in *believe it*):

```
lex(it,I,[(K,K)=[cat(d),-case(I,J)]]):-
      J is I+1.
```

Since `I` $= 1$ in the example the following edge results `edge(K, K, [cat(d),-case(1,2)]], [])`. We know that *it* has been moved to cover positions `1` to `2`, but we do not know (yet) where it was base generated.

We cannot go into any further detail how the actual parser works. Nevertheless, the propagation rule for merging complementizers shown in Tab. 3 demonstrates how easily one can implement parsers for more advanced types of grammars.[5]

[5]The predicate `check` implements the checking of the features and indices and recomputes the new values for the resulting edge.

3.2 Compiling the Grammar Rules into the Inference Rules

A proposal for improving the approach consists in moving the test for rule applicability from the guards into the heads of the CHR rules. One can translate a given context-free grammar under a given set of inference rules into a CHR program which contains constraint propagation rules for each grammar rule, thereby making the processing more efficient. For simplicity, we discuss only the case of bottom-up parsing.

For the translation from a CF grammar into a constraint framework we have to distinguish two types of rules: those with from those without an empty RHS. We treat the trivial 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, but appropriately matching string positions and a fixed label. The new, propagated edge constraint spans the entire range of the positions of the daughters and is labeled with the (nonterminal) symbol of the LHS of the CF rule. In our example, the resulting propagation rule for S looks as follows:

```
edge(I,K,np), edge(K,J,vp) ==> edge(I,J,s)
```

The translation is a little bit more complicated for rules with empty RHSs. Basically, we create a propagation rule for each empty rule, e.g., `A` $\longrightarrow \varepsilon$, 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, resulting in CHR rules of the following type:

```
edge(I,J,_Sym) ==> J is I+1 |
    edge(I,I,A), edge(J,J,A)
```

But obviously rules of this type lead to nontermination since they would propagate further constraints on their own output which is avoided 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. Recall that storing and using already existing edge constraints is avoided with an absorption rule. Since these empty constraints can be reused an arbitrary number of times, we get the desired effect without having to fear nontermination. Although this is not an elegant solution, it seems that other alternatives such as analyzing and transforming 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.

With this technique, the parsing times achieved were better by a factor of a third compared to the Shieber et al. implementation. Although now the process of the compilation obscures the direct connection between parsing-as-deduction and constraint propagation somewhat, the increase in speed makes it a worthwhile exercise.

4 Conclusion

In the paper, the similarity between parsing-as-deduction and constraint propagation is used to propose a flexible and simple system which is easy to implement and therefore offers itself as a testbed for different parsing strategies (such as top-down or bottom-up), for varying modes of processing (such as left-to-right or right-to-left) or for different types of grammars (such as for example minimalist grammars). Compared to the Shieber approach, the pure version seems to be lacking in efficiency. This can be remedied by providing an automatic compilation into more efficient specialized parsers.

While the paper has shown that existing constraint systems are powerful enough to allow chart parsing, more work has to be invested in the realization of such a larger system combining these techniques with constraint solvers for existing constraint-based natural language theories to see whether further benefits can be gotten from using parsing as constraint propagation. Due to the flexibility of the CHR system, one can now use the constraint propagation approach to drive other constraint solving or constraint resolution techniques (also implemented in CHR) resulting in a homogenous environment which combines both classical constraint solving with a more operational generator.

Specifically, one can use each created edge to post other constraints, for example about the well-formedness of associated typed feature structures. By posting them, they become available for other constraint handling rules. In particular, systems directly implementing HPSG seem to suffer from the problem how to drive the constraint resolution process efficiently. Some systems, as for example ALE (Carpenter and Penn, 1998) use a phrase structure backbone to drive the process. The proposal here would allow to use the ID/LP schemata directly as constraints, but nevertheless as the driving force behind the other constraint satisfaction techniques. However, for the moment this remains speculative.

References

Abdennadher, S. (1998). *Analyse von regelbasierten Constraintlösern*, PhD thesis, Ludwig-Maximilians-Universität München.

Abdennadher, S., Frühwirth, T. and Meuss, H. (1996). On confluence of constraint handling rules, *LNCS* **1118**, Springer.

Carpenter, B. and Penn, G. (1998). ALE: The attribute logic engine, version 3.1, *User manual*, Carnegie Mellon University, Pittsburgh.

Chomsky, N. (1995). *The Minimalist Program*, Vol. 28 of *Current Studies in Linguistics*, MIT Press.

Frühwirth, T. (1998). Theory and practice of constraint handling rules, *Journal of Logic Programming* **37**.

Götz, T. and Meurers, D. (1997). Interleaving universal principles and relational constraints over typed feature logic, *ACL/EACL Conference '97*, Madrid, Spain.

Graham, G., Harrison, M. G. and Ruzzo, W. L. (1980). An improved context–free recognizer, *ACM Transactions on Programming Languages and Systems 2 (3)*.

Manandhar, S. (1994). An attributive logic of set descriptions and set operations, *ACL Conference '94*.

Matiasek, J. (1994). *Principle-Based Processing of Natural Language Using CLP Techniques*, PhD thesis, TU Wien.

Morawietz, F. (1995). A Unification-Based ID/LP Parsing Schema, *Proceedings of the 4th IWPT*, Prag.

Pereira, F. C. N. and Warren, D. H. D. (1983). Parsing as deduction, *ACL Conference '83*.

Shieber, S. M., Schabes, Y. and Pereira, F. C. N. (1995). Principles and implementation of deductive parsing, *Journal of Logic Programming* **24**(1–2).

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

Smolka, G. (1995). The Oz programming model, *in* J. van Leeuwen (ed.), *Computer Science Today, LNCS* **1000**, Springer.

Stabler, E. (1997). Derivational minimalism, *in* C. Retoré (ed.), *Logical Aspects of Computational Linguistics*, *LNAI* **1328**, Springer.

Stabler, E. (2000). Minimalist grammars and recognition, Presented at the Workshop *Linguistic Form and its Computation* of the SFB 340 in Bad Teinach, Universität Tübingen. Draft.