Arbeitspapiere des SFB 340
Bericht Nr. 00, Xxxx 19??

CHART PARSING AS CONSTRAINT PROPAGATION

FRANK MORAWIETZ

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. Extending the proposal in Shieber et al. (1995) which provides a meta interpreter for several deduction systems, we show how to directly realize the inference rules for chart parsers as Constraint Handling Rules (Frühwirth; 1998) by viewing the items of a conventional chart parser as constraints and the constraint base as a chart. This 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 extended in Shieber et al. (1995) and the parsing schemata defined as special deduction systems in Sikkel (1997) are well established parsing paradigms in computational linguistics. Their main strengths are their flexibility and level of abstraction concerning control information inherent in parsing algorithms. This allows for a uniform presentation and the comparison of a variety of 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 based on Oz (Smolka; 1995), or conditional constraints (Matiasek; 1994), or advanced compilation techniques (Götz and Meurers; 1997) or specialized constraint solvers (Manandhar; 1994). But to my knowledge, none of these approaches uses constraint programming techniques to implement standard chart parsing algorithms directly in a constraint system.

The research presented in this report was sponsored by Teilprojekt A8 of the Sonderforschungsbereich 340 of the Deutsche Forschungsgemeinschaft. I want to thank Gerald Penn and Tom Cornell for helpful discussion. Needless to say, all errors and infelicities that remain are my own.

ISSN 0947-6954/??

In this paper, I will bring these two paradigms together, thereby 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).[1] 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.

2. Parsing as Deduction

Parsing in general is the process of analyzing the structure of an input by decomposing it into subparts according to some given grammar. If the process can be completed successfully, the input is a valid expression of the language defined by the grammar.

Therefore, to define how to parse a string, one needs the following ingredients:

- a way of initializing the parsing process, i.e., some statements about the acceptability of (parts of) strings which are used to start the parsing process;

[1] CHR are provided as a package for SICStus Prolog and ECLiPSe.

- some rules which allow to derive new statements from existing ones, e.g., the initial ones, which actually define the parsing mode;
- a way of controlling the process, i.e., of deciding which statements to try next for possible rule applications;
- and a way of identifying a final statement, i.e., a statement which means that we successfully decomposed the string.

Parsing-as-deduction basically takes formulas of a logic and uses its axioms and deduction rules to drive the parsing process.

Obviously, parsing-as-deduction is an instantiation of the scheme given above. It derives statements about the acceptability of (parts of) strings from other such statements via inference rules, starts the process with axioms and stops when the possibilities for further deductions have been exhausted or a predefined goal has been reached. What does not have to be defined and which actually is seen as an advantage for the specification of parsing algorithms, is the control information. This search procedure is not relevant for the specification of a specific algorithm, but rather for its realization or implementation.

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 $w = w_1 \ldots w_n$, for the given input string. 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

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

where the A_i and C are formula schemata. The A_i are called antecedents and C the consequence. Note that the deduction schemes may refer to the string positions, i.e., the indices of the alphabet symbols of the input string, in their side conditions. We say that such a scheme R *applies* if we have as antecedents a sequence of formulas $F_1, \ldots, F_k$, $k \geq n$ such that each F_i, $1 \leq i \leq n$ matches A_i under an appropriate substitution of terms for the meta-variables and the side conditions are satisfied. For a given deduction system, the derivation of a formula F from assumptions $A_1, \ldots, A_p$ ($A_1, \ldots, A_p \vdash F$) is defined to be a sequence of formulas $F_1, \ldots, F_q$ such that $F = F_q$ and each F_i, $1 \leq i \leq q$, is either an instance of an axiom, one of the A_j, $1 \leq j \leq p$, or the result of applying a deduction scheme with $F_i = C$ such that there exist the necessary antecedents $A_{i_1}, \ldots, A_{i_k}, i_1, \ldots, i_k \leq i$.

To improve the performance of the implementation of a parser, one also wants to record work which has been done such that recomputations of (partial) results can be avoided. To simply store recognized substrings is not enough since one cannot differentiate between for example two identical occurrences of substrings. Therefore the statements are enriched by indexing the category recognized, i.e., the terminal or nonterminal symbol labeling a node in the parse tree, with the position(s) spanned by that category. These statements are then stored in a well-formed substring table, also called a chart. The chart does not play a direct role for the specification of grammatical deduction systems but since I will present examples below, it is necessary to introduce the complex formulas or statements used there.

Unless specified differently, we assume that we are given a context-free grammar $\mathcal{G} = \langle\ N, T, S, P\ \rangle$ with nonterminals N, terminals T, start symbol S and set of productions P.[2] Each production is of the form $A \longrightarrow \alpha$ with $A \in N$, $\alpha \in (N \cup T)^*$.

As a running example, I will use the simple PP-attachment grammar given in Fig. 1. It is left to the reader to calculate example derivations for the three example algorithms for a sentence such as *John hit the dog with the stick*.

$$\mathcal{G} = \langle\ N, T, S, P\ \rangle$$
$$NT = \{\text{S, NP, VP, V, PP, P, PN, Det, N1, N}\}$$
$$T = \{\textit{hit, John, dog, stick, with, the}\}$$

and P as given below:

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

FIGURE 1. Example Grammar: PP-attachment

All the parsing systems used in this paper are then defined by specifying a class of items, a set of axioms, a set of inference rules and a subset of the items,

[2]For Earley's algorithm we also assume a new start symbol S' which is not in N.

the goals or final items. For better readability, I follow Shieber et al. in using the familiar dotted items.

Notational conventions are: n for the length of the string to be parsed, for arbitrary formulas or nonterminals $A, B, C, \dots$, for terminals $a, b, c, \dots$, 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.

The three example algorithms, we will use to illustrate our technique can now be presented as grammatical deduction systems as given in Tab. 1 (taken from Shieber et al.; 1995). In the table i, j, k are always between 1 and n.

Although I assume familiarity with these three basic algorithms, I will briefly comment on the intended interpretations of the respective items. In the case of the naive bottom-up algorithm, the items are of the form $[j, \alpha \bullet]$ with the interpretation that we have a stack α of items we recognized reaching up to position j (which implies that we are still looking forward to parsing the substring starting at j). In the case of the top-down algorithm, the items $[j, \bullet\beta]$ mean that we are looking for categories β starting at position j (implying that we already recognized the string up to that position). The items in Earley's algorithm $[i, j, A, \alpha \bullet \beta]$ are to be interpreted that we recognized a substring from i to j with the rule $A \longrightarrow \alpha\beta$ having already found α, but looking for categories β.

3. Constraint Handling Rules

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. A complete introduction to constraint programming can be found for example 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 both SICStus Prolog and ECLiPSe (Intelligent Systems Laboratory; 1995; Frühwirth; 1998). All of these CP systems provide mechanisms for solving constraints. They 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.

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	$\frac{[j, \alpha \bullet]}{[j+1, \alpha w_{j+1} \bullet]}$	$\frac{[j, \bullet w_{j+1} \beta]}{[j+1, \bullet \beta]}$	$\frac{[i, j, A, \alpha \bullet w_{j+1} \beta]}{[i, j+1, A, \alpha w_{j+1} \bullet \beta]}$
Predict		$\frac{[j, \bullet B \beta]}{[j, \bullet \gamma \beta]} \langle B \longrightarrow \gamma \rangle$	$\frac{[i, j, A, \alpha \bullet B \beta]}{[j, j, B, \bullet \gamma]} \langle B \longrightarrow \gamma \rangle$
Complete	$\frac{[j, \alpha \gamma \bullet]}{[j, \alpha B \bullet]} \langle B \longrightarrow \gamma \rangle$		$\frac{[i, k, A, \alpha \bullet B \beta] \quad [k, j, B, \gamma \bullet]}{[i, j, A, \alpha B \bullet \beta]}$

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 (symbol <=>) which replace a number of constraints in the store with new constraints; secondly propagation rules (symbol ==>) which add new constraints to the store in case a number of constraints is already present; and thirdly "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.

To improve efficiency, 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.[3] A pragma declaration affects the way the rule is compiled. Our approach will only introduce the *passive* declaration, so we ignore the other possibilities. Declaring a constraint in the head passive simply means that this constraint will not trigger the rule on its own.

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).

4. 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, i.e., strings starting with uppercase letters stand for variables and words with lowercase ones for atoms.

We use the following two types of constraints. The constraints corresponding to the items will be called `edge` constraints, see Tab. 4. They have two arguments in case of the two naive algorithms and five in the case of Earley's

[3] 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). This is also part of CHR, but not used in our approach.

TABLE 2. Parsing systems as CHR programs: Items, Axioms & Goals

	Bottom-Up	**Top-Down**
Items	`edge(X,N)`	`edge(X,N)`
Axiom	`edge([],0)`	`edge([s],0)`
Goal	`edge([s],Len)`	`edge([],Len)`

	Earley
Items	`edge(A,Alpha,Beta,I,J)`
Axiom	`edge(sprime,[],[s],0,0)`
Goal	`edge(sprime,[s],[],0,Len)`

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/2` is only used for efficiency. Since in grammar rules we do not use the lexical items directly but rather their categories, I do not use edges to represent that we found a word, but the `word` constraint, as will become apparent in the scanning rules. This avoids using constraints on the input *words* (compared to their category) in the other inferences needlessly.

The grammars are 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 resulting algorithms are then simple to implement by specifying the inference rules as constraint propagation rules, the axioms and the goal items. A summarization is presented in Tab. 3.

It is obvious how to link the code for the axioms and goals given to the definitions in Tab. 1. Let us consider Earley's algorithm for a closer look as to what is going on in the CHR propagation rules.

In principle 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.[4]

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 with the positional index appropriately incremented.

The prediction step is a little 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 (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 corresponding tests avoid vacuous predictions and therefore 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. Additionally, the rule contains a pragma declaration on the first head which is declared to be *passive*. While I cannot go into detail here, this

[4] Note that calling a CHR constraint as a Prolog goal means to insert it into the constraint store such that it becomes available for constraint resolution steps.

TABLE 3. Parsing systems as CHR programs: Inference Rules

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)#Id,
             edge(B,Gamma,[],K,J) ==>
      edge(A,[B|Alpha],Beta,I,J)
             pragma passive(Id).
```

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.
```

declaration leads to the fact that this rule is only tried for inference if a constraint matching the non-passive head is posted. More on pragma declarations can be found in Frühwirth (1998).

In the table there is one more 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. We come back to this point a little below.

Note that we do not have to specify how to insert edges into either a chart or into an agenda. The chart and the agenda are in fact represented by the constraint store and therefore built-in. We do not need a specialized deduction engine as was necessary for the implementation described in Shieber et al. (1995). In fact, the utilities needed are extremely simple, see Fig. 2.

```
parse(L) :-
         axiom,
         post_constraints(L, 0, Length),
         report(Length).

post_constraints([], Length, Length).
post_constraints([Word|String], InLength, Length) :-
         findall(Cat, lex(Word, Cat), Cats),
         post_words(Cats, InLength, Word),
         NewLength is InLength + 1,
         post_constraints(String, NewLength, Length).

post_words([],_Position,_Word).
post_words([Cat|Cats], Position, Word):-
        word(Position, Cat-Word),
        post_words(Cats, Position, Word).
```

FIGURE 2. The utilities for CHR-based deductive parsing

All we have to do for parsing (predicate `parse/1`) is to post the axiom[5] 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.

[5] `axiom/0` is a predicate which just calls the edge(s) defined in Tab. 3.

Coming back to the issues of chart and agenda: the constraint store functions as chart and agenda at the same 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.[6] So, the parser works incrementally and recursively tries all possible inferences for each constraint added to the store before continuing with the posting of new constraints from the `post_constraints/3` predicate. The way this predicate works at the moment is to traverse the string from left-to-right. It is trivial to alter the predicate such as 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 connection with the absorption rules. We absorb the newer edge since we can assume that all possible propagations have already 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, the output of an Earley-parse for *John hit the dog with the stick* assuming the grammar given in Fig. 1 is presented in Fig. 3.

The entire constraint store is printed to the screen after the constraint resolution process stops producing new edges. And furthermore, 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. Naturally the given string was ambiguous but 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. (1995) 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 running SICStus Prolog. This factor was preserved under 5 and 500 repetitions of the same parse. This may or may not be seen as a serious problem. However, there are two remarks I want to make. Firstly, speed was not the main issue in developing this setup, but rather simplicity and ease of implementation. And secondly it seems that the implementation of the attributed variables package of SICStus Prolog which is the basis for the CHR package, is too general to be maximally efficient. Gerald

[6] 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),             edge(n1,[n],[],3,4),
 word(1,v-hit),               edge(np,[n1,det],[],2,4),
 word(2,det-the),             edge(vp,[np,v],[],1,4),
 word(3,n-dog),               edge(s,[vp,np],[],0,4),
 word(4,p-with),              edge(sprime,[s],[],0,4),
 word(5,det-the),             edge(vp,[np,v],[pp],1,4),
 word(6,n-stick),             edge(pp,[p],[np],4,5),
 edge(sprime,[],[s],0,0),  edge(np,[],[det,n1],5,5),
 edge(s,[],[np,vp],0,0),   edge(np,[],[pn],5,5),
 edge(np,[],[det,n1],0,0), edge(np,[det],[n1],5,6),
 edge(np,[],[pn],0,0),     edge(n1,[],[n],6,6),
 edge(np,[pn],[],0,1),     edge(n1,[],[n,pp],6,6),
 edge(s,[np],[vp],0,1),    edge(n1,[n],[pp],6,7),
 edge(vp,[],[v,np],1,1),   edge(pp,[],[p,np],7,7),
 edge(vp,[],[v,np,pp],1,1),edge(n1,[n],[],6,7),
 edge(vp,[v],[np,pp],1,2), edge(np,[n1,det],[],5,7),
 edge(np,[],[det,n1],2,2), edge(pp,[np,p],[],4,7),
 edge(np,[],[pn],2,2),     edge(vp,[pp,np,v],[],1,7),
 edge(vp,[v],[np],1,2),    edge(s,[vp,np],[],0,7),
 edge(np,[det],[n1],2,3),  edge(sprime,[s],[],0,7),
 edge(n1,[],[n],3,3),      edge(n1,[pp,n],[],3,7),
 edge(n1,[],[n,pp],3,3),   edge(np,[n1,det],[],2,7),
 edge(n1,[n],[pp],3,4),    edge(vp,[np,v],[],1,7),
 edge(pp,[],[p,np],4,4),   edge(vp,[np,v],[pp],1,7) ?
```

FIGURE 3. Chart for *John hit the dog with the stick*

Penn (Penn; pc,i) reports two instances of special applications where the efficiency of the implementation was improved by using specific predicates and not the full library. On implementing co-routining in a typed feature language built on top of Prolog, it turned out that a speed-up by a factor of about 50 could be achieved by using `when/2`, and on implementing typed feature structure unification that, by using some undocumented internal predicates directly rather than `get_atts/2` and `put_atts/2`, a speed-up of a little over a factor of 28 could be achieved with the ALE (Carpenter and Penn; 1998) Head-Driven Phrase Structure (HPSG) grammar. If we can apply similar techniques, the implementation may be able to improve on its performance.

To sum up this section, the advantages of the approach lie in its flexibility and its availability for rapid prototyping of different parsing algorithms since it avoids any explicit handling of chart or agenda. While we used the examples from the Shieber et al. article, one can also implement all the different parsing schemata presented as deduction schemes in Sikkel (1997). This also includes advanced schemes 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 overcome this limitation is discussed in the next section.

5. 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 to this approach as well. Instead, I want to consider parsing minimalist grammars (Chomsky; 1995) as defined in recent work by Stabler (1997, 1999).[7]

5.1. **Minimalist Parsing.** We cannot cover the theory behind the derivational minimalism approach 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)` and `spec(C)` for `=c`.[8] 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 of the tree under consideration and its merging in at the root node. On completion of these operations the clashing feature pairs are removed (checked). The lexical items are of the form of linked sequences of trees. Accessibility of features is defined via an order on the nodes in this chain

[7] The code for the 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.

[8] The split of the *merge* rule between merging complementizers and specifiers allows for a more perspicuous presentation of the inference rules, as one can see in Tab. 5.

of trees.[9] 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. This is the nice thing about Stabler's proposal – minimalist parsing can be reduced to CYK parsing.

An initial edge or axiom in this minimalist parsing system cannot simply be assumed to only cover the part of the string where it was found since it could have been the result of a movement operation. So the elements of the lexicon which will have to be moved (they contain a phrasal 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. The same is true for the empty elements appearing explicitly in the lexicon.

TABLE 4. A CHR-based minimalist parser: Items, Axioms and Goals

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 there is a `-X` in `[Chain|Chains]`
	and `I` and `I+1` occur in `[Chain|Chains]`
Goal	`edge(0, Length, [cat(C)], [])`

As an example consider the set of items as defined by the axioms, see Tab. 5.1. 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. The axioms post edges corresponding to the lexical items. As an example

[9] We ignore certain aspects for simplicity. The reader is referred to the original literature for the full details.

TABLE 5. A CHR-based minimalist parser: Inference Rules

Absorption

```
edge(I, J, L, Ls) \ edge(P, Q, L, Ls) <=>
      unif_vars(I, J, P, Q) |
      true.
```

Merge$_{Comp}$

```
edge(I, J, [comp(X)|RestHead], Chains0),
   edge(K, L, [cat(X)|RestComp], Chains1) ==>
      test_vars(I, J, K, L, P, Q, R, S),
      check(RestHead, NewHead, P, S, A, B,
            RestComp, Chains0, Chains1, Chains) |
   edge(A, B, NewHead, Chains).
```

Merge$_{Spec}$

```
edge(I, J, [cat(X)|RestSpec], Chains0),
   edge(K, L, [spec(X)|RestHead], Chains1) ==>
      test_vars(I, J, K, L, P, Q, R, S),
      check(RestHead, NewHead, P, S, A, B,
            RestSpec, Chains0, Chains1, Chains) |
   edge(A, B, NewHead, Chains).
```

Move

```
edge(I, J, [+(X)|RestHead], Chains0) ==>
      test_vars(I, J, P, Q),
      Link =.. [X,K,P],
      select([-Link|RestLinks], Chains0, Chains1),
      add(RestLinks, Chains1, Chains),
      linkStart(RestHead, NewHead, K, Q, L, M) |
   edge(L, M, NewHead, Chains).
```

for the actual edges and to illustrate the discussion about the possibly variable string positions in the edges, consider the recognition of the string *believe it* with a remnant movement analysis. The lexical item for *it* takes the form:

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

where `I=1` if we instantiate it as an axiom. The resulting edge will be `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.

These non-ground edges actually have an effect on the inference rules, see Tab. 5. Apart from this complication, they pretty much reflect a bottom-up approach to merging and moving lexical items. Therefore we will not discuss the details of how the conditions in the guards achieve to test for the correct configurations. The interested reader is referred to Stabler (1999).

I will briefly discuss the predicates `unif_vars/4`, `test_vars/4` and `test_vars/8`.[10] In the Shieber at al. system, we compared ground positional indices to find matching edges. Since CHR are sensitive to the instantiation of variables, it is not possible to do the same here.[11] The above mentioned predicates basically test for their arguments to be variables. If they are, new ones are returned such that the relationship (identity) between the members of pairs are preserved. If they are ground, they are unified. It is obvious that the price we pay is a decrease in efficiency since the CHR system actually has to enter the guards to determine applicability of the rules.

We cannot go into any detail how the actual parser work. Nevertheless, the implementation shown in Tab. 5 demonstrates how easily one can implement even complicated parsers without the need for extra utilities to deal with chart and agenda. Furthermore, it offers the possibility to implement feature cancellation as real constraint solving which opens up an interesting new perspective for processing.

5.2. **Compiling the Grammar Rules into the Inference Rules.** Another proposal for extending 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 process more efficient. For simplicity, we discuss only the simplest 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

[10] Since we are dealing with the positions, the arguments always are members of a pair stemming from one edge.

[11] CHR have a compile time option which allows the disabling of this sensitivity. But doing this does not help since CHR are committed choice and the unification cannot be undone via backtracking.

– since empty RHSs constitute a problem for bottom-up chart parsing.[12] 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, but appropriately matching string positions but 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)
```

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, e.g., $\mathtt{A} \longrightarrow \epsilon$, 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) ==> 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.[13] 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. There are some further ways to improve the performance of the

[12] Top-down parsing, on the other hand, has to be careful with left recursion and care has to be taken in the parser to avoid nontermination.

[13] 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.

resulting CHR parsers involving for example pragma declarations or more sophisticated ways of handling empty categories via a precompiled link relation which I cannot go into here for reasons of space.

6. 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.

More work has to be invested in the realization of 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. To be more specific, 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 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, this remains speculative.

References

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

Abdennadher, S., Fruehwirth, T. and Meuss, H. (1996). On confluence of constraint handling rules, *Lecture Notes in Computer Science* **1118**: 1–15.

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.

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**(1–3): 95–138. Special Issue on Constraint Logic Programming.

Götz, T. and Meurers, D. (1997). Interleaving universal principles and relational constraints over typed feature logic, *ACL/EACL Conference '97*, Madrid, Spain, pp. 1–8.

Graham, G., Harrison, M. G. and Ruzzo, W. L. (1980). An improved context–free recognizer, *ACM Transactions on Programming Languages and Systems 2 (3)*, ACM, pp. 415–462.

Intelligent Systems Laboratory (1995). SICStus Prolog user's manual, *Technical report*, Swedish Institute of Computer Science.

Manandhar, S. (1994). An attributive logic of set descriptions and set operations, *Proceedings of the 32nd. Annual Meeting of the Association for Computational Linguistics*, ACL.

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

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 International Workshop on Parsing Technologies*, ACL/SIGPARSE, Prag, pp. 162–173.

Penn, G. (in press). An optimized prolog encoding of typed feature structures, *Proceedings of the 16th International Conference on Logic Programming*, Las Cruces, New Mexico.

Penn, G. (pc). Implementing co-routining in a typed language, *To be published in the Arbeitspapiere des SFB 340*, Universität Tübingen.

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.

Shieber, S. M., Schabes, Y. and Pereira, F. C. N. (1995). Principles and implementation of deductive parsing, *Journal of Logic Programming* **24**(1–2): 3–36.

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

Smolka, G. (1995). The Oz programming model, *in* J. van Leeuwen (ed.), *Computer Science Today*, Vol. 1000 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 324–343.

Stabler, E. (1997). Derivational minimalism, *in* C. Retoré (ed.), *Logical Aspects of Computational Linguistics*, Springer, Berlin, pp. 68–95. LNAI 1328.

Stabler, E. (1999). Performance models for a derivational minimalism, Presented at the Workshop *Linguistic Form and its Computation* of the SFB 340 in Bad Teinach. Draft.

SEMINAR FÜR SPRACHWISSENSCHAFT, SFB 340, PROJEKT A8, UNIVERSITÄT TÜBINGEN, WILHELMSTRASSE 113, D-72074 TÜBINGEN, GERMANY
E-mail address: frank@sfs.nphil.uni-tuebingen.de