Memoization in Constraint Logic Programming

Mark Johnson*
Brown University

1   Introduction

This paper shows how to apply memoization (caching of subgoals and associated answer substitutions) in a constraint logic programming setting. The research is is motivated by the desire to apply constraint logic programming (CLP) to problems in natural language processing.

In general, logic programming provides an excellent theoretical framework for computational linguistics [13]. CLP extends "standard" logic programming by allowing program clauses to include constraints from a specialized contraint language. For example, the CLP framework allows the feature-structure constraints that have proven useful in computational linguistics [15] to be incorporated into logic programming in a natural way [1, 5, 16].

Because modern linguistic theories describe natural language syntax as a system of interacting "modules" which jointly determine the linguistic structures associated with an utterance [2], a grammar can be regarded as a conjunction of constraints whose solutions are exactly the well-formed or grammatical analyses. Parsers for such grammars typically coroutine between a tree-building component that generates nodes of the parse tree and the well-formedness constraints imposed by the linguistic modules on these tree structures [4, 6, 8]. Both philosophically and practically, this fits in well with the CLP approach.

But the standard CLP framework inherits some of the weaknesses of the SLD resolution procedure that it is based on. When used with the standard formalization of a context-free grammar the SLD resolution procedure behaves as a recursive descent parser. With left-recursive grammars such parsers typically fail to terminate because a goal corresponding to a prediction of a left-recursive category can reduce to an identical subgoal (up to renaming), producing an "infinite loop". Standard techniques for left-recursion elimination [12, 13] in context-free grammars are not always directly applicable to grammars formulated as the conjunction of several constraints [10].

With memoization, or the caching of intermediate goals (and their corresponding answer substitutions), a goal is solved only once and its solutions are cached; the solutions to identical goals are obtained from this cache. Left recursion need not lead to non-termination because identical subgoals are not evaluated, and the infinite loop is avoided. Further, memoization can sometimes provide the advantages of dynamic programming approaches to parsing: the Earley deduction proof procedure (a memoized version of SLD resolution) simulates an Earley parse [3] when used with the standard formalization of a context-free grammar [14].

```
parse(String, Tree) :- wf(Tree, s), y(Tree, String, []).

y(_-Word, [Word|Words], Words).
y(_/[Tree1], Words0, Words) :- y(Tree1, Words0, Words).
y(_/[Tree1,Tree2], Words0, Words) :-
      y(Tree1, Words0, Words1), y(Tree2, Words1, Words).

wf(np-kim, np).                                    % NP → Kim
wf(n-friend, n).                                   % N → friend
wf(v-walks, v).                                    % V → walks
wf(s/[Tree1, Tree2], s) :- wf(Tree1, np), wf(Tree2, vp).    % S → NP VP
wf(np/[Tree1, Tree2], np) :- wf(Tree1, np), wf(Tree2, n).   % NP → NP N
wf(vp/[Tree1], vp) :- wf(Tree1, v).                % VP → V
```

Figure 1: A grammar fragment

Thus constraint logic programming and memoization are two recent developments in logic programming that are important for natural language processing. But it is not obvious how, or even if, the two can be combined in a single proof procedure. For example, both Earley Deduction [14] and OLDT resolution [17, 20] resolve literals in a strict left-to-right order, so they are not capable of rudimentary constraint satisfaction techniques such as goal delaying. The strict left-to-right order restriction is relaxed but not removed in [18, 19], where literals can be resolved in any local order. This paper describes soundness and completeness proofs for a proof procedure that extends these methods to allow for goal delaying. In fact, the lemma table proof procedure generalizes naturally to constraint logic programming over arbitrary domains, as described below.

The lemma table proof procedure generalizes Earley Deduction and OLDT resolution in three ways.

- Goals can be resolved in any order (including non-local orders), rather than a fixed left-to-right or a local order.

- The goals entered into the table consist of non-empty sets of literals rather than just single literals. These sets can be viewed as a single program literal and zero or more constraints that are being passed down into the subsidary proof.

- The solutions recorded in the lemma table may contain unresolved goals. These unresolved goals can be thought of as constraints that are being passed out of the subsidary proof.

2   A linguistic example

Consider the grammar fragment in Figure 1 (cf. also [4, pages 142–177]). The parse relation holds between a string and a tree if the yield of the tree is the string to be parsed and tree satisfies a well-formedness condition. In this example, the well-formedness condition is that the tree is generated by a simple context-free grammar, but in more realistic fragments the constraints are considerably more complicated.

Trees are represented by terms. A tree consisting of a single pre-terminal node labelled C whose single child is the word W is represented by the term C-W. A tree consisting of a root node labelled C dominating the sequence of trees $T_1 \ldots T_n$ is represented by the term $C/[T_1, \ldots, T_n]$.

The predicate wf(*Tree, Cat*) holds if *Tree* represents a well-formed parse tree with a root node labelled *Cat* for the context free grammar shown in the comments. The predicate y(*Tree, S0, S*) holds if *S0–S* is a "difference list" representing the yield of *Tree*; it collects the terminal items in the familiar tree-walking fashion. From this program, the following instances of parse can be deduced (these are meant to approximate possessive constructions like *Kim's friend's friend walks*).

```
parse([kim,walks], s/[np-kim,vp/[v-walks]]).
parse([kim,friend,walks], s/[np/[np-kim,n-friend],vp/[v-walks]]).
parse([kim,friend,friend,walks],
s/[np/[np/[np-kim,n-friend],n-friend],vp/[v-walks]]).
. . .
```

The parsing problem is encoded as a goal as follows. The goal consists of an atom with the predicate parse whose first argument instantiated to the string to be parsed and whose second argument is uninstantiated. The answer substitution binds the second argument to the parse tree. For example, an answer substitution for the goal parse([Kim,friend,walks], Tree) will have the parse tree for the string *Kim friend walks* as the binding for Tree.

Now consider the problem of parsing using the program shown in Figure 1. Even with the variable String instantiated, both of the subgoals of *parse* taken independently have an infinite number of subsumption-incomparable answer substitutions. Informally, this is because there are an infinite number of trees generated by the context-free grammar, and there are an infinite number of trees that have any given non-empty string as their yield. Because the standard memoization techniques mentioned above all compute all of the answer substitutions to every subgoal independently, they never terminate on such a program.

However, the set of answer substitions that satisfy both constraints is finite, because the number of parse trees with the same yield with respect to this grammar is finite. The standard approach to parsing with such grammars takes advantage of this by using a selection rule that "coroutines" the goals wf and y, delaying all wf goals until the first argument is instantiated to a non-variable. In such a system, the wf goals function as constraints that filter the trees generated by the goals y.

In this example, however, there is a second, related, problem. Coroutining is not sufficient to yield a finite SLD tree, even though the number of refutations is finite. Informally, this is because the grammar in Figure 1 is left recursive, and the search space for a recursive descent parser (which an SLD refutation mimics with such a program) is infinite.

Figure 2 shows part of an infinite SLD derivation from the goal parse(KW, T), where KW is assumed bound to [kim,walks] (although the binding is actually immaterial, as no step in this refutation instantiates this variable). The selection rule expresses a "preference" for goals with certain arguments instantiated. If there is a literal of the form wf(T, C) with T instantiated to a non-variable then the left-most such literal is selected, otherwise if there is a literal of the form y$(T, S0, S)$ with $S0$ instantiated to a non-variable then the left-most such literal is selected, otherwise the left-most literal is selected. The selected literal is underlined, and the new literals introduced by each reduction are inserted to the left of the old literals.

Note that the y and wf literals resolved at steps (6) and (7) in Figure 2 are both children

```
(1)  parse(KW,T).
(2)  wf(T,s),  y(T,KW,[]).
(3)  wf(_/[T1,T2],s),  y(T1,KW,S1),  y(T2,S1,[]).
(4)  wf(T1,np),  wf(T2,vp),  y(T1,KW,S1),  ...
(5)  y(T3,KW,S3),  y(T4,S3,S1),  wf(_/[T3,T4],np),  ...
(6)  wf(T3,np),  wf(T4,n),  y(T3,KW,S3),  ...
(7)  y(T5,KW,S5),  y(T6,S5,S3),  wf(_/[T5,T6],np),  ...
        ...
```

Figure 2: An infinite SLD refutation, despite co-routining

of and variants of the literals resolved at steps (5) and (6). This sequence of resolution steps can be iterated an arbitrary number of times. It is a manifestation of the left recursion in the well-formedness constraint wf.

One standard technique for dealing with such left-recursion is memoization [14, 13]. But there are two related problems in applying the standard logic programming memoization techniques to this problem.

First, because the standard methods memoize and evaluate at the level of an individual literal, the granularity at which they apply memoization is is to small. As noted above, in general individual wf or y literal can have an infinite number of answer substitutions. The lemma table proof procedure circumvents this problem by memoizing *conjunctions* of literals (in this example, a conjunction of wf and y literals which has only a finite number of subsumption-incomparable valid instances).

The second problem is that the standard memoization techniques restrict the order in which literals can be resolved. In general, these restrictions prevent the "goal delaying" required to co-routine among several constraints. The lemma table proof procedure lifts this restriction by allowing arbitrary selection rules.

3   The Lemma Table proof procedure

Like the Earley Deduction and the OLDT proof procedures, the Lemma Table proof procedure maintains a lemma table that records goals and their corresponding solutions. After a goal has been entered into the lemma table, other occurences of instances of that goal can be reduced by the solutions from the lemma table instead of the original program clauses.

We now turn to a formal presentation of the Lemma Table proof procedure. In what follows, lower-case letters are used for variables that range over atoms. Upper-case letters are used for variables that range over *goals*, which are sets of atoms. Goals are interpreted conjunctively; a goal is satisfied iff all of its members are.

A goal G *subsumes* a goal G' iff there is some substitution θ such that $G' = G\theta$. (Note that m.g.u.'s for sets of goals are in general not unique even up to renaming).

The "informational units" manipulated by the lemma table proof procedure are called generalized clauses. A *generalized clause* is a pair of goals, and is written $G_1 \leftarrow G_2$. G_1 is called the *head* of the clause and G_2 is called the *body*. Both the head and body are interpreted conjunctively; i.e., $G_1 \leftarrow G_2$ should be read as "if each of the G_2 are true, then all of the G_1 are true". A generalized

4

clause has a natural interpretation as a goal subject to constraints: G_1 is true in any interpretation which satisfies the constraints expressed by G_2.

A *lemma table* is a set of *table entries* of the form $\langle G, T, S \rangle$, where

1. G is a goal (this entry is called a table entry *for G*),

2. T is a lemma tree (see below), and

3. S is a sequence of clauses, called the *solution list* for this entry.

A *lemma tree* is a tree constructed by the algorithm described below. Its nodes have two labels. These are

1. a clause $A \leftarrow B$, called the *clause labelling* of the node, and

2. an optional tag, which when present is one of solution, program(b) for some $b \in B$, or table(B', p) where $\emptyset \subset B' \subseteq B$ and p is either the null pointer nil or a pointer into a solution list of a table entry for some G that subsumes B'.

Untagged nodes are nodes that have not yet been processed. All nodes are untagged when they are created, and they are assigned a tag as they are processed. The tags indicate which kind of resolution has been applied to this clause. A node tagged program(b) is resolved against the clauses defining b in the program. A node tagged table(B', p) is resolved against the instances of a table entry E for some goal that subsumes B'; the pointer p keeps track of how many of the solutions from E have been inserted under this node (just as in OLDT resolution). Finally, a node tagged solution is not resolved, rather its clause labelling is added to the solution list for this table entry.

Just as SLD resolution is controlled by a selection rule that determines which literal will be reduced next, the Lemma Table proof procedure is controlled by a control rule R which determines the next goal (if any) to be reduced and the manner of its reduction.

More precisely, R must tag a node with clause labelling $A \leftarrow B$ with a tag that is either solution, program(b) for some $b \in B$, or table(B', nil) such that $\emptyset \subset B' \subseteq B$. Further, R must tag the root node of every lemma tree with the tag program(b) for some b (this ensures that some program reductions are performed in every lemma tree, and hence that a lemma table entry cannot be used to reduce itself vacuously).

Finally, as in OLDT resolution, the Lemma Table proof procedure allows a user-specified *abstraction operation* α that maps goals to goals such that $\alpha(G)$ subsumes G for all goals G. This is used to generalize the goals in the same way as the term-depth abstraction operation in OLDT resolution, which it generalizes.

The Lemma Table proof procedure can now be presented.

Input: A non-empty goal G, a program P, an abstraction operation α, and a control rule R.

Output: A set γ of clauses of the form $G' \leftarrow C$, where G' is an instance of G.

Algorithm: Create a lemma table with one table entry $\langle G, T, [] \rangle$, where T contains a single untagged node with the clause labelling $G \leftarrow G$. Then repeat the following operations until no operation applies. Finally, return the solution list from the table entry for G.

The operations are as follows.

Prediction Let v be an untagged node in a lemma tree T of table entry $\langle G, T, S \rangle$, and let v's clause labelling be $A \leftarrow B$. Apply the rule R to v, and perform the action specified below depending on the form of the tag R assigned to v.

solution : Add $A \leftarrow B$ to the end of the solution list S.

program(b) : Let $B' = B - \{b\}$. Then for each clause $b' \leftarrow C$ in P such that b and b' unify with a m.g.u. θ, create an untagged child node v' of v labelled $(A \leftarrow B' \cup C)\theta$.

table(B', nil) : The action in this case depends on whether there already is a table entry $\langle G', T', S' \rangle$ for some G' that subsumes B'. If there is, set the pointer in the tag to the start of the sequence S'. If there is not, create a new table entry $\langle \alpha(B'), T'', [] \rangle$, where T'' contains a single untagged node with clause labelling $\alpha(B') \leftarrow \alpha(B')$. Set the pointer in v's tag to point to the empty solution list of this new table entry.

Completion Let v be a node with clause labelling $A \leftarrow B$ and tagged table(B', p) such that p points to a non-null portion S' of a solution list of some table entry. Then advance p over the first element $B'' \leftarrow C$ of S' to point to the remainder of S'. Further, if B' and B'' unify with m.g.u. θ, then add a new untagged child node to v labelled $(A \leftarrow (B - B') \cup C)\theta$.
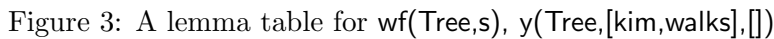
It may help to consider an example based on the program in Figure 1. The computation rule R used is the following. Let v be a node in a lemma tree and let $A \leftarrow B$ be its clause label. If v is the root of a lemma tree and B contains a literal of the form $\mathsf{y}(T, S0, S)$ then $R(v) = \mathsf{program}(\mathsf{y}(T, S0, S))$. If B is empty then $R(v) = \mathsf{solution}$ (no other tagging is possible for such nodes). If B contains a literal of the form $\mathsf{wf}(T, C)$ where T is a non-variable then $R(v) = \mathsf{program}(\mathsf{wf}(T, C))$ (there is never more than one such literal). Otherwise, if B contains two literals of the form $\mathsf{wf}(T, C), \mathsf{y}(T, S0, S)$ where $S0$ is a non-variable, then $R(v) = \mathsf{table}(\{\mathsf{wf}(T, C), \mathsf{y}(T, S0, S)\}, \text{nil})$. These four cases exhaust all of the node labelling encountered in the example.[1]

Figure 3 depicts the completed lemma table constructed using the rule R for the goal $\mathsf{wf}(\mathsf{Tree},\mathsf{s})$, $\mathsf{y}(\mathsf{Tree},[\mathsf{kim},\mathsf{walks}],[])$. To save space, kim and walks are abbreviated to k and w respectively. Only the tree from each table entry is shown because the other components of the entry can be read off the tree. The goal of each table entry is the head of the clause labelling its root clause, and is shown in bold face. Solution nodes are shown in italic face. Lookup nodes appear with a dashed line pointing to the table entry used to reduce them.

The first few steps of the proof procedure are the following; each step corresponds to the circled node of the same number.

(1) The root node of the first table entry's tree restates the goal to be proven. Informally, this node searches for an S located at the beginning of the utterance. The literal $\mathsf{y}(\mathsf{Tree},[\mathsf{kim},\mathsf{walks}],[])$ is selected for program reduction. This reduction produces two child nodes, one of which "dies" in the next step because there are no matching program nodes.

(2) The reduction (1) partially instantiates the parse tree Tree. The literal $\mathsf{wf}(\mathsf{C}/[\mathsf{T1},\mathsf{T2}])$ is selected for program reduction.

---

[1] When used with a program encoding a context-free grammars in the manner of Figure 1, the Lemma Table proof procedure with the control rule R simulates Earley's CFG parsing algorithm [3]. The operations in the Lemma Table proof procedure are named after the corresponding operations of Earley's algorithm.

wf(Tree,s), y(Tree,[k,w],[]) ←wf(Tree,s), <u>y(Tree,[k,w],[])</u> ①

wf(C/[T1],s), y(C/[T1],[k,w],[]) ←   wf(C/[T1,T2],s), y(C/[T1,T2],[k,w],[]) ←   ②
  <u>wf(C/[T1],s)</u>, y(T1,[k,w],[])   <u>wf(C/[T1,T2],s)</u>, y(T1,[k,w],S1), y(T2,S1,[])
|

wf(s/[T1,T2],s), y(s/[T1,T2],[k,w],[]) ←   ③
  <u>wf(T1,np)</u>, wf(T2,vp), <u>y(T1,[k,w],S1)</u>, y(T2,S1,[])
|

wf(s/[np-kim,T2],s), y(s/[np-kim,T2],[k,w],[]) ← ⑧
  <u>wf(T2,vp)</u>, <u>y(T2,[w])</u>
|

wf(s/[np-k,vp/[v-w]],s), y(s/[np-k,vp/[v-w]],[k,w],[]) ← ∅

wf(T,np), y(T,[k,w],S) ← wf(T,np), <u>y(T,[k,w],S)</u> ④

wf(C-k,np), y(C-k,[k,w],[w]) ← ⑤   ⑥   wf(C/[T1,T2],np), y(C/[T1,T2],[k,w],S) ←
  <u>wf(C-k,np)</u>   <u>wf(C/[T1,T2],np)</u>, y(T1,[k,w],S1), y(T2,S1,S)
| |

wf(np-k,np), y(np-k,[k,w],[w]) ← ∅ ⑦   ⑨ wf(np/[T1,T2],np), y(np/[T1,T2],[k,w],S) ←
  <u>wf(T1,np)</u>, wf(T2,n), <u>y(T1,[k,w],S1)</u>, y(T2,S1,S)
|

wf(C/[T1],np), y(C/[T1],[k,w],S) ←   wf(np/[np-k,T2],np), y(np/[np-k,T2],np) ←
  <u>wf(C/[T1],np)</u>, y(T1,[k,w],S)   <u>wf(T2,n)</u>, <u>y(T2,[w],S)</u>

wf(T,n), y(T,[w],S) ← wf(T,n), <u>y(T,[w],S)</u>

wf(C-w,n), y(C-w,[w],[]) ←   wf(C/[T1,T2],n), y(C/[T1,T2],[w],S) ←
  <u>wf(C-w,n)</u>   <u>wf(C/[T1,T2],n)</u>, y(T1,[w],S1), y(T2,S1,S)

wf(C/[T1],n), y(C/[T1],[w],S) ←
  <u>wf(C/[T1],n)</u>, y(T1,[w],S)

wf(T,vp), y(T,[w],S) ← wf(T,vp), <u>y(T,[w],S)</u>

wf(C-w,vp), y(C-w,[w],[]) ←   wf(C/[T1],vp), y(C/[T1],[w],S) ←
  <u>wf(C-w,vp)</u>   <u>wf(C/[T1],vp)</u>, y(T1,[w],S)
|

wf(C/[T1,T2],vp), y(C/[T1,T2],[w],S) ←   wf(vp/[T1],vp), y(vp/[T1],[w],S) ←
  <u>wf(C/[T1,T2],vp)</u>, y(T1,[w],S1), y(T2,S1,S)   <u>wf(T1,v)</u>, <u>y(T1,[w],S)</u>
|

wf(vp/[v-w],vp), y(vp/[v-w],[w],[]) ← ∅

wf(T,v), y(T,[w],S) ← wf(T,v), <u>y(T,[w],S)</u>

wf(C-w,v), y(C-w,[w],[]) ←   wf(C/[T1,T2],v), y(C/[T1,T2],[w],S) ←
  <u>wf(C-w,v)</u>   <u>wf(C/[T1,T2],v)</u>, y(T1,[w],S1), y(T2,S1,S)
|

wf(v-w,v), y(v-w,[w],[]) ← ∅   wf(C/[T1],v), y(C/[T1],[w],S) ←
  <u>wf(C/[T1],v)</u>, y(T1,[w],S)

Figure 3: A lemma table for wf(Tree,s), y(Tree,[kim,walks],[])

(3) This produces a clause body that contains literals that refer to the subtree T1 and literals that refer to the subtree T2. The computation rule in effect partitions the literals and selects those that refer to the subtree T1 (because they are associated with an instantiated left string argument).

(4) A new table entry is created for the literals selected in (3). Informally, this entry searches for an NP located at the beginning of the utterance. Because this node is a root node, the literal y(T,[kim,walks],S) is selected for program expansion.

(5–6) The literals with predicate wf are selected for program expansion.

(7) The body of this node's clause label is empty, so it's label is added to the solutions list of the table entry. Informally, this node corresponds to the string [kim] having been recognized as an NP.

(8) The solution found in (7) is incorporated into the tree beneath (3). A new table entry is generated to search for a VP spanning the string [walks].

(9) Just as in (3), the literals in the body of this clause's label refer to two distinct subtrees, and as before the computation rule selects the literals that refer to T1. However there is already a table entry (4) for the selected goal, so a new table entry is not created. The solutions already found for (4) generate a child node to search for an N beginning at walks. No solutions are found for this search.

4 Soundness and Completeness

This section demonstrates the soundness and completness of the lemma table proof procedure. Soundness is straight-forward, but completeness is more complex to prove. The completeness proof relies on the notion of an unfolding of a lemma tree, in which the table nodes of a lemma tree are systematically replaced with the tree that they point to. In the limit, the resulting tree can be viewed a kind of SLD proof tree, and completeness follows from the completeness of SLD resolution.

Theorem 1 (Soundness) *If the output of lemma table proof procedure contains a clause $G \leftarrow C$, then $P \models C \rightarrow G$.*

Proof: Each of the clause labels on lemma tree nodes is either a tautology or derived by resolving other clause labels and program clauses. Soundness follows by induction on the number of steps taken by the proof procedure. ∎

As might be expected, the completeness proof is much longer than the soundness proof. For space reasons it is only sketched here.

For the completeness proof we assume that the control rule R is such that the output γ of the lemma table proof procedure contains only clauses with empty bodies. This is reasonable in the current context, because non-empty clause bodies correspond to goals that have not been completely reduced.

Then completeness follows if for all P, G and σ, if $P \models G\sigma$ then there is an instance G' on the solution list that subsumes $G\sigma$.

Further, without loss of generality the abstraction operation α is assumed to be the identity function on goals, since if $\alpha(G(\vec{t})) = G(\vec{t'})$, the goal $G(\vec{t})$ can be replaced with the equivalent $G(\vec{t'}) \cup \{\vec{t'} = \vec{t}\}$ and α taken to be the identity function.

Now, it is a corollary of the Switching Lemma [11, pages 45–47] that if $P \models G\sigma$ then there is an n such that for any computation rule there is an SLD refutation of G of length n whose computed answer substitution θ subsumes σ. We show that if θ is a computed answer substitution for an SLD derivation of length n then there is a node tagged solution and labelled $G\theta \leftarrow \emptyset$ in lemma tree T for the top-level goal.

First, a well-formedness condition on lemma trees is introduced. Every lemma tree in a lemma table at the termination of the lemma table proof procedure is well-formed. Well-formedness and the set of nodes tagged solution are preserved under an abstract operation on lemma trees called expansion. The expansion of a lemma tree is the tree obtained by replacing each node tagged table(B', p) with the lemma tree in the table entry pointed to by p.

Because expansions preserves well-formedness, the lemma tree T' resulting from n iterated expansions of the lemma tree T for the top-level goal is also well-formed. Moreover, since the root node of every lemma tree is required to be tagged program, all nodes in T' within distance n arcs of the root will be tagged program or solution. This top part of T' is isomorphic to the top part of an SLD tree T_s for G, so if θ is a computed answer substitution for an SLD derivation in T_s of length n or less then there is a node tagged solution and labelled $G\theta \leftarrow \emptyset$ in T', and hence T. Since n was arbitrary, every SLD refutation in T_s has a corresponding node tagged solution in T' and hence in T.

5   Conclusion

This paper generalizes standard memoization techniques for logic programming to allow them to be used for constraint logic programming. The basic informational unit used in the Lemma Table proof procedure is the generalized clause $G \leftarrow C$. Generalized clauses can be given a constraint interpretation as "any interpretation which satisfies the constraints C also satisfies G". The lemmas recorded in the lemma table state how sets of literals are reduced to other sets of literals. Because the heads of the lemmas consist of sets of literals rather than just individual literals, the lemmas express properties of systems of constraints rather than just individual constraints. Because the solutions recorded in the lemma table can contain unresolved constraints, it is possible to pass constraints out of a lemma into the superordinate computation.

In this paper G and C were taken to be sets of literals and the constraints C were defined by Horn clauses. In a more general setting, both G and C would be permitted to contain constraints drawn from a specialized constraint language not defined by a Horn clause program. Höhfeld and Smolka [5] show how to extend SLD resolution to allow general constraints over arbitrary domains. Their elegant relational approach seems to be straight-forwardly applicable to the Lemma Table proof procedure, and would actually simplify its theoretical description because equality (and unification) would be treated in the constraint system. Unification failure would then be a special case of constraint unsatisfiability, and would be handled by the "optimization" described by Höfeld and Smolka that permits nodes labelled with clauses $G \leftarrow C$ to be deleted if C is unsatisfiable.

References

[1] Chen, W. and D.S. Warren. 1989. *C-Logic of Complex Objects*. ms. Department of Computer Science, State University of New York at Stony Brook.

[2] Chomsky, N. 1986. *Knowledge of Language: Its nature, origins and use*. Praeger. New York.

[3] Earley, J. 1970. "An efficient context-free parsing algorithm", in *Comm. ACM* 13:2, pages 94–102.

[4] Giannesini, F., H. Kanoiui, R. Pasero and M. van Canegham. 1986. *Prolog*. Addison-Wesley. Reading, Massachusetts.

[5] Höhfeld, M. and G. Smolka. 1988. *Definite Relations over Constraint Languages* Lilog report 53, IBM Deutschland.

[6] Johnson, M. 1989. "The Use of Knowledge of Language", in *Journal of Psycholinguistic Research*, 18.1.

[7] Johnson, M. 1990. "Features, frames and quantifier-free formulae", in P. Saint-Dizier and S. Szpakowicz, eds., *Logic and Logic Grammars for Language Processing*, Ellis Horwood, New York, pages 94–107.

[8] Johnson, M. 1991. "Deductive Parsing: The Use of Knowledge of Language", in R.C. Berwick, S.P. Abney and C. Tenny, eds., *Principle-based Parsing: Computational and Psycholinguistics*, Kluwer Academic Publishers, Dordrecht, pages 39–65.

[9] Johnson, M. 1991. "Techniques for deductive parsing", in C.G. Brown and G. Koch, eds., *Natural Language and Logic Programming III*, North Holland, Amsterdam, pages 27–42.

[10] Johnson, M. 1992. "The left-corner program transformation", ms., Brown University.

[11] Lloyd, J. 1984. *Foundations of Logic Programming*. Springer-Verlag, Berlin.

[12] Matsumoto, M., H. Tanaka, H. Hirakawa, H. Miyoshi and H. Yasukawa. 1983. "BUP: a bottom-up parser embedded in Prolog", in *New Generation Computing* 1:2, pages 145–158.

[13] Pereira, F. and S. Shieber. 1987 *Prolog and Natural Language Analysis*. CSLI Lecture Notes Series, Chicago University Press.

[14] Pereira, F. and D.H. Warren. 1983. "Parsing as Deduction", in *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*. MIT, Cambridge, Mass.

[15] Shieber, S. 1985. *Introduction to Unification-based theories of Grammar*. CLSI Lecture Notes Series, Chicago University Press.

[16] Smolka, G. 1992. "Feature Constraint Logics for Unification Grammars", in *The Journal of Logic Programming* 12:1-2, pages 51–87.

[17] Tamaki, H. and T. Sato. 1986. "OLDT resolution with tabulation", in *Proceedings of Third International Conference on Logic Programming*, Springer-Verlag, Berlin, pages 84–98.

[18] Vieille, L. 1987. "Database-complete proof procedures based on SLD resolution", in *Logic Programming: Proceedings of the fourth international conference*, The MIT Press. Cambridge, Massachusetts.

[19] Vieille, L. 1989. "Recursive query processing: the power of logic", Theoretical Computer Science 69, pages 1–53.

[20] Warren, D. S. 1992. "Memoing for logic programs", in *Communications of the ACM* 35:3, pages 94–111.