Computational Intelligence, Volume 10, Number 4, 1994

LOGICAL EMBEDDED PUSH-DOWN AUTOMATA IN TREE-ADJOINING GRAMMAR PARSING

MARK JOHNSON

Cognitive and Linguistic Sciences, Box 1978, Brown University, Providence, RI 02906, U.S.A.

This paper extends the "parsing as deduction" approach to tree-adjoining grammars by showing how a TAG recognition problem can be reduced to a Datalog deduction problem, and presents an SLD selection rule that makes the proof search correspond to a top-down parse using the original grammar. Just as in the DCG extension of context-free grammars, this approach permits nodes to be labeled with first-order terms (rather than only atomic symbols). Finally the paper discusses implementation matters, and describes how the control rule can be efficiently implemented in Prolog.

Key words: natural language, logic programming, program transformations.

1. INTRODUCTION

The connection between context-free grammar (CFG) parsing and Horn-clause deduction is well-known (Pereira and Warren 1983). At the 1990 TAG conference Bernard Lang (1990) showed how this approach could in principle be extended to tree-adjoining grammars (TAGs) by describing a method for translating them into a finite set of Horn clauses. This logical formulation of TAGs extends straightforwardly to the case where nodes are labeled with first-order terms instead of atomic symbols, extending TAGs in the same way that DCGs extend CFGs. Thus it can express feature-based TAGs (Vijay-Shanker and Joshi 1988), albeit using the "position-value" notation of first-order terms rather the standard "attribute-value" notation of unification grammars.

Since Prolog programs are finite sets of Horn clauses, one might also hope to obtain a useful TAG parser by simply running the output of this translation procedure as a Prolog program, in the same way that definite clause grammars (DCGs) are translated into and executed as Prolog programs. But as Lang noticed, when executed by Prolog these programs function as generate-and-test parsers that fail to terminate in most cases. In this paper we diagnose this problem as stemming from Prolog's selection rule. Unlike the DCG case in which Prolog's native left-to-right selection rule results in a top-down recognizer for CFGs, the left-to-right selection rule does not suffice in general for the TAG case; rather, a coroutining selection rule is required to achieve termination. We provide a simple coroutining rule which produces SLD proofs that correspond to top-down parses using TAGs (just as SLD proofs using DCG clauses correspond to top-down parses using CFGs).

It turns out that the coroutining selection rule needed for termination has an interesting structure. If the negative literals awaiting reduction are stored in an embedded push-down automaton (EPDA) (Weir 1988), then the selection rule has a particularly simple form. This should come as no surprise, since EPDAs provide the basic organizational structure needed for top-down TAG parsing.

Interestingly, this approach in which an SLD resolution proof is regarded as an accepting derivation of automata whose tokens are first-order atoms (rather than atomic symbols) was pioneered by Bernard Lang in other work on logical push-down automata (Lang 1988, 1991). To emphasize this connection, we dub the SLD proof procedure with the EPDA-based selection rule a logical embedded push-down automaton (LEPDA).

The final part of the paper focuses on the implementation of the resulting system. An LEPDA interpreter is easily written in Prolog, but the resulting parser is inefficient because of the extra layer of interpretation. However, partial evaluation can be used to remove the

© 1994 Blackwell Publishers, 238 Main Street, Cambridge, MA 02142, USA, and 108 Cowley Road, Oxford, OX4 1JF, UK.

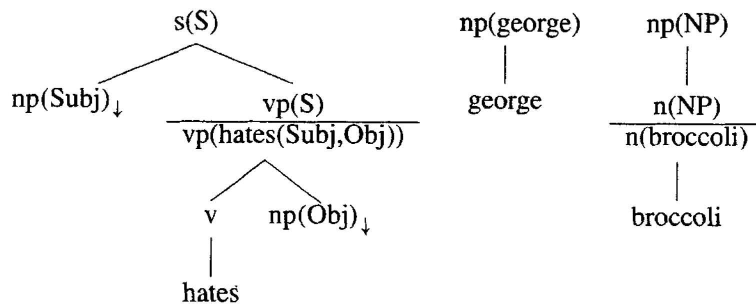

FIGURE 1. The initial trees in the example grammar.

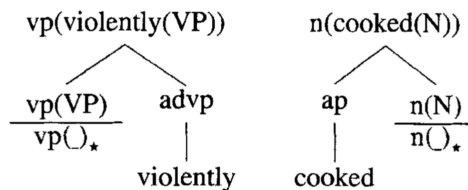

FIGURE 2. The auxiliary trees in the example grammar.

interpretation overhead, and results in a program which can be evaluated using a standard Prolog interpreter.

2. THE HORN AXIOMATIZATION OF TAGS

This section reviews Lang's method for axiomatizing a TAG in Horn clauses (Lang 1990), using a grammar inspired by the one presented by Shieber and Schabes (1990). The initial trees in the example grammar are shown in Fig. 1 and the auxiliary trees are shown in Fig. 2. The subscript down arrow indicates an obligatory substitution point and the subscript star indicates the adjunction foot. Adjunction sites are given *two* labels; they correspond directly to the 'upper' and 'lower' feature components in feature-based TAGs (Vijay-Shanker and Joshi 1988). The procedure which translates TAGs into Horn clauses has two parts. The first part consists of three steps. The first two steps deal with logical variables that appear in node labels, and have no effect in a grammar in which all labels are atomic symbols.

1. Add all the arguments appearing in the foot node of an auxiliary tree as additional arguments of the root node.
2. Introduce additional variables so that all dependencies are local. This affects only internal nodes.
3. Rename uniquely the category label of each interior node in every tree.

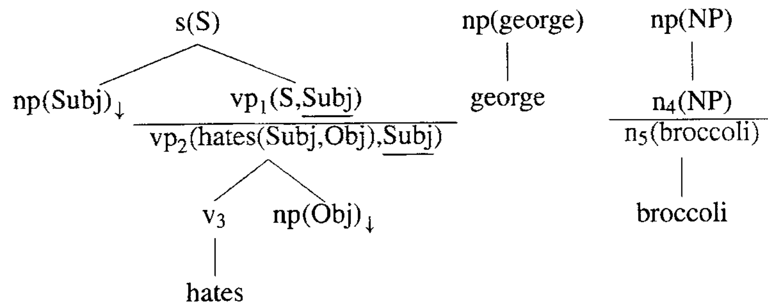

FIGURE 3. The initial trees in the example grammar after assigning unique internal node labels and making all dependencies local.

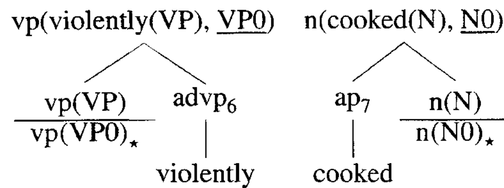

FIGURE 4. The auxiliary trees in the example grammar after assigning unique internal node labels and making all dependencies local.

Step 1 copies the foot node features into the corresponding root node, so that both foot and root node features are accessible in a single place. Steps 2 and 3 localize all of the nonlocal dependencies in the initial and auxiliary trees. Because each local tree (i.e., a node and its immediate descendants) will be encoded as a separate clause, the first part of the translation procedure localizes all nonlocal variable dependencies by adding the shared variables to all intermediate nodes. In addition, the procedure assigns every internal node a unique label. Thus the resulting node labels identify a particular node in a particular tree.

Figures 3 and 4 show renamed versions of the initial and auxiliary trees of Figs. 1 and 2, respectively, where renaming is performed by adding a unique number subscript. In addition, variables added in steps 1 and 2 are shown underlined. The second part of the translation converts each local tree into a Horn clause. For simplicity in specifying the translation, in this section category labels are assumed to be atomic symbols. To deal with non-atomic category labels, additional arguments are added to literals just as in the DCG translation procedure.

Informally, each occurence of a label in an initial tree will be associated with two string positions, just as in a DCG. Because adjunction sites have two labels (the 'upper' and 'lower' components), an adjunction node is associated with four string positions. Figure 5 diagrams these four positions. Suppose an auxiliary tree (shown darkly shaded) is adjoined to a node labeled X in an initial tree (shown lightly shaded), producing the composite structure depicted

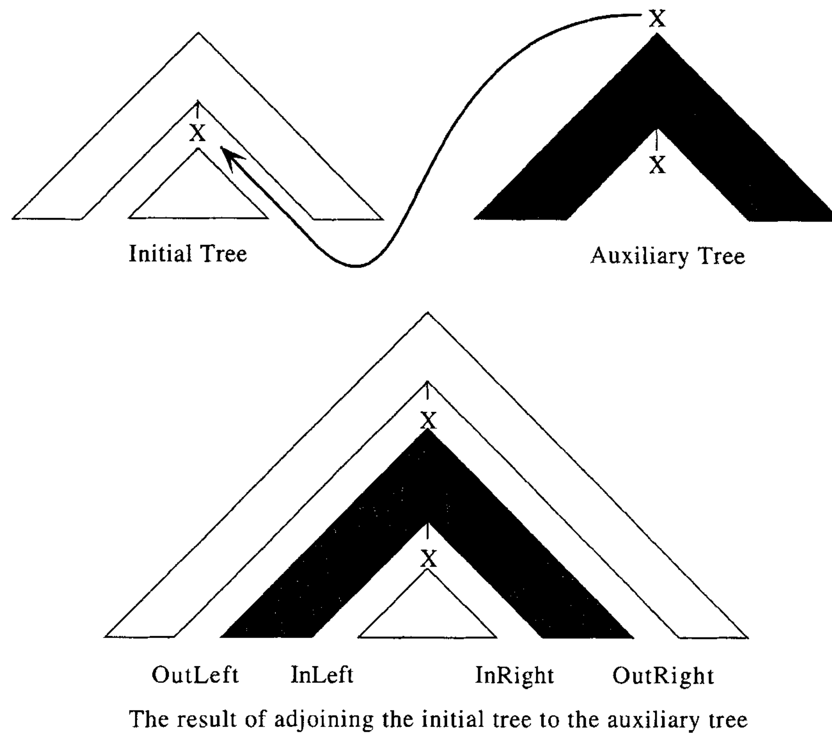

FIGURE 5. The four string positions involved in an adjunction.

at the bottom of the figure. The pair of string positions labeled OutLeft and OutRight are associated with the upper component of the adjunction site X in the initial tree, and the pair of string positions labeled InLeft and InRight are associated are associated with its lower component. The root node of the auxiliary tree is also associated with the four string positions OutLeft, InLeft, InRight and OutRight.

More precisely, a local tree consists of a parent node with label α and its sequence of children nodes with labels $\beta_1, \ldots, \beta_n$. If the parent node is an adjunction site (i.e., has a fractional label) then α is the *lower* component of the label, whereas if the ith child node is an adjunction site then β_i is the *upper* component of the label.

The Horn translation of a local tree depends on whether it occurs on the path from the foot node to the root node in an auxiliary tree, and whether or not it contains a lexical item. There are three cases to consider; the first two are exactly the same as the corresponding cases in the DCG translation procedure. In the following S, S_i and T_i for $0 \leq i \leq n$ are distinct logical variables.

1. A local tree consisting of a parent category α dominating the terminal w is translated as the unit clause[1]

$$\alpha([w|S], S). \qquad (1)$$

[1]Here it is assumed that lexical items are always exhaustively dominated by some node. It is tedious, but possible, to remove this restriction.

For example, the local tree rooted at v_3 in Fig. 3 is translated as the clause

```
v_3([hates|S], S).
```

2. A local tree not lying on a foot to root path of an auxiliary tree is translated as the clause

$$\alpha(S_0, S_n) \leftarrow \beta_1(S_0, S_1), \ldots, \beta_n(S_{n-1}, S_n). \tag{2}$$

For example, the local tree rooted at vp_2 in Fig. 3 is translated (when argument variables are taken into account) as the clause

```
vp_2(S0, S, hates(Subj,Obj), Subj) :-
       v_3(S0, S1),
       np(S1, S, Obj).
```

3. A local tree lying on a foot to root path of an auxiliary tree, where the mth child node dominates the foot node, is translated as the clause

$$\begin{aligned} \alpha(S_0, S_m, T_m, T_n) \leftarrow & \\ & \beta_1(S_0, S_1), \ldots, \beta_{m-1}(S_{m-2}, S_{m-1}), \\ & \beta_m(S_{m-1}, S_m, T_m, T_{m+1}), \\ & \beta_{m+1}(T_{m+1}, T_{m+2}), \ldots, \beta_n(T_{n-1}, T_n). \end{aligned} \tag{3}$$

For example, the local tree rooted at the root node of the left-most auxiliary tree of Fig. 4 is translated as the following clause.

```
vp(S0, S, T0, T, violently(VP), VP0) :-
       vp(S0, S, T0, T1, VP, VP0),
       advp_7(T1, T).
```

Now we turn to internal adjunction sites. Each internal adjunction site is translated as a Horn clause. There are two cases to be considered here.

4. An internal adjunction site not lying on a foot to root path of an auxiliary tree with a numerator labeled α and a denominator labeled β is translated as the clause

$$\alpha(S_0, T_1) \leftarrow \gamma(S_0, S_1, T_0, T_1), \beta(S_1, T_0) \tag{4}$$

where γ is the "original" label of the node before the renaming step (3) above. For example, the adjunction site vp_1/vp_2 is translated as the following clause.

```
vp_1(S0, T, SS, Subj) :-
       vp(S0, S, T0, T, SS, hates(Subj,Obj)),
       vp_2(S, T0, hates(Subj,Obj), Subj).
```

5. An internal adjunction site lying on a foot to root path of an auxiliary tree with labels as above is translated as the clause

$$\alpha(S_0, S_2, T_0, T_2) \leftarrow \gamma(S_0, S_1, T_1, T_2), \beta(S_1, S_2, T_0, T_1). \tag{5}$$

Finally, we require a clause for each (unrenamed) category that labels an adjunction site which expresses that adjunction is optional. (If these clauses are omitted, then adjunction at adjunction sites will be obligatory).

```
:- op(700, xfx, <-).

s(S0, S, SS) <-
       [np(S0, S1, Subj)]-[vp_1(S1, S, SS, Subj)]-[].

vp_1(S0, T, SS, Subj) <- []-[
       vp(S0, S, T0, T, SS, hates(Subj,Obj)),
       vp_2(S, T0, hates(Subj,Obj), Subj)]-[].

vp_2(S0, S, hates(Subj,Obj), Subj) <-
       [v_3(S0, S1)]-[np(S1, S, Obj)]-[].

v_3([hates|S0], S0) <- []-[]-[].

vp(S, S, T, T, VP, VP) <- []-[]-[].
vp(S0, S, T0, T, violently(VP), VP0) <-
       []-[vp(S0, S, T0, T1, VP, VP0)]-[advp_7(T1, T)].

advp_7([violently|S], S) <- []-[]-[].

np([george|S], S, george) <- []-[]-[].

np(S0, S1, NP) <- []-[n_4(S0, S1, NP)]-[].

n_4(S0, T1, N) <-
       []-[n(S0, S1, T0, T1, N, N0), n_5(S1, T0, N0)]-[].

n_5([broccoli|S], S, broccoli) <- []-[]-[].

n(S, S, T, T, N, N) <- []-[]-[].
n(S0, S, T0, T, cooked(N0), N) <-
       [ap_6(S0, S1)]-[n(S1, S, T0, T, N0, N)]-[].

ap_6([cooked|S], S) <- []-[]-[].
```

FIGURE 6. Horn clause grammar axioms.

6. For each distinct γ that appeared in steps (4) and (5) above, add a clause of the form

$$\gamma(S, S, T, T). \tag{6}$$

   For example, the translation contains the following clause, which expresses the fact that adjunction at vp is optional.

```
vp(S, S, T, T, VP, VP).
```

The Horn clauses that result in this translation are shown in Fig. 6, albeit in a slightly strange notation. The left arrow <- should be read as a right-to-left implication sign, and the negative literals of the clause are the result of appending the atoms in all three lists to its right (the reason for separating the negative literals into three groups is explained below).

The Horn clauses produced by the translation procedure correctly specify the strings generated by the grammar, but the Prolog program functions as a 'generate and test' parser which in general does not terminate on all input strings.

The problem arises because Prolog's native left-to-right selection rule is not flexible enough to ensure that (goals corresponding to) terminals are resolved in the order they appear in the input to be parsed. Consider again the tree depicted in Fig. 5 that results from adjunction. The clause corresponding to (the root node of) the auxiliary tree directly or indirectly introduces subgoals corresponding to the terminals between the pair of string positions OutLeft and InLeft, as well between the pair of string positions InRight and OutRight.

Now, the clause corresponding to the initial tree directly or indirectly introduces subgoals that correspond to the terminals between InLeft and InRight. How should these subgoals be ordered in the clause with respect to the goal that introduces the auxiliary tree, which is also contained in the clause corresponding to the initial tree? It turns out that no goal ordering is satisfactory. Because subgoals corresponding to terminals in the auxiliary tree are responsible for instantiating the string position variable InLeft, none of these subgoals can be ordered before the goal that introduces auxiliary tree, otherwise InLeft would be uninstantiated. Because every terminal can be 'recognized' at an uninstantiated left string position, such a goal ordering results in a 'generate-and-test' parsing strategy.

The other option—that the subgoals corresponding to the terminals between InLeft and InRight are ordered after the goal that introduces the auxiliary tree—also results in a 'generate-and-test' behavior, because the goals corresponding to the terminals between InRight and OutRight in the auxiliary tree would be recognized before the terminals in the initial tree that span InLeft to InRight.

It seems as if what is needed is an ability to *insert* subgoals from the clause corresponding to the initial tree *between* the subgoals from the auxiliary tree. Prolog's native left-to-right selection rule provides no way to do this, but the EPDA-based selection rule described below can do this. It is no surprise that an EPDA-based selection rule suffices to control a proof that corresponds to a top-down TAG parse, since EPDAs are the basic automata model of TAG parsing.[2]

3. THREE IMPLEMENTATIONS

This section describes three successively refined implementations of a top-down TAG parser that uses SLD resolution on clauses produced by the translation just described. All of the implementations reduce goals in the same order (which corresponds exactly to the order in which the corresponding nodes would be enumerated in a top-down parse).

1. The first implementation uses a selection rule which only selects goals whose first (i.e., leftmost) string arguments are instantiated (all other goals are delayed). It is easy to implement this strategy using a metainterpreter, and it can be tolerably efficiently implemented in extended Prolog implementations which incorporate coroutining control extensions such as `freeze` or `wait`.
2. The second implementation uses a metainterpreter which stores unresolved goals in an embedded push-down stack. This metainterpreter, called a logical embedded push-down automaton (LEPDA), uses the original Horn clause axiomatization annotated with control information (as in Fig. 6) which tells it how to manipulate the embedded push-down

[2] Similiarly, Prolog's native PDA-based selection rule enables a resolution proof to correspond directly to a top-down CFG parse.

stacks. Even though it resolves goals in the same order as implementation 1, it does not require coroutining primitives.

3. Finally, a Prolog program can be obtained by partially evaluating the metainterpreter just mentioned with the grammar axioms and performing additional program transformations. The resulting program passes a stack of additional goals from goal to goal in 'continuation-passing' style.[3] It is probably the most efficient of all of the implementations, and does not require coroutining primitives.

The crucial observation that the second and third control strategies depend on is that the order in which literals should be selected can be enumerated by an embedded push-down automaton, so the inference procedure can be implemented by a LEPDA.

The state information of the LEPDA metainterpreter of implementation 2 consists of an EPDS that holds the goals that still remain to be reduced. A LEPDA program is set of Horn clauses in which the negative literals have been partitioned into three sequences, so each Horn clause is of the form $\alpha \leftarrow \beta_1 - \beta_2 - \beta_3$, where each β_i is a sequence of atoms.[4]

There are two basic operations in a LEPDA that are continued until the EPDS is empty.

1. If the top stack of the EPDS is empty, pop it.
2. Remove the top goal α from the top stack of the EPDS, and nondeterministically attempt to unify it with the head α' of each clause $\alpha' \leftarrow \beta_1 - \beta_2 - \beta_3$ in the program. If this succeeds, then push β_2 in reverse order onto the top stack, insert β_3 as a "new" stack immediately below the top stack, and finally push β_1 as a new stack on top of the top stack.

An LEPDA implements an SLD resolution proof procedure; the goal sequences in the clauses determine only the order in which goals are reduced.

Figure 6 is in fact a LEPDA program that recognizes the grammar depicted in Figs. 1 and 2. The control information, which determines how the negative literals are assigned to stacks when the clause's head is reduced, is generated as follows. (The "type" of a clause refers to the translation rule used to produce it).

1. In clauses of type (2) each negative literal is assigned to its own stack. (Since the heads of these clauses can only appear at the bottom of an EPDS stack, it does not matter which stacks these are).
2. In clauses of type (3) each negative literal is assigned to its own stack, and in addition the literal corresponding to the mth child (the one on the path from the foot to the root) must be in the current stack (i.e., it must be pushed onto the stack the head literal was popped from).
3. In clauses of type (4) and (5) both negative literals are assigned to the current stack (i.e., they are both pushed on to the stack that was just popped).

Figure 7 contains a Prolog interpreter for LEPDA programs. Note that the "outer stack" of the EPDS is implemented using Prolog's own stack of pending goals.

While conceptually elegant, the metainterpreter approach of implementation 2 is quite inefficient. The metainterpreter overhead can be eliminated by partial evaluation and other program transformation techniques. Specifically, the LEPDA interpreter can be partially

[3]Martin Kay has independently developed a Prolog encoding of TAGs very similar to this one.

[4]The language of LEPDA programs can be extended to allow the creation of an arbitrary number of new stacks per reduction, at the cost of a minor increase in complexity of the metainterpreter described below, but the language presented here suffices for a grammar with at most binary branching.

```
prove(Goal) :-
        lepda([Goal]).

lepda([]).
lepda([Goal|Goals0]) :-
        Goal <- Before-Goals1-After,
        append(Goals1, Goals0, Goals),
        lepda(Before),
        lepda(Goals),
        lepda(After).
```

FIGURE 7. A Prolog interpreter for LEPDA programs.

evaluated with respect to LEPDA program clauses and the append clauses in the usual manner (see, e.g., Pereira and Shieber 1987 for details). In addition to partial evaluation, the Prolog program for implementation 3, in Fig. 8, is obtained by

- Partially evaluating any deterministic call to lepda,
- Promoting category labels to predicate names (i.e., systematically replacing every atom of the form lepda([$P(T_1, \ldots, T_n)$ | L]) with the atom $P(T_1, \ldots, T_n, L)$), and
- Promoting the category labels over the list of pending goals manipulated by lepda.[5]

When we do this, we see that the list of goals is playing the role of a *continuation* in the manner described by Sato and Tamaki (1989); suggesting the renaming of the lepda relation to cont and using true to name the "empty list" of pending goals. In the program in Fig. 8 the last argument holds the continuation of goals, and the predicate cont calls the appropriate goal with the appropriate arguments. Roughly speaking, adjunction pushes a goal on to the current continuation (cf., the clauses for vp_1 and np_4), which is activated when the left-hand side of that adjunction has been processed.

Note that further optimization of this program is possible. For example, since the heads of clauses of types (1) and (2) can only appear at the bottom of some stack in the EPDS, the continuation associated with these heads will always be true; these clauses can be specialized for this case and the continuation argument omitted.

4. CONCLUSION

This paper has shown that there is a systematic, automatizable method of proceeding from Lang's Horn clause axiomatization of a TAG to a Prolog program that functions as a top-down parser of that TAG. Only a comparatively small residue of the LEPDA interpreter (the cont predicate) remains.

But like all top-down parsers, these parsers will fail to terminate on certain inputs when the grammar is left-recursive. The VP adjunction rule in the grammar in Fig. 2 is an example of such a left-recursive construction.

In the domain of CFG parsing, alternative control strategies, such as left-corner and shift-reduce strategies, terminate with left-recursive grammars, and the left-corner algorithm can be naturally implemented as a Horn program transformation. It would be interesting to inves-

[5]For example, by promoting function symbols the list [a,b,c] becomes the term a(b(c([]))).

```
s(S0, S, SS, C) :-
       np(S0, S1, Subj, true), vp_1(S1, S, SS, Subj, C).

vp_1(S0, T, SS, Subj, C) :-
       vp(S0, S, T0, T, SS, hates(Subj,Obj),
             vp_2(S, T0, hates(Subj,Obj), Subj, C)).

vp_2(S0, S, hates(Subj,Obj), Subj, C) :-
       v_3(S0, S1, true), np(S1, S, Obj, C).

v_3([hates|S0], S0, C) :- cont(C).

vp(S, S, T, T, VP, VP, C) :- cont(C).
vp(S0, S, T0, T, violently(VP), VP0, C) :-
       vp(S0, S, T0, T1, VP, VP0, C), advp_7(T1, T, true).

advp_7([violently|S], S, C) :- cont(C).

np([george|S], S, george, C) :- cont(C).

np(S0, S1, NP, C) :- n_4(S0, S1, NP, C).

n_4(S0, T1, N, C) :-
       n(S0, S1, T0, T1, N, N0, n_5(S1, T0, N0, C)).

n_5([broccoli|S], S, broccoli, C) :- cont(C).

n(S, S, T, T, N, N, C) :- cont(C).
n(S0, S, T0, T, cooked(N0), N, C) :-
       ap_6(S0, S1, true), n(S1, S, T0, T, N0, N, C).

ap_6([cooked|S], S, C) :- cont(C).

cont(true).
cont(vp_2(A1, A2, A3, A4, A5)) :- vp_2(A1, A2, A3, A4, A5).
cont(n_5(A1, A2, A3, A4)) :- n_5(A1, A2, A3, A4).
```

FIGURE 8. The result of partial evaluation and transformation.

tigate how these techniques can be formalized as control rules for appropriate axiomatizations of TAGs.

REFERENCES

LANG, B. 1988. Complete evaluation of Horn clauses: an automata-theoretic approach. Research Report 913, INRIA.

LANG, B. 1990. Horn axiomatization of tree-adjoining grammars. Presentation at the TAG conference, Schloss Dagstuhl.

LANG, B. 1991. Towards a uniform formal framework for parsing. *In* Current issues in parsing technology. *Edited by* M. Tomita. Kluwer Academic. Boston, MA.

PEREIRA, F., and S. SHIEBER. 1987. Prolog and natural language analysis. C.S.L.I. Lecture Notes Series 10. Chicago University Press, Chicago.

PEREIRA, F., and D. H. D. WARREN. 1983. Parsing as deduction. *In* The Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics, Massachusetts Institute of Technology, pp. 137–144.

SATO, T., and H. TAMAKI. 1989. Existential continuation. New Generation Computing, **6**(4):421–438.

SHIEBER, S. M., and Y. SCHABES. 1990. Synchronous tree-adjoining grammars. *In* The Proceedings of the 13th International Conference on Computational Linguistics (COLING '90), Helsinki, Finland.

VIJAY-SHANKER, K., and A. K. JOSHI. 1988. Feature based tree adjoining grammars. *In* Proceedings of the 12th International Conference on Computational Linguistics, Budapest, Hungary.

WEIR, D. J. 1988. Characterizing mildly context-sensitive grammar formalisms. Ph.D. thesis, University of Pennsylvania.