Generalizing Earley Deduction for Constraint-Based Grammars

Jochen Dörre

1 Introduction

The motivation for this work stems mainly from the attempt to apply ideas of chart-parsing, or more generally, tabled deduction to the new family of extended feature-based formalisms employed for (linguistic or general) knowledge representation, such as TFS [EZ90, Zaj92], LIFE [AKP91], CUF [DD93], or others. For linguistic applications these formalisms are of special interest, because they allow for a direct implementation of principle-based grammars, which model grammatical principles as systems of interacting constraints. Principles ranging from syntactic constraints on word order up to binding constraints for anaphora resolution can thus be expressed very naturally and in a uniform manner. In view of the expressive power of these formalisms one could even envisage modelling semantic inference therein.

The problem we are concerned here is how to use a system of definite clauses encoding a grammar for deduction of the grammaticality of given strings or for the generation of grammatical strings in a way which explores best the restricting nature of constraints.

An underlying assumption shall be that linguistic specifications are written in a direct declarative way without concern of processing strategies which will be used to "run" the specifications to perform a certain task. Instead, we want to separate as much as possible the declarative from the procedural aspects of a specification, the latter being provided by task-dependent control information that accompanies the declarative specification. This separation helps to keep linguistic specifications in line with linguistic theory and gives implementations of them more weight in the task of theory verification.

We assume here that constraints, or better: predicates, are defined using definite clauses over some feature constraint language in the sense of Höhfeld and Smolka's CLP-scheme (constraint logic programming scheme) [HS88], as it is done in CUF. To be more precise, a clause has the form

$$p \leftarrow q_1 \wedge \ldots \wedge q_n \wedge \phi.$$

Here the so-called relational atoms p and $q_1 \ldots, q_n$ are atoms built only from a predicate name and distinct variables as arguments. ϕ is a possibly complex constraint from the given constraint language ranging over those variables. To avoid confusion we will call constraints from this built-in constraint language *built-in constraints*, whereas the term 'constraint' shall keep its general intuitive meaning. Note that the property 'built-in' here refers to the manner how the so-qualified constraints are put to use. Their solution is the task of a hidden constraint solver and this is regarded as a primitive operation from the level of the resolution procedure in just the same way as unification is considered primitive in ordinary resolution. Actually, ordinary logic programming is a proper instance of this CLP-scheme, where the constraint language allows ϕ to be a conjunction of equations between first-order terms. We can hence think of the built-in constraint solver as being a generalized unification procedure and of the resulting (normalized) constraints as being generalized variable bindings.

Since we are concerned in this paper only with the higher level deduction strategies, we will ignore such details of the integration of the built-in constraint language altogether. To simplify the presentation, we discuss SLD-resolution and Earley deduction in the ordinary setting using unification. It should however be kept in mind that we always can generalize unification to constraint solving and replace the application of a variable substitution to a clause by the addition of the respective binding constraints to the clause constraint.

Let us now consider more closely the standard method for the processing of definite specifications, SLD resolution (linear resolution of definite clauses with a selection function).

$$
\begin{array}{rc}
\text{(1) Goal:} & \leftarrow q_1 \wedge \ldots \wedge \underline{q_i} \wedge \ldots \wedge q_n \\
\text{(2) Clause:} & q_i' \leftarrow p_1 \wedge \ldots p_m \\
\text{(3) Unifier:} & \sigma \; = \; mgu(q_i, q_i') \\
\hline
\text{(4) Next Goal:} & \leftarrow \sigma[q_1 \wedge \ldots \wedge q_{i-1} \wedge p_1 \wedge \ldots \wedge p_m \wedge q_{i+1} \wedge \ldots \wedge q_n]
\end{array}
$$

Figure 1: SLD resolution rule

The SLD resolution rule defines a nondeterministic procedure to prove a goal G (Fig. 1). Here q_i is the selected literal of goal G and σ is the substitution which is the most general unifier of p_i and p_i'. Which literal in a goal will be selected is given by a selection function, called the computation rule. SLD resolution is made into a deduction algorithm by supplying this selection function and embedding it into a search procedure which takes care of the nondeterminism. For instance, we obtain the PROLOG proof strategy if we select always the first literal of a goal and use depth-first search with backtracking to explore the search space.

The strategy of selecting always the first literal has certain disadvantages if specifications are given more in a constraint-based fashion. When using this strategy a proof of a goal $p_1 \wedge \ldots \wedge p_n$ can be strictly divided into a proof of p_1 and a proof of the other literals, where during the first the other literals are completely ignored. They are not used to restrict the search space of the proof of p_1, although they could. They only may later on reject some of its solutions.

In the CUF prototype [DE91] we therefore used the SLD resolution method[1] with the following computation rule:

Select the first deterministic literal, if there is such a literal, else select the first literal.

With a deterministic literal we mean a literal for which at most one clause exists with which it might resolve, i.e., for which we can perform a resolution step without introducing a choice point. We call this strategy *deterministic closure*, since choice points are only introduced after all literals which are deterministic have been expanded.[2] As concerns the choice points themselves, the ordinary PROLOG strategy is used: they are considered using backtracking in a depth-first manner.

As an example, consider the following specification of the n-queens problem, i.e. the problem of positioning n queens on an $n \times n$ chess board such that no queen threatens any other.

```
n_queens(N,Solution) :-
      list_of_n(N,List),
      permute(List,Solution),
      check_sw_to_ne(Solution,1,[]),
      check_nw_to_se(Solution,1,[]).
```

[1] where ordinary unification of terms is replaced by unification of feature structures

[2] Actually, the strategy employed in the CUF system can be refined by the declaration of delay statements. These statements, if present for a certain predicate, tell the computation rule how much instantiated a nondeterministic literal has to be in order to be a candidate for selection. A similar strategy is called *residuation* in [Smo91].

```
list_of_n(0,[]).
list_of_n(N,[N|L]) :-
      N > 0,
      N1 is N-1,
      list_of_n(N1,L).

permute([],[]).
permute(L,[F|R]) :-
      delete(F,L,R1),
      permute(R1,R).

delete(F,[F|R],R).
delete(F,[H|R],[H|R1]) :-
      delete(F,R,R1).


check_sw_to_ne([],_,_).
check_sw_to_ne([I|R],N,DiffSoFar) :-
      Diff is I-N,
      non_member(Diff,DiffSoFar),
      N1 is N+1,
      check_sw_to_ne(R,N1,[Diff|DiffSoFar]).

check_nw_to_se([],_,_).
check_nw_to_se([I|R],N,SumSoFar) :-
      Sum is I+N,
      non_member(Sum,SumSoFar),
      N1 is N+1,
      check_nw_to_se(R,N1,[Sum|SumSoFar]).
```

Solutions are given as permutations of the list $1 \ldots n$, where a j in the i-th position means that queen i stands in column i, row j. Note that this implies already that each row as well as each column holds at most one queen. We only have to make sure that this also holds for the diagonals, which is the task of the constraints `check_sw_to_ne` and `check_nw_to_se`. They simply have to make sure that the difference, respectively the sum, of the two coordinates of a queen is distinct to that of each other queen. Now, the interesting thing about these two constraints is that they can work incrementally as the list `Solution` grows, if we use the deterministic closure strategy. Consider for instance the following state of computation during the evaluation of `n_queens(4,S)`.

```
delete(F,[4,3,2,1],R1),
permute(R1,R),
check_sw_to_ne([F|R],1,[]),
check_nw_to_se([F|R],1,[]).
```

When choosing the first clause of `delete` we get the bindings `F=4`, `R=[3,2,1]`. In the next step we substitute `permute(R1,R)` by its subgoals using the second clause of `permute`, the only matching one. We thus obtain:

```
delete(F',[3,2,1],R1'),
permute(R1',R'),
check_sw_to_ne([4,F'|R'],1,[]),
check_nw_to_se([4,F'|R'],1,[]).
```

Note that a binding of `F'=3` here would cause the last constraint `check_nw_to_se` to fail, no matter how other variables would be bound, since $4 + 1 = 3 + 2$, i.e., the two queens share a NW–SE diagonal. Note also that we can detect this failure by purely deterministic expansions. The effect of this failure is that generation of the permutations that start with $\langle 4, 3\rangle$ is inhibited at all. It is clear that for a larger n this will cause dramatic reductions in the search. Thus, performing eager expansions of the goals `check_sw_to_ne` and `check_nw_to_se` as soon as they (or their subgoals) become deterministic actively restricts the search space for the computation of the permutation.

We have chosen this toy example, because it shows clearly the structure of the problems for which the deterministic closure strategy is to prefer. We assume that many linguistic problems exhibit the same kind of structuring, namely that one part of the specification serves as a generator for recursive structures while other parts which are recursive on the same structures are set up to filter out certain combinations. Especially, if grammars are written modularly in the sense of GB theory, this kind of structuring is inevitable.

It should also become clear in the example above in which way this strategy is more advantageous than the ordinary PROLOG strategy. The important thing to note here is that before the selection of a nondeterministic literal, i.e., before the creation of a choice point, every literal in the goal will be considered whether it can supply information to the current state by expanding deterministically. Hence, literals which are further to the end of the list can react on bindings of variables like active constraints. The evaluation strategy becomes data-driven and requires generally less backtracking.[3]

However, even when applying the strategy of deterministic closure as described above, one big problem still remains, namely the handling of ambiguity. The use of chronological backtracking may lead to unnecessary recomputations of the same part of a proof over and over again in cases where somewhere earlier in the proof multiple solutions occurred, which happen not to affect this part. A well-known remedy to this problem in parsing is the use of a chart, also known as "well-formed substring table", together with algorithms which generate and exploit this data structure, e.g., the Earley algorithm.

We therefore want to investigate how the Earley deduction method [PW83], the generalization of Earley's parsing method to general deduction for Horn clauses, can be explored in a constraint-based setting.

The next section illustrates the problems that occur if we want to use Earley deduction directly for constraint specifications as introduced above. In Section 3 we consider two approaches to attack the deduction problem by generalizing Earley deduction in two different ways. These two generalizations address two independent aspects in the deduction procedure and can straightforwardly be combined. Next, we consider the implications that generalized Earley deduction has for the treatment of disjunctive information, if applied, e.g., for parsing. A generalization of OLDT resolution [TS86] which was recently devised by Mark Johnson [Joh93] and which has many similarities to our method is discussed shortly in Section 5. Finally, we present a conclusion.

2 Problems

In this section we describe the problems that occur if we want to apply the general Earley deduction method to deduction in constraint-based formalisms. The aim we have in mind, namely that constraints should be used collectively to perform an active reduction of the search space, like in the strategy of deterministic closure described above, is not as easily achieved with this method as it is with SLD resolution.

Earley deduction is a proof procedure that works on two sets of Horn clauses, the program

[3] Another implementation technique for this kind of deduction strategy, which PROLOG programmers might think that the example calls for, is the use of the co-routining facility `freeze`. The disadvantage of this technique, however, is that we have to specify the delaying of subgoals in the code, thereby anticipating how the structure building should proceed. A better idea would be to think of introducing this extra code automatically with a compiler, thus using the co-routining facilities to implement deterministic closure. Unfortunately, there are situations in which the binding of variables forces predicates to become deterministic which cannot be captured by the `freeze` technique, namely when a variable gets bound to another variable. In this case no melting occurs. Hence, the deterministic closure strategy can only be approximated using `freeze`.

clauses P and the set of derived lemmata S. Derived lemmata correspond to constructed edges in chart parsing. As we can have active and inactive edges in charts, we distinguish here between an active lemma, one with a nonempty body, and a passive lemma, one with an empty body (unit clause). A selection function determines for each active lemma its selected negative literal (normally the first literal of the body). The algorithm takes as input the set of program clauses P and a goal clause $\leftarrow G$, which constitutes the initial content of the table S. The following two inference rules are used to continually add clauses to S.[4]

1. **Prediction:** If $B \in S$ is an active clause with selected literal q_i and $q \leftarrow \gamma$ is an (active or inactive) program clause such that q_i and q unify with m.g.u. σ, then add $\sigma(q \leftarrow \gamma)$ to S.
2. **Completion:** If $C \in S$ is the passive clause q and $D \in S$ is the active clause $p \leftarrow q_1 \wedge \ldots \wedge q_i \wedge \ldots \wedge q_n$ with selected literal q_i such that q_i and q unify with m.g.u. σ, then add $\sigma(p \leftarrow q_1 \wedge \ldots q_{i-1} \wedge q_{i+1} \ldots \wedge q_n)$ (the resolvent of C and D) to S.

However, the rules are used only if they actually add "new" information to S, i.e. if the inferred clause is not subsumed by a clause already in S. Otherwise this rule application is *blocked*.

For a more detailed consideration of the problem and solutions to it we use the PROLOG implementation of Earley deduction given in [PS87] (see Fig. 2) as the basis for the discussion of our modifications.

Now, let us consider again the example given in the introduction.

After a number of steps which correspond to the (unproblematic) evaluation of `list_of_n(4,L)` the following lemma is produced:

```
n_queens(4,Solution) <*=
      permute([4,3,2,1],Solution),
      check_sw_to_ne(Solution,1,[]),
      check_nw_to_se(Solution,1,[]).
```

The prediction step then has to initiate the subcomputation of the literal `permute([4,3,2,1],Solution)` by adding the lemma

```
permute([4,3,2,1],[F|R]) <*=
      delete(F,[3,2,1],R1),
      permute(R1,R).
```

But this initiates the computation of all solutions of `permute([4,3,2,1],Solution)` and the algorithm cannot take into account the restrictions that are imposed by the two remaining subgoals `check_sw_to_ne` and `check_nw_to_se` from above.

Notice that exactly this kind of problem occurs with declarative specifications of HPSG or GB grammars. By the modularity of these types of grammar, which attempt to describe different principles of language in independent modules, the number of structures, which fulfill only one of these principles is in general infinite. Only by looking at different principles simultaneously, which all have to be fulfilled, it is possible to construct meaningful feature structures which describe the correct structure of a sentence.

There is a deeper reason why the prediction step in the Earley deducer introduces the isolation of the selected subgoal as explained above. Hence, the problem described here is not a superficial one. Infact, for the Earley strategy to work one must be able to recognize when a subproof, which has been computed once, can be used another time in the proof. This procedure is the more effective the more we abstract subproofs from their (irrelevant) context of their first occurrence. Hence, we have to trade here reusability of a subproof against the chance of its termination.

[4]Prediction is also sometimes called *instantiation* and completion is sometimes called *resolution* or *reduction*.

```prolog
:- op(900,xfx,<=).
:- op(900,xfx,<*=).

:- dynamic (<*= / 2).

prove(Goal) :-
    predict(Goal,Agenda),
    process(Agenda),
    Goal <*= [].

process([]).
process([Head<*=Body | OldAgenda]) :-
    process_one(Head,Body,SubAgenda),
    append(SubAgenda,OldAgenda,Agenda),
    process(Agenda).

process_one(Head,[],Agenda) :-
    complete_pas(Head,Agenda).

process_one(Head,[F|Body],Agenda) :-
    predict(F,Front),
    complete_act(Head<*=[F|Body], Back),
    append(Front,Back,Agenda).

predict(Goal,Agenda) :-
    findall(Clause,
            Goal^prediction(Goal,Clause),
            Agenda).

prediction(Goal, Goal<*=Body) :-
    Goal <= Body,
    store(Goal <*= Body).

complete_pas(Fact,Agenda) :-
    findall(
       Clause,
       Fact^p_completion(Fact,Clause),
       Agenda).

p_completion(Fact, Goal<*=Body) :-
    Goal <*= [Fact|Body],
    store(Goal <*= Body).

complete_act(Clause,Agenda) :-
    findall(
       NewCl,
       Clause^a_completion(Clause,NewCl),
       Agenda).

a_completion(Hd<*=[F|Body], Hd<*=Body) :-
    F <*= [],
    store(Hd <*= Body).

store(Clause) :-
    \+ subsumed(Clause),
    assert(Clause).

subsumed(Clause) :-
    GenHead <*= GenBody,
    subsumes(GenHead<*=GenBody,Clause).

subsumes(Gen,Spec) :-
    numbervars(Spec,0,_),
    Gen=Spec.
```

Figure 2: PROLOG predicates implementing Earley deduction

Notice that for programs which are designed to work with the PROLOG strategy this abstraction from the context poses no problem, since in the PROLOG strategy during the proof of the first literal of a goal the remaining literals are completely ignored, i.e., they are not used to restrict the search space of the first anyway. But for specifications which require a co-routined evaluation of two or more literals this abstraction presents a severe obstacle. Due to the complete isolation of the selected literal in its subproof Earley deduction is in principle not able to simulate a co-routining strategy, no matter how the computation rule is chosen.

The only information that goes into new subproofs are the bindings of the variables which correspond to the arguments of the selected literal.[5] This is interesting, because bindings for a variable X or subgoals which contain X are just two different forms of restricting the solutions for X.

Hence, one way to proceed could be to consider a certain subset of the literals, namely those which should work as filters, as a generalized form of binding, called constraint, just like in the SLD resolution scheme for CLP of Höhfeld and Smolka [HS88]. Constraints would be handled analogous to bindings in Earley deduction. But this would mean that we would have to classify beforehand which literals should be considered as ordinary literals and which ones should be constraints, a distinction that we didn't had to make for the deterministic closure strategy described above.

[5] Actually, when using a restrictor (see, for instance, the discussion in [PS87, p. 208]), we abstract even from parts of these bindings.

Still, this approach shows us a fundamental method to escape the isolation dilemma: we have to combine related literals to bundles and perform isolated subproofs only on these bundles. Starting from this idea we consider two approaches in the next section.

3 Approaches

3.1 Earley Deduction with Bundled Goals

The first approach we want to describe involves the explicit declaration of what combinations of literals should be bundled. These declarations are called *folder declarations*.[6] We can apply these in the step where the deducer fetches a clause from the program, which means that we can hide these applications in the interface of the deduction engine to the program database. Bundled literals appear to the deduction engine as a single literal.

Consider the following linguistic example. Though it is a gross simplification it shows the general structure of a modular grammar with an independent formulation of different linguistic principles.

```
:- op(700, xfx, ==>).

yield(_-Word, [Word|Ws], Ws).
yield(_/[Tree], Ws0, Ws) :- yield(Tree, Ws0, Ws).
yield(_/[Tree1,Tree2], Ws0, Ws) :-
    yield(Tree1, Ws0, Ws1),
    yield(Tree2, Ws1, Ws).

xbar(Cat-Word, Cat) :- lex(Word, Cat).
xbar(Cat/[Tree], Cat) :-
    Cat ==> [Cat1],
    xbar(Tree, Cat1).
xbar(Cat/[Tree1,Tree2], Cat) :-
    Cat ==> [Cat1, Cat2],
    xbar(Tree1, Cat1),
    xbar(Tree2, Cat2).

s ==> [np, vp].
np ==> [np, n].
vp ==> [v].
vp ==> [v, np].

lex(kim, np).
lex(friend, n).
lex(sleeps, v).

wf_s(String, Tree) :- yield(Tree, String, []), xbar(Tree, s).
```

In this little program syntax trees are represented recursively using terms of the form `Category-Word` for a tree of height one consisting of the root `Category` and the leaf `Word` and terms of the form `Category/DL` for a tree with root `Category` and with `DL` being the list of immediate subtrees (**d**aughters **l**ist). `Category` and `Word` are atomic terms in this grammar representing the grammatical category and the word labeling a leaf, respectively. `yield` is a predicate relating a

[6]The idea of using folders is due to Mark Johnson, who presented a sketch of this method in his "Feature Structures" seminar in summer 1992 in Stuttgart. The examples of this section have been taken over with minor modifications from his presentation.

tree to the difference list of its leaves and `xbar(Tree,Cat)` is true if `Tree` has root label `Cat` and is well-formed according to the little context-free grammar encoded with the predicates `==>` and `lex`. In order to parse a given string S we simply have to solve the goal `wf_s(`S`,Tree)`.

We use the following folder declaration:

```
bundle1(Tree, Cat, S1, S2) :- yield(Tree, S1, S2), xbar(Tree, Cat).
```

The processing of a goal `wf_s([kim,sleeps],T)` now proceeds as usual by predicting lemmata whose head matches the goal. However, our interface to the program database substitutes occurrences of instances of the combination `yield(Tree, S1, S2), xbar(Tree, Cat)` by `bundle1(Tree, Cat, S1, S2)`. Hence, we get as the first lemma

```
(1) wf_s([kim,sleeps],T) <*= bundle1(T,s,[kim,sleeps],[]).
```

In the next step we have to predict for `bundle1(T,s,[kim,sleeps],[])`. This means we have to retrieve clauses for `yield` and `xbar` with the respective arguments. Now, we actually have the choice of how much unfolding of these literals, i.e., application of their program clauses, we want to perform. Let us assume for the moment that we unfold every literal in the bundle once.[7] We will come back to this point later on. The following two combinations of the subgoals of `yield` and `xbar` are consistent and will be generated with the variable substitutions indicated:

```
yield(T1,[kim,sleeps],[]), s==>[C], xbar(T1,C)              T=s/[T1]

yield(T1,[kim,sleeps],S1), yield(T2,S1,[]),
s==>[C1,C2], xbar(T1,C1), xbar(T2,C2)                       T=s/[T1,T2]
```

But on both we can apply the folder declaration yielding the lemmata

```
(2) bundle1(s/[T1],s,[kim,sleeps],[]) <*=
       s==>[C], bundle1(T1,C,[kim,sleeps],[]).
(3) bundle1(s/[T1,T2],s,[kim,sleeps],[]) <*=
       s==>[C1,C2], bundle1(T1,C1,[kim,sleeps],S1), bundle1(T2,C2,S1,[]).
```

Two points are worth mentioning here. The first point concerns the unfolding strategy. Due to the perfectly parallel recursion of the two predicates `yield` and `xbar` we come through with the simplistic strategy of performing exactly one unfolding step on each literal in the bundle. In general, however, it is not so easy to get a synchronization of two or more constraints which work recursively through the same structure. Note that if we cannot apply the folder once, e.g., because one of the partners in the bundle has not been generated, we risk that the deduction engine performs prediction on the other which in a program like the above inevitably would lead to nontermination. For example, if we use a slightly more general version of `yield`, which works not only on unary or binary branching trees, as given below, we are lost.

```
yield(_-Word, [Word|Ws], Ws).
yield(_/[], Ws, Ws).
yield(C/[Tree1|RT], Ws0, Ws) :-
    yield(Tree1, Ws0, Ws1),
    yield(C/RT, Ws1, Ws).
```

The third clause of `yield` together with the third clause of `xbar` result in the sequence

```
yield(T1,[kim,sleeps],S1), yield(C/[T2],S1,[]),
s==>[C1,C2], xbar(T1,C1), xbar(T2,C2)                    T=s/[T1,T2].
```

[7]Note that if we would just unfold one literal and predict on it, we would end up in the situation that we wanted to prevent.

Since `yield(C/[T2],S1,[])` and `xbar(T2,C2)` cannot be bundled we are forced at some time to select one of them for prediction. But both generate an infinite set of solutions, even when taking into account the substitutions `S1=[sleeps]` and `C2=vp` resulting from the successful reduction of the remaining literals. Let us call this problem the synchronization problem.

As a second point we observe that the bundling of literals can be precomputed, since our strategy allows only the subgoals of bundled literals to appear together in a lemma. Notice that in the example above the predicates `yield` and `xbar` don't appear at all in lemmata. Infact, we can transform the program statically introducing a new recursive predicate `bundle1` which replaces these two by merging their clauses, as shown below.

```
bundle1(Cat-Word, Cat, [Word|Ws], Ws) :- lex(Word, Cat).
bundle1(Cat/[Tree], Cat, Ws0, Ws) :-
    Cat ==> [Cat1],
    bundle1(Tree, Cat1, Ws0, Ws).
bundle1(Cat/[Tree1,Tree2], Cat, Ws0, Ws) :-
    Cat ==> [Cat1, Cat2],
    bundle1(Tree1, Cat1, Ws0, Ws1),
    bundle1(Tree2, Cat2, Ws1, Ws).
```

Actually, our strategy of bundling simply simulates ordinary Earley deduction using this predicate. Hence, we have to consider the question whether a static program transformation is preferable over the dynamic folding and unfolding in the course of the deduction. Obviously, in this simple example it is.

We will now consider a more stable strategy of determining how much unfolding we should perform on goal bundles before their addition as bodies of lemmata.

> Let $G = l_1, \ldots, l_n$ be the goal bundle or the literal (in which case $n = 1$) which was selected for prediction. Choose the first deterministically expandable literal in G, or if none such literal exists, the first literal, and unfold it once. Take the deterministic closure of the result and return it as the body of the lemma to be recorded.

Notice that at most one nondeterministic expansion is performed. The number of generated lemmata in this step will thus be maximally equal to the number of choices that this choice point introduces. Using this strategy in the example above we would get

```
2: bundle1(s/[T1,T2],s,[kim,sleeps],[]) <*=
      bundle1(T1,np,[kim,sleeps],S1), bundle1(T2,vp,S1,[]).
```

instead of the lemmata 2 and 3 from above. Note that `s==>[C]` is deterministic with zero solutions. Furthermore, there are no problems any more with the more general definition of `yield`. The literal `yield(C/[T2],S1,[])` deterministically reduces to `yield(T2,S1,[])` in two steps.

In order to fully explore the constraint propagation effect of deterministic closure it makes sense to use this strategy in the completion step of the deduction algorithm as well. Hence, when a lemma $H \leftarrow l_1, B$ is resolved with a fact l'_1, we generate the lemma $det\_closure(\sigma(H \leftarrow B))$, where $\sigma = mgu(l_1, l'_1)$ and $det\_closure$ computes the deterministic closure of a clause (always with respect to the program P). Actually, this allows also for a simplification of the unfolding strategy above, since now lemma bodies only contain nondeterministic literals.

When using this additional modification the n–queens example now also shows the expected co-routining behaviour when using the following folder declaration

```
bundle2(L,S,N1,DL1,N2,DL2) :-
      permute(L,S),
      check_sw_to_ne(S,N1,DL1),
      check_nw_to_se(S,N2,DL2).
```

```
get_expanded_clause(Goal, Body) :-
    folder(Goal, [FGoal|R]), !,
    FGoal <= FBody,
    append(FBody,R,Body1),
    expand_body1(Body1, Body).

get_expanded_clause(Goal, Body) :-
    Goal <= Body1,
    expand_body1(Body1, Body).

expand_body(Body0, Body) :-
    expand_folders(Body0, Body1),
    expand_body1(Body1, Body).

expand_body1(Body1, Body) :-
    det_closure(Body1,Body2),
    compress_folders(Body2,Body).
```

```
expand_folders([], []).
expand_folders([F|R], R1) :-
    folder(F, Body), !,
    expand_folders(R, R0),
    append(Body, R0, R1).
expand_folders([F|R], [F|R1]) :-
    expand_folders(R, R1).

compress_folders(Body, [F|Rest]) :-
    folder(F, FBody),
    match_body(FBody, Body, B0), !,
    compress_folders(B0, Rest).
compress_folders(Body, Body).
```

Figure 3: PROLOG predicates for Earley deduction with folders

In order to give more precision to the ideas introduced above, we show how the program in Fig. 2 has to be modified to implement the method. In the prediction predicate we call a new predicate instead of accessing the program clauses directly.

```
prediction(Goal,Goal <*= Body) :-
    get_expanded_clause(Goal, Body),
    store(Goal <*= Body).
```

We assume here that the selected literal is always placed in the first position of the body, before an active lemma is stored. Secondly, we hook in a predicate for the expansion of bodies into `p_completion`, as shown below, and analogously into `a_completion`.

```
p_completion(Fact, Goal<*=Body) :-
Goal <*= [Fact|Body0],
expand_body(Body0, Body),
store(Goal <*= Body).
```

Fig. 3 shows the definition of the additional predicates. When prediction is to be performed on a folder predicate, we first substitute it with its body (which we assume to be uniquely defined using the predicate `folder/2`) and then unfold the first literal, otherwise we simply unfold the literal to be predicted on. In both cases the deterministic closure of the resulting body is computed and subsequently split into goal bundles using `compress_folders`. During completion we perform the same last two steps (predicate `expand_body1` on the body after having expanded all folder predicates therein. The heaviest overhead is produced by the determination of the goal bundles in `compress_folders`. A folder definition is only applicable if it actually subsumes the body and this is in general a rather expensive test. Instead of supplying this general test, however, we propose to compile out folder declaration into applicability tests which only access the relevant parts of the structure. For instance, we could have compiled the introduction of the `bundle1` folder above into the predicate:

```
matching_folder(Body, F, NewBody) :-
    del(yield(Tree,S0,S1), Body, B1),
    del(xbar(Tree,Cat), B1, NewBody),
    T==T1, !,
    F=bundle1(Tree,Cat,S0,S1).
```

A call to `matching_folder(Body,F,B0)` now would replace the calls to `folder` and `match_body` in `compress_folders`.

Let us summarize the strategy described in this section. We use folder declarations to bundle together goals which should not be treated in separation. This is necessary, because an important aspect of the Earley deduction method is to split a single literal from the current goal and treat this in isolation. So, we have generalized this goal splitting allowing now for a *set* of literals to be treated in an independent subsidiary proof.

However, we have generalized Earley deduction also in another aspect by allowing multiple unfolding steps in the prediction as well as in the completion rule to act on the clauses fetched from the program database or on the reduced lemma, respectively. We have limited the additional unfolding steps to deterministic ones. But there is in principle no necessity for this limitation. All we need is a method or a parameter that determines when we should stop unfolding and generate the new lemma.

Since this aspect of allowing to switch between Earley deduction, where intermediate goals as well as their solutions are recorded, and ordinary SLD resolution, in which no intermediate lemmas are produced for further use, is an orthogonal generalization to the use of folder declarations, we describe it as an independent modification in the next section.

But before this, let us consider again the question of static vs. dynamic transformation raised above. Since the extent to which unfolding is done during one prediction step depends on the degree of instantiation of the data structures, there seems to be no way to obtain a respective static transformation which doesn't use extralogical (or high-order) devices. Note that the determinism test itself is extralogical. However, it would be interesting to investigate a compilation technique which produced clauses with applicability tests, maybe in form of PROLOG code.

3.2 Earley Deduction with Trigger Goals

As explained above we assume here a modification of Earley deduction in which the body of a lemma, produced either by the prediction or the completion procedure, which would normally be stored, undergoes SLD resolution until a certain criterion is met, before it is stored.[8] The criterion we use is simple. Together with a program we assume the declaration of so-called trigger goals, which are simply single atomic formulae. Now the strategy is the following:

> Before storing a lemma unfold its body until an instance of a trigger goal appears in it. Record the lemma with the trigger goal being the selected literal.

The basic intuition behind this method is that if memoizing ever makes sense for a given program we should be able to tell actually which goals are worth being recorded (together with their solutions). These goals need to be specific enough to be executable in isolation and are declared as trigger goals. Hence, when a trigger goal T (actually an atomic formula) occurs in the current goal, an independent subproof is started just for T with its solution being cached. Notice that, when no trigger goal appears (maybe none are specified) we end up with ordinary SLD resolution for the complete proof. Furthermore, the heads of the stored lemmata can only be instances of trigger goals.

We will now discuss a larger linguistic example with which we show the simplicity and usefulness of the mechanism of trigger goals. Figures 4 and 5 present the kernel of an HPSG-style grammar.

Some remarks on notation are in order here. As mentioned in the introduction we use as an underlying framework the CLP-scheme of Höhfeld and Smolka [HS88]. In contrast to previous examples we assume now that the built-in constraint language is a feature constraint logic as described by [Smo89]. Thus, clauses have the form

$$H \leftarrow B_1, \ldots, B_n, \phi$$

[8] For the SLD resolution component we assume a computation rule implementing the deterministic closure strategy.

```
sign(X) :- headed_phrase(X).
sign(X) :- word(X).

headed_phrase(X) :-
        sign(HD), signs(CD),
        constituent_order_principle(X,HD,CD),
        universal_principles(X,HD,CD),
        rule(X,HD,CD).

universal_principles(X, HD, CD) :-
        subcat_principle(X,HD,CD),
        head_feature_principle(X,HD),
        semantics_principle(X,HD,CD).

subcat_principle(syn:loc:subcat:SX, syn:loc:subcat:SH, CD) :-
        append(CD,SX,SH).

head_feature_principle(syn:loc:head:X, syn:loc:head:X).

semantics_principle(sem:(cont:Con & indices:I ),
                    (HD & sem:cont:SH),
                    CD) :-
         successively_combine_semantics(SH,CD,Con),
         collect_indices([HD|CD],I).

signs([]).
signs([F|R]) :- sign(F), signs(R).

/* language-dependent principles (for phrasal signs) */
constituent_order_principle((phon_in:Pi & phon_out:Po),HD,CD) :-
         permute([HD|CD],OD),
         checklp(OD,[HD|CD]),
         conc_phon(OD,Pi,Po).

conc_phon([],P,P).
conc_phon([phon_in:P & phon_out:Po|RList],Res) :-
         conc_phon(RList,Po,Res).
```

Figure 4: Parametrized version of the HPSG-style approach, principles

where the head H and the body literals B_i are atomic formulae in which only variables may occur as arguments and ϕ is a feature constraint over these variables. However, in order to have a more compact notation we use an abbreviated syntax where feature constraints are notated as feature terms (generalized feature matrices, see [DE91]) occurring in the (argument) place of the variable that they restrict. Thus, if we write out the first clause of `rule` above (see Fig. 5), we obtain (in PATR-II-style notation using first/rest encoding of lists):

```
rule(X, Y, Z) :-
    <X syn loc subcat> = [],
    <Y syn loc lex> = minus,
    <Z rest> = [].
```

This grammar is a straightforward formulation of an HPSG-style principle-based grammar architecture using definite clauses. Signs have features `phon_in`, `phon_out`, `syn` and `sem`. They can

```
/* rules of the grammar */
/* rule 1: yielding saturated signs, only non-lex. heads */
rule(syn:loc:subcat:[], syn:loc:lex:minus, [_]).

/* rule 2: bind compl. non-inverted, lex. heads  */
rule(syn:loc:subcat:[_],
     syn:loc:(head:inv:minus &
              lex:plus),
      _ ).

/* rule 3: inverted phrases */
rule(syn:loc:subcat:[],
     syn:loc:(head:inv:plus &
              lex:plus),
     _ ).
```

Figure 5: Parametrized version of the HPSG-style approach, rules

be either words, in which case the combination of these parts is determined directly by the lexicon relation `word`, or they can be phrases, which are dependent on principles and rules. Principles and rules are formulated as three-place relations of the sign `X` to be "constructed", its head daughter sign `HD` and the list of its complement daughter signs `CD` (in the order of obliqueness).[9] The two features `phon_in` and `phon_out` represent the "phonology" string as a difference list. These two parts of a phrasal sign are constrained by the constituent order principle, which simply states that the phonology of the mother sign `X` consists of the concatenated phonology parts of a list of signs which is an lp-admissible permutation of the list `[HD|CD]`. Similarly the subcategorization and head feature principle constrain the combined instantiation of mother sign `X`, head daughter sign `HD` and the list of complement daughter signs `CD` in the expected way. These principles can be seen as the factored information that is common to all admissible local tree configurations, whereas the `rule` predicate enumerates different possible configurations.

A reasonable strategy for parsing HPSG grammars is a head-driven strategy, because the subcategorization list of the head constituent determines how much and what complements there are in a local tree. Since the head need not appear as the first daughter, one has to guess the string position where the head starts. Unfortunately we cannot simulate this strategy in an SLD-resolution proof using the grammar above, because the string is consumed only by the lexical sign goals. However, with a small modification, an addition of a redundant literal that makes explicit the fact that the `phon_out` feature of a sign must be a tail of the list of its `phon_in` feature, we are able to delay the evaluation of signs corresponding to complements located on the left of the head. We implement this modification in the clause of `constituent_order_principle`:

```
constituent_order_principle((phon_in:Pi & phon_out:Po),
                            (HD & phon_in:PH),CD) :-
          suffix(PH,Pi),
          permute([HD|CD],OD),
          checklp(OD,[HD|CD]),
          conc_phon(OD,Pi,Po).

suffix(L,L).
suffix(L,[_|R]) :- suffix(L,R).
```

[9] This formulation of principles and rules is superior in terms of efficiency to a formulation using "daughters" features, since it allows us to build much smaller structures, while keeping the logic of the grammar the same (see also the discussion in [DE91]).

Using this grammar one can arrange that deduction follows a (top-down) head-driven strategy by delaying goals of predicate `sign` which are not instantiated for both `phon_in` and `syn:loc:head`. The resulting strategy relies only on the facts that lexical entries have instantiated (possibly empty) subcategorization lists and that the rules don't allow arbitrary embeddings of heads.

It is now reasonable to use the `sign` goals which are sufficiently instantiated to be undelayed as triggers. These goals can be evaluated independently from their contexts. Actually, it is a rather trivial observation that we are able to parse, if we have hold of the string to parse and additionally know some head information. What makes the whole thing so complicated is that we have to make sure that such goals will eventually show up in the proof and we might not run into a loop in which such goals never occur.

Another important advantage of the declaration of trigger goals is that these declarations tell us how far goals are at least instantiated for which we have to perform a table lookup. We use this information in the compiler to create efficient code for indexed access of the lemma table.

3.2.1 Combining the Approaches

The strategy described in this section so far allowed triggers to be only single literals, not sets of literals. This is sensible as long as only single literals are separated in the predictor step to initiate isolated subproofs. However, if we want to use in parallel goal bundles specified with folder declarations, we should also be able to switch off unfolding when an instance of a goal bundle occurs in a lemma body, since the solutions to this bundle might be worth memoizing. In other words, we also want to allow folders as triggers. Note that for these we only need a more complicated check whether a trigger has occurred, since this can then only be checked by considering combinations of literals. Nothing else has to be changed.

4 Dealing with Disjunctive Information

In this section we would like to focus on the question how disjunctive information is treated within implementations of constraint-based deduction like the above. We want to elucidate the potential that a parsing-as-deduction approach based on our generalized Earley deduction has in this respect.

The most important aspect here is probably the possibility to represent and deal with ambiguity in three different layers. Firstly, there is the layer of the *built-in constraint language*, which we used for the description of feature structures. On this layer the satisfiability of the accumulated constraints of this type are checked in each resolution step. It is useful to allow for a limited form of disjunction here, as it is done in CUF (cf. [DD93]). The constraints dealt with on this level in CUF are (conjunctive) type and the feature constraints. Since types can be set up as disjunctions of other types, this form of underspecification is a form which is representable on this layer of typed feature structures. For instance, we could represent here information like

$$\mathit{case} : (\mathit{nom} \vee \mathit{acc}).$$

Second, on the layer of the SLD resolution steps which are performed between the caching of lemmata ambiguity turns up as nondeterministicly expanding literals. The strategy used here is in the first place delaying and, only if we are forced, considering choice combinations by chronological backtracking.

The third mode of representing disjunctive information is the lemma cache. Clearly the purpose of this layer is after all the *factored representation* of solutions of parts of the goals.

Each of these three different levels for representing disjunctive information has its own method for using this information. On the lowest level an eager satisfiability check ensures that information entered in this level is always consistent. Since efficiency on this level naturally is a prerequisite for overall efficiency we have to be very careful not to overload this layer with consistency checks of too complex disjunctions.

The second layer is in contrast to the first very lazy concerning the use of disjunctive information. It is normally an advantage to delay nondeterministic goals as long as possible. Also the backtracking method used on this level can be very space and time efficient compared to other methods determining the compatible combinations of disjunctive information. But even with a deterministic closure strategy it is still efficiency-critical in which order the computation rule selects literals which introduce choice points. Here a fine-tuning of the computation rule using, e.g., delay declarations as in CUF, is very helpful to control the search.

Finally the third layer, the lemma table, provides a means for factoring disjunctive information on the goal level. We thus can avoid redundant recomputations of the same goals over and over again.

The approach now allows for the flexibility to shift disjunctive information between the different layers, depending on how it is formulated. Note that we always can raise a disjunction from the constraint language level to the goal level by introducing a new unary predicate. Also the declaration of more or less triggers can balance the load between the backtracking and the memoizing component. Hence, we have reason for the hope that this flexibility enables us to build more efficient natural language systems than with the traditional architectures for constraint-based grammar formalisms.

5 Johnson's lemma table proof procedure

Mark Johnson has independently worked out an approach [Joh93] that seems in its effect to be very similar to our method, although superficially it looks different. His method is essentially a generalization of the OLDT proof procedure [TS86], which is like Earley deduction a derivative of SLD-resolution allowing the memoization (caching) of intermediate goals and solutions.

The basic informational units in Johnson's method are the so-called *generalized clauses*. These have the form $G_1 \leftarrow G_2$, where G_1 and G_2 are goals, i.e. sets of atoms. Both these sets are interpreted conjunctively: "if each of the G_2 are true, then all of the G_1 are true". Lemmata (table entries) are triples $\langle G, T, S \rangle$ where G is a goal, T is a partial proof tree for G, and S is the so-called *solution list* for G, which is a list of clauses $G' \leftarrow C$ where G' is an instance of G. The partial proof tree T of a lemma is constructed incrementally top-down by starting with a root node labeled $G \leftarrow G$ and applying a computation rule R (which is a parameter to the method) to yet unprocessed nodes until all nodes have been processed. The rule R specifies one of three operations to be performed on a node with label $A \leftarrow B$:[10]

program(b): (where $b \in B$) An SLD-resolution step against the program is performed on $A \leftarrow B$ with b being the selected literal. For every matching program clause a daughter node is created with the respective resolvent $\sigma(A \leftarrow B')$ as label. On root nodes only an operation of this type is allowed.

table(B'): (where $B' \subseteq B$) Here a resolution step is performed against solutions of a lemma E for some goal that subsumes B' (which is created if not already present). For every solution $B'' \leftarrow C$ of E the respective daughter node carries the label $\sigma(A \leftarrow (B - B') \cup C)$, where $\sigma = mgu(B', B'')$.

solution: The node's label $A \leftarrow B$ is added to the solution list. The node receives no children. This is the only possible operation if B is empty.

We observe that the resolution steps performed in a `program(b)` operation correspond exactly to our unfolding steps of lemma bodies using SLD resolution. Also when a `table(B)` or a `solution` operation is encountered here, this corresponds to the storing of a new lemma in the Earley deduction part of our algorithm. Moreover, like in our method a whole set of literals is isolated and proven

[10] To ease understanding we omitted the pointer management which is needed to ensure that all solutions which once will be computed for a table entry will be considered when it is used for resolution in a `table(B)` operation.

in a subsidiary proof. Actually, when using a computation rule that simulates our choices of when to apply a program or a table operation, then the constructed lemmata in our method are (modulo substitution of folders) just the node labels of `table(B)` or `solution` nodes.

An interesting feature of Johnson's method is that the computation rule allows for a more flexible notion of solution constraints, although he leaves open what a sensible choice for these would be like. In our method only the empty resolvent, i.e. when only (solved) built-in constraints are left in the antecedent, counts as a solution. There is also another aspect in which his method is more general than ours. The explicit representation of proof trees even for the SLD resolution steps makes it possible to work on more than one such tree in parallel. In this respect our method can be regarded as a specific implementation of Johnson's. We assume that when unfolding a lemma body, the set of all such unfoldings is computed at once. In the view of Johnson's method we could paraphrase that as working on one tree as long as there are unprocessed program nodes. This specific choice of the evaluation order allows for the well-known efficient implementation techniques of depth-first search with backtracking. We thus can avoid computing copies of clauses for program nodes. We only have to copy at table or solution nodes.

To summarize, the new method by Mark Johnson presents solutions to the problem of allowing constraint propagation in tabled deduction which aim into the same direction as the techniques we have discussed above, namely to use sets of literals rather than single literals for isolated subproofs and to allow for a flexible switching between uncached and cached resolution steps. In some respects Johnson's method is more general than ours.

The computation rule, for instance, is a very general device for configuring the actual strategy of the proof procedure. Johnson, however, does not discuss what sensible choices for the computation rule would be like. In a practical implementation one would certainly not assume this being an input to the algorithm. Rather one would support certain fixed, perhaps parametrizable strategies between which one can switch and which allow for more efficient data structures to be used compared to the general case.

Our triggers and folders can be such parameters which control a more restricted computation rule. Trigger goals tell the computation rule when to perform a `table(B)` operation rather than `program(b)` and folder declarations specify what goal bundles are allowed and help to view them as a single literal.

6 Conclusion

We have presented generalizations of Earley deduction which allow techniques of the processing of constraints such as goal delaying or co-routining to be combined with tabled deduction. An important aspect of our work is that the actual strategy of the proof procedure is to a high degree determined by parameters which are given as control declarations. Thus, we can fine-tune the strategy to the specific needs of the program (grammar) in question. In contrast to [Joh93], where recently a very similar and in some respects more general method was devised, we focus here on issues of efficient implementation.

A first experimental implementation, which added a generalized Earley deduction engine to the CUF system, has been carried out by Simone Teufel in her Studienarbeit [Teu93] and showed promising results. Due to the flexibility that the specification of trigger goals add to the control the overhead of the Earley deducer could be compensated already for a relatively small grammar[11] and short unambiguous sentences. We hope to even be able to improve over this in an implementation currently being undertaken. Further experimentation, also with larger programs, however, is needed to estimate the usefulness of control devices like triggers and folders in practice.

[11] essentially the one of Fig. 4 and 5 together with a small lexicon

References

[AKP91] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. PRL Research Report 11, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France, 1991.

[DD93] Jochen Dörre and Michael Dorna. CUF — a formalism for linguistic knowledge representation. DYANA deliverable, 1993. This volume.

[DE91] Jochen Dörre and Andreas Eisele. A Comprehensive Unification-Based Grammar Formalism. DYANA Deliverable R3.1.B, ESPRIT Basic Research Action BR3175, Jan. 1991.

[EZ90] Martin Emele and Rémi Zajac. Typed unification grammars. In *Proceedings of the 13th International Conference on Computational Linguistics*, Helsinki, Finland, 1990.

[HS88] Markus Höhfeld and Gert Smolka. Definite relations over constraint languages. LILOG Report 53, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, W. Germany, October 1988.

[Joh93] Mark Johnson. Memoization in constraint logic programming. ms., Department of Cognitive Science, Brown University, 1993. Presented at the 1st International Conference on Constraint Programming, Newport, Rhode Island; to appear in the Proceedings.

[PS87] Fernando C.N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis.* CSLI Lecture Notes 10. Center for the Study of Language and Information, Stanford University, 1987.

[PW83] Fernando C.N. Pereira and David H.D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the ACL, Massachusetts Institute of Technology*, pages 137–144, Cambridge, Mass., 1983.

[Smo89] Gert Smolka. Feature constraint logics for unification grammars. IWBS Report 93, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, W. Germany, November 1989. To appear in: Wedekind, Rohrer (eds.), Unification in Grammar, MIT Press, 1992.

[Smo91] Gert Smolka. Residuation and guarded rules for constraint logic programming. DFKI report RR–91–13, DFKI, Stuhlsatzenhausweg 3, D–6600 Saarbrücken, Germany, 1991.

[Teu93] Simone Teufel. Einbindung spezialisierter parsing-strategien in einen feature-logik-formalismus. Studienarbeit Nr. 1188, Institut für Informatik, Universität Stuttgart, 1993.

[TS86] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proceedings of the Third International Conference on Logic Programming*, pages 84–98, Berlin, 1986. LCNS 225 Springer-Verlag.

[Zaj92] Rémi Zajac. Inheritance and constraint-based grammar formalisms. *Computational Linguistics*, 18(2):159–182, June 1992.