An Implementation of a Computational Model for the Analysis of Arguments An Introduction to the First Attempt

Trevor J. Smedley

Research Report CS-86-26
July 1986

An Implementation of a Computational Model for the Analysis of Arguments
An Introduction to the First Attempt

Trevor J. Smedley
University of Waterloo

Introduction

The following is a description of a first attempt at a Prolog implementation of a computational model for the analysis of arguments. The implementation of the model can be found in /u/tjsmedley on watdragon. For a detailed description of the algorithms, and other theoretical details see the papers and thesis by R. Cohen[1, 2, 3]. The three different algorithms have been implemented; pre-order, post-order and hybrid. The code particular to each algorithm can be found under the directories; super_pre, super_post and super_hybrid. The directory front_end contains code which is common to all three, and there is a symbolic link to this directory in each of the three directories mentioned above.

Directory *front_end*

The predicates in this directory handle all the common stuff, and are pretty self explanatory. It is assumed that the argument tree is represented by a collection of *father* (*father ,son*) relations, and a single *root* (*root*) relation. Recall that, for an argument, a proposition is recorded as *son* if it is found to be evidence for the *father* (claim). Thus, after the analysis, these relations will reflect the tree structure of the argument, with the overall claim as the root, and each evidence propostion a son to its father claim. These relations will be put into the database by the algorithm for building the argument tree. The evidence oracle, which is described in the next section, is contained in this directory.

2.1. The Evidence Oracle

The evidence oracle decides whether one sentence in an argument is evidence for another. It is probably the section of the implementation which is most open to modifications and improvements.

In this first implementation, the evidence oracle was made as simple as possible. It simply asks the user if one sentence is evidence for another, and makes no use of any knowledge base. It succeedes if the user replies 'y', and fails if the user answers Which questions it asks is determined by the algorithm being use for the analysis. This is normally a small subset of all possible questions. It is not entirely stupid in that it will not ask the same question twice. It stores the data from previous questions in the form of *asserted_evidence* (*claim evidence*) and *unasserted_evidence* (*claim non –evidence*) clauses. It will always check if the question has already been answered before asking it. One remaining possible modification which would be easy to implement would be the following. If it already knows that sentence s1 is evidence for sentence s2 then it could decide that s2 is not evidence for s1. However, since in the current model this question is never asked, this modification would not be of any use. Any other modification would most likely involve a considerable amount of work.

2.2. Other Predicates

The other predicates in this directory are fairly self-explanatory. There are a few predicates which do not appear in the *front_end* directory but which will be mentioned here as they appear in each of the other directories, and although the implementations are not the same for each algorithm, the functions are.

2.2.1. Predicate *special*

This predicate handles any special processing which is to be done when an answer other than 'y' or 'n' is given to the evidence oracle. Examples are; for the post-order algorithm the stack can be printed out by replying with 's', and in the hybrid analysis the tree which has been created so far will be displayed if 't' is entered.

2.2.2. Predicate *get_children*

This predicate will get a list of the children of a node ordered from left to right. This is used when printing out the argument tree.

Directory *super_pre /pre_order*

In this directory are the predicates which do the analysis for pre-order type arguments. The only predicate of any importance, *pre_order_* , is quite easy to understand when you see the algorithm which it is based upon.

3.1. Predicate *pre_order_*

This predicate has two arguments. The first is a list of sentences yet to be processed, and the second is what R. Cohen refers to as L , for "last", in her description of the algorithm.

The predicate will initially set L to be the first sentence in the list, and make it the root of the argument by placing a *root* relation in the database.

For each sentence in the list it checks with the evidence oracle whether this sentence is evidence for L . If it is, this information is placed in the database with a *father* relation, and the predicate calls itself with the sentence which was the head of the list as L , and the list with the first sentence removed as the new sentence list. If the first sentence is not evidence for L , then the predicate is called again with the same list of sentences, and the father of L as the new L . It will finish when there are no new sentences to be processed.

Directory *super_post /post_order*

This directory contains the predicates for analysing post-order type arguments. The two predicates of concern are *post_order_* and *remove_sons* .

4.1. Predicate *post_order_*

This predicate preforms the algorithm for the post-order analysis. It has two arguments. The first is the list of sentences not yet processed, and the second is the stack of subtrees for which fathers have not yet been found.

The predicate will initially push the first sentence onto the stack, and will finish when there are no sentences left to be processed and there is only one sentence left on the stack. This one remaining sentence will be made the root of the argument tree.

The algorithm takes the first sentence from the list and checks if it is evidence for the sentence on the top of the stack (with the evidence oracle). If it is not it simply places the sentence on the stack, and calls itself with the new stack, and the list of sentences with the first one removed. If it is evidence, that fact will be placed in the database in the form of a *father* relation. Then any other members of the stack which are also evidence for the sentence are removed from the stack by calling the predicate *remove_sons*

(see next section). The predicate will then call itself as above.

4.2. Predicate *remove_sons*

This predicate takes three arguments; *Father* , *Old_stack* , and *New_stack* . It will pop all the elements of *Old_stack* which are evidence for *Father* , place the information in the database in terms of *father* relations, and succeed when a sentence which is not evidence for *Father* is on the top of the stack. *New_stack* will be *Old_stack* with the above elements removed.

Directory *super_hybrid / hybrid*

The predicates in this directory are for hybrid analysis of arguments. This is considerably more complicated than the previous algorithms, and also involves keeping track of some additional data.

5.1. Additional Data Relations

In the hybrid algorithm, the rightmost child of a node is of importance, and thus the order in which sons are added must be kept track of. This is done with relations of the type *rightmost_child* (*Father Son*) and *left_brother* (*Left Right*) where the rightmost child of *Father* is *Son* , and *Left* and *Right* are children of the same father with *Left* to the left of *Right* . This information is used when printing out the tree so that the sons of a node are printed out in the correct order. It is also used in the predicates *attach_sons* and *no_sons_evidence* .

5.2. Predicate *hybrid_*

This is the predicate which actually performs the algorithm for the analysis of an argument. It has three arguments. The first is the list of sentences yet to be processed, the second is what R. Cohen refers to as *New* in her papers, and the third is her *L* for "last". When this predicate is first called its first argument is a list of sentences without the first sentence of the argument, the second argument is the first sentence of the argument, and the third is the value *dummy* . Before the predicate is called, an assertion is placed in the database indicating that anything is evidence for *dummy* . This is the dummy node at the top of the tree referred to by R. Cohen. Once the predicate *hybrid_* returns, the top level predicate *build_tree* checks that dummy has only one son, and then this son is asserted to be the root of the argument tree.

The last clause in the predicate handles the case when *New* is not evidence for *L* . In this case the predicate *hybrid_* is executed with the same list of sentences, the same *New* and the father of *L* as the new *L* . In all the other clauses, it is first checked that *New* is evidence for *L* .

The first two clauses handle the processing when there are no more sentences besides *New* and *L* . The first is executed when the rightmost child of *L* is not evidence for *N* , and in this case *L* is asserted to be the father of *New* . The predicate then succeeds. The second is for when some sons of *L* are evidence for *New* . In this case the sons of *L* which are evidence are asserted to be sons of *New* using the predicate *attach_sons* , and the predicate returns.

The next two clauses are similar to the first two, except they handle the cases when there are sentences left to be processed. In the third clause, after the same processing as in the first one is done, the predicate *hybrid_* is called with the list of sentences being the original list with the first sentence removed, the new *New* being the first sentence on the original list, and the new *L* being the old *New* . For the fourth clause, the processing is the same as the second one, and then *hybrid_* is called again with the arguments as above, except the new *L* is the same as the original *L* .

Referring to R. Cohen's thesis we see that the first and third clauses correspond to step *B* 1 of the algorithm; the second and fifth clauses refer to step *B* 2, and the last clause is step *B* 3.

5.3. Predicate *no_sons_evidence*

This is the predicate called by clauses one and three of *hybrid_* . It checks if any sons of a node are evidence for some sentence. Note that it checks only if the rightmost child of a node is evidence, as indicated in the comments to the algorithm by R. Cohen.

5.4. Predicate *attach_sons*

This is the predicate called by clauses two and four of the predicate *hybrid_* . It has two arguments, *Father* and *Son* . It will go through the sons of *Father* from right to left, attaching all those which are evidence for *Son* below *Son* until one is encountered which is not evidence. When attaching a son of *Father* below *Son* we must make sure that the left-right ordering of both the sons of *Father* , and the sons of *Son* is preserved. In order to ensure this two predicates are used; *assert_father_on_left* , which adds a son to *Son* which is to the left of all the other sons, and *remove_rightmost* , which takes the rightmost son of *Father* and makes it no longer a son, updating the *rightmost_child* relation to indicate the new rightmost son of *Father* . For a more detailed description of these predicates, see the comments in the code.

5.5. Other Predicates

The other predicates in this directory deal mainly with the handling of the information in the database. See the code for information on these predicates.

Code from the *Front_end* Directory

```
%
%    This is the user level predicate of the agrgument analysis
%    package. The argument is passed as a list of sentences, each
%    sentence being a list of words. The predicate will first clear
%    the auxillary database and insert three assertions in order
%    that "unknown predicate" errors do not occur, then print out
%    the argument, and call the predicate for buliding the argument
%    tree. Which method is used (pre_order, post_order or hybrid)
%    depends upon which module is used. The predicate build_tree
%    will place assertions in the database. These will be of the
%    form root(X) or father(X Y) where these assertions define the
%    argument tree. Finally, the argument tree is printed out.
%
analyse(Argument) <-
     clear_aux(usr)
     assert(usr asserted_evidence([] []) [])
     assert(usr unasserted_evidence([] []) [])
     assert(usr father([] []) [])
     print_argument(Argument)
     build_tree(Argument)
     printf("Argument tree;" [])
     nl
     root(X)
     write(X)
     nl
     print_tree(X,1);
%
%    This is the preliminary evidence oracle. The first clause
%    checks that the information is not already in the database.
%    The second clause will first make sure that the information
%    is not already in the database. Then it asks the user whether
%    the first sentence is evidence for the second, does any special
%    processing which may be indicated by the input character,
%    places the information in the database (with the predicate
%    assert_evidence) and then fails if the input character is not
%    'y'.
%
evidence_oracle(E A _) <-
     asserted_evidence(E A);
evidence_oracle(E A SpecialArg) <-
     not(asserted_evidence(E A))
     not(unasserted_evidence(E A))
     printf("Is 0 [])
     write(E)
     nl
     printf("evidence for 0 [])
     write(A)
     nl
     printf("(answer y or n)  " [])
     get_first_char(X)
```

```
    nl
    special(X OutX SpecialArg)
    good_answer(OutX GoodX)
    assert_evidence(GoodX E A)
    !
    eq(GoodX 'y');
%
%   This predicate puts the information from the evidence oracle
%   into the database. This prevents redundant questions from
%   being asked. It also stores the information that one sentence
%   is *not* evidence for another.
%
assert_evidence('y' E A) <-
    assert(usr asserted_evidence(E A) []);
assert_evidence('n' E A) <-
    assert(usr unasserted_evidence(E A) []);
%
%   This predicate will print out the input agrument using the
%   predicate print_argument_
%
print_argument(X)
    nl
    printf("The argument is;0 [])
    print_argument_(X)
    nl;
%
%   This predicate will print out the sentences in the argument
%
print_argument_([]);
print_argument_([H | T]) <-
    write(H)
    nl
    print_argument_(T);
%
%   This predicate will print out the completed argument tree. It
%   assumes that the root (Root) has beed printed, finds all the
%   children of the root, and then calls the predicate print_children
%   which prints out the chidren, and the subtrees beneath them.
%   The predicate get_children is specific to the routines for the
%   type of analysis being done in order to make sure that the
%   list is returned in the correct order.
%
print_tree(Root Depth) <-
    get_children(Root Children)
    print_children(Children Depth);
%
%   This predicate prints one child of a list, then prints the tree
%   below that child (at one more level of indentation), and then
%   prints the remaining children in the same way.
%
print_children([] _);
print_children([ H | T ] Depth) <-
    tab(Depth)
    write(H)
```

```
    nl
    plus(Depth 1 New_Depth)
    print_tree(H New_Depth)
    print_children(T Depth);
%
%   Repeatedly prompts until either a 'y' or a 'n' is input
%
good_answer('y' 'y');
good_answer('n' 'n');
good_answer(X GoodX) <-
    ne(X 'y')
    ne(X 'n')
    printf("(answer y or n please) " [])
    get_first_char(NewX)
    good_answer(NewX GoodX);
%
%   This reads in all remaining characters on a line and discards them
%
end_of_line('0);
end_of_line(_) <-
    get0(X)
    end_of_line(X);
%
%   This returns the first character which is input on a line
%
get_first_char(X) <-
    get0(X)
    end_of_line(X);
```

Code for the Pre-order Algorithm

```
%
%    This calls the predicate which builds the argument tree
%
build_tree(Argument) <-
     pre_order_(Argument nil);
%
%  This predicate will get the list of children of Root in the
%  order in which they are to be printed out
%
get_children(Root Children)
     all_of(Children X father(Root X));
%
%    This will build a pre-order argument tree. The algorithm is by
%    R. Cohen. See the documentation for a detailed description.
%
pre_order_([] _);
pre_order_([H | T] nil) <-
     assert(usr root(H) [])
     pre_order_(T H);
pre_order_([H | T] L) <-
     ne(L nil)
     evidence_oracle(H L _)
     assert(usr father(L H) [])
     pre_order_(T H);
pre_order_([H | T] L) <-
     ne(L nil)
     not(evidence_oracle(H L _))
     father(New_L L)
     pre_order_([H | T] New_L);
%
%    This will perform any special functions required. There are
%    none at this time.
%
special(X X _);
```

Examples of the Pre-order Algorithm

These two examples will be used to describe how to use the system. Things printed by the system are in normal type, and things entered by the user are in bold.

To start the system, call wup with the name of the directory for the type of analysis you want to do as the argument. In this case the directory name was *super_pre* . The system will respond with a question mark. When you get this prompt you must call the predicate *analyse* . This predicate takes one argument which should be a list whose entries are lists of words. Each list of words corresponds to a sentence in the argument. Be sure to end with a semicolon, as shown in the examples. The program will then ask questions about the evidence relations between the sentences. Answer each question with *y* or *n* . When the program has determined the argument structure it will be printed out. The printout corresponds to the argument tree with each level of indentation being a level in the tree. Refer to the examples to see how this works.

Note that when doing post-order analysis it is possible to view the stack by answering *s* when asked a question by the evidence oracle, and in hybrid you can view the tree constructed so far by responding with a *t* . See the examples in the appropriate sections.

Example 1

```
[1]% wup super_pre
Waterloo Unix Prolog
?    analyse([[jones,is,a,good,president],
[he,has,lots,of,experience],
[he,was,on,the,board,ten,years],
[and,hes,honest],
[he,refused,bribes,many,times]]);

The argument is;
[jones,is,a,good,president]
[he,has,lots,of,experience]
[he,was,on,the,board,ten,years]
[and,hes,honest]
[he,refused,bribes,many,times]

Is
[he,has,lots,of,experience]
evidence for
[jones,is,a,good,president]
(answer y or n)  y

Is
[he,was,on,the,board,ten,years]
evidence for
[he,has,lots,of,experience]
(answer y or n)  y

Is
[and,hes,honest]
evidence for
[he,was,on,the,board,ten,years]
(answer y or n)  n

Is
[and,hes,honest]
evidence for
[he,has,lots,of,experience]
(answer y or n)  n

Is
[and,hes,honest]
evidence for
[jones,is,a,good,president]
(answer y or n)  y

Is
[he,refused,bribes,many,times]
evidence for
[and,hes,honest]
(answer y or n)  y

Argument tree;
[jones,is,a,good,president]
 [he,has,lots,of,experience]
  [he,was,on,the,board,ten,years]
 [and,hes,honest]
  [he,refused,bribes,many,times]
yes
?quit;
```

Example 2

```
[1]% wup super_pre
Waterloo Unix Prolog
?    analyse([[the,city,is,a,mess],
[the,parks,are,a,mess],
[the,playground,area,is,run,down],
[the,sandboxes,are,dirty],
[the,swings,are,broken],
[the,highways,needs,revamping]]);

The argument is;
[the,city,is,a,mess]
[the,parks,are,a,mess]
[the,playground,area,is,run,down]
[the,sandboxes,are,dirty]
[the,swings,are,broken]
[the,highways,needs,revamping]

Is
[the,parks,are,a,mess]
evidence for
[the,city,is,a,mess]
(answer y or n)  y

Is
[the,playground,area,is,run,down]
evidence for
[the,parks,are,a,mess]
(answer y or n)  y

Is
[the,sandboxes,are,dirty]
evidence for
[the,playground,area,is,run,down]
(answer y or n)  y

Is
[the,swings,are,broken]
evidence for
[the,sandboxes,are,dirty]
(answer y or n)  n

Is
[the,swings,are,broken]
evidence for
[the,playground,area,is,run,down]
(answer y or n)  y

Is
[the,highways,needs,revamping]
evidence for
[the,swings,are,broken]
(answer y or n)  n

% Note that only the last son is eligible as evidence

Is
[the,highways,needs,revamping]
evidence for
[the,playground,area,is,run,down]
(answer y or n)  n

Is
[the,highways,needs,revamping]
evidence for
[the,parks,are,a,mess]
(answer y or n)  n

Is
[the,highways,needs,revamping]
evidence for
[the,city,is,a,mess]
(answer y or n)  y

Argument tree;
[the,city,is,a,mess]
 [the,parks,are,a,mess]
  [the,playground,area,is,run,down]
   [the,sandboxes,are,dirty]
   [the,swings,are,broken]
 [the,highways,needs,revamping]
yes
?quit;
```

Code for the Post-order Algorithm

```
%
%    Calls the predicate for building the argument tree
%
build_tree(Argument) <-
     post_order_(Argument []);
%
%  This predicate will get the list of children of Root in the
%  order in which they are to be printed out
%
get_children(Root Children)
     all_of(Children X father(Root X));
%
%    This builds a post order argument tree. The algorithm is by
%    R. Cohen. See the documentation for a detailed explanation.
%
post_order_([H | T] []) <-
     post_order_(T [H]);
post_order_([] [X]) <-
     assert(usr root(X) []);
post_order_([H | T] [Top | Stack]) <-
     evidence_oracle(Top H [Top | Stack])
     assert(usr father(H Top) [])
     remove_sons(H Stack New_Stack)
     post_order_(T [H | New_Stack]);
post_order_([H | T] [Top | Stack]) <-
     not(evidence_oracle(Top H _))
     post_order_(T [H Top | Stack]);
%
%    This will print out the stack used when constructing the
%    argument tree
%
print_stack([]);
print_stack([H | T]) <-
     write(H)
     nl
     print_tree(H 1)
     print_stack(T);
%
%    This recursively removes the sons from the stack. See the
%    description of the algorithm in the documentation.
%
remove_sons(_ [] []);
remove_sons(Father [H | T] New_stack) <-
     evidence_oracle(H Father [H | T])
     assert(usr father(Father H) [])
     remove_sons(Father T New_stack);
remove_sons(Father [H | T] [H | T]) <-
     not(evidence_oracle(H Father [H | T]));
%
%    This will perform any special functions required. The only one
```

```
%    so far is outputting the stack.
%
special('s' X Stack) <-
    printf("Stack is;0 [])
    nl
    print_stack(Stack)
    nl
    printf("(answer y or n)  " [])
    get_first_char(X);
special(X X _);
```

Examples of the Post-order Algorithm

Example 1

```
[1]% wup super_post
Waterloo Unix Prolog
?    analyse([[the,benches,are,broken],
[the,trails,are,choppy],
[the,trees,are,dying],
[in,sum,the,parks,are,a,mess]]);

The argument is;
[the,benches,are,broken]
[the,trails,are,choppy]
[the,trees,are,dying]
[in,sum,the,parks,are,a,mess]

Is
[the,benches,are,broken]
evidence for
[the,trails,are,choppy]
(answer y or n)  n

Is
[the,trails,are,choppy]
evidence for
[the,trees,are,dying]
(answer y or n)  n

Is
[the,trees,are,dying]
evidence for
[in,sum,the,parks,are,a,mess]
(answer y or n)  y

Is
[the,trails,are,choppy]
evidence for
[in,sum,the,parks,are,a,mess]
(answer y or n)  y

Is
[the,benches,are,broken]
evidence for
[in,sum,the,parks,are,a,mess]
(answer y or n)  y

Argument tree;
[in,sum,the,parks,are,a,mess]
 [the,trees,are,dying]
 [the,trails,are,choppy]
 [the,benches,are,broken]
yes
?quit;
```

Example 2

```
[1]% wup super_post
Waterloo Unix Prolog
?    analyse([[it,rained,all,may],
[it,snowed,all,february],
[there,was,no,sun,in,the,summer],
[the,weather,has,been,awful],
[i,lost,my,job],
[loads,of,planes,crashed],
[many,volcanoes,erupted],
[there,have,been,lots,of,disasters],
[its,been,a,terrible,year]]);

The argument is;
[it,rained,all,may]
[it,snowed,all,february]
[there,was,no,sun,in,the,summer]
[the,weather,has,been,awful]
[i,lost,my,job]
[loads,of,planes,crashed]
[many,volcanoes,erupted]
[there,have,been,lots,of,disasters]
[its,been,a,terrible,year]

Is
[it,rained,all,may]
```

```
evidence for
[it,snowed,all,february]
(answer y or n)  n

Is
[it,snowed,all,february]
evidence for
[there,was,no,sun,in,the,summer]
(answer y or n)  n

Is
[there,was,no,sun,in,the,summer]
evidence for
[the,weather,has,been,awful]
(answer y or n)  y

Is
[it,snowed,all,february]
evidence for
[the,weather,has,been,awful]
(answer y or n)  y

Is
[it,rained,all,may]
evidence for
[the,weather,has,been,awful]
(answer y or n)  y

Is
[the,weather,has,been,awful]
evidence for
[i,lost,my,job]
(answer y or n)  n

Is
[i,lost,my,job]
evidence for
[loads,of,planes,crashed]
(answer y or n)  n

Is
[loads,of,planes,crashed]
evidence for
[many,volcanoes,erupted]
(answer y or n)  n

Is
[many,volcanoes,erupted]
evidence for
[there,have,been,lots,of,disasters]
(answer y or n)  s

Stack is;
[many,volcanoes,erupted]
[loads,of,planes,crashed]
[i,lost,my,job]
[the,weather,has,been,awful]
   [there,was,no,sun,in,the,summer]
   [it,snowed,all,february]
   [it,rained,all,may]

(answer y or n)  y

Is
[loads,of,planes,crashed]
evidence for
[there,have,been,lots,of,disasters]
(answer y or n)  y

Is
[i,lost,my,job]
evidence for
[there,have,been,lots,of,disasters]
(answer y or n)  n

Is
[there,have,been,lots,of,disasters]
evidence for
[its,been,a,terrible,year]
(answer y or n)  y

Is
[i,lost,my,job]
evidence for
[its,been,a,terrible,year]
(answer y or n)  y

Is
[the,weather,has,been,awful]
evidence for
[its,been,a,terrible,year]
(answer y or n)  y

Argument tree;
[its,been,a,terrible,year]
 [there,have,been,lots,of,disasters]
  [many,volcanoes,erupted]
  [loads,of,planes,crashed]
 [i,lost,my,job]
 [the,weather,has,been,awful]
  [there,was,no,sun,in,the,summer]
  [it,snowed,all,february]
  [it,rained,all,may]
yes
?quit;
```

The previous example illustrates how the current stack can be printed out. It is done by responding with *s* when asked a question by the evidence oracle.

Code for the Hybrid Algorithm

```
%
%   This predicate puts the information that F is the father of C in
%   the database. It keeps the appropriate information for obtaining
%   the rightmost child of the father.
%
assert_father(F C) <-
     not(rightmost_child(F _))
     assert(usr father(F C) [])
     assert(usr rightmost_child(F C) []);
assert_father(F C) <-
     rightmost_child(F X)
     delax(usr rightmost_child(F X))
     assert(usr left_brother(X C) [])
     assert(usr rightmost_child(F C) [])
     assert(usr father(F C) []);

%
%   This predicate puts the information that F is the father of C in
%   the database. It keeps the appropriate information for obtaining
%   the rightmost child of the father.
%
assert_father_on_left(F C) <-
     not(rightmost_child(F _))
     assert(usr father(F C) [])
     assert(usr rightmost_child(F C) []);
assert_father_on_left(F C) <-
     rightmost_child(F X)
     assert(usr left_brother(C X) [])
     assert(usr father(F C) []);
%
% This predicate will attach all the sons of L which are evidence for
% N below N. It will also make them no longer sons of L. The sons are
% added in "reverse" order so that the correct rightmost_child relations
% are preserved.
%
attach_sons(L _) <-
     not(father(L _));
attach_sons(L N) <-
     rightmost_child(L Son)
     not(evidence_oracle(Son N _));
attach_sons(L N) <-
     rightmost_child(L Son)
     evidence_oracle(Son N _)
     assert_father_on_left(N Son)
     remove_rightmost(L)
     attach_sons(L N);
```

```
%
%    Calls the predicate for building the argument tree.
%    The first call to assert is to prevent unknown predicate
%    errors from occurring. After building the tree with the
%    predicate "hybrid", the root is found (failing if there
%    is more than one root), and the proper assertion is put in
%    the database.
%
build_tree([H | T]) <-
     assert(usr asserted_evidence(X dummy) [])
     assert(usr left_brother(nil nil) [])
     assert(usr rightmost_child(nil nil) [])
     hybrid_(T H dummy)
     !
     all_of(Roots X father(dummy X))
     eq(Roots [Root])
     assert(usr root(Root) []);
%
%  This predicate will get the list of children of Root in the
%  order in which they are to be printed out
%
get_children(Root Children) <-
     get_children_(Root [] Children);
%
%  This predicate gets the children of Root ordered with the leftmost
%  first
%
get_children_(Root [] []) <-
     not(rightmost_child(Root _));
get_children_(Root [] Children) <-
     rightmost_child(Root Child)
     get_children_(Root [Child] Children);
get_children_(Root [H | T] [H | T]) <-
     not(left_brother(_ H));
get_children_(Root [H | T] Children) <-
     left_brother(Left H)
     get_children_(Root [Left H | T] Children);
%
% This builds the argument tree for hybrid type of arguments. The
% algorithm is from R. Cohen. See the documentation for a detailed
% description.
%
hybrid_([] N L) <-
     evidence_oracle(N L _)
     no_sons_evidence(L N)
     assert_father(L N);
hybrid_([] N L) <-
     evidence_oracle(N L _)
     attach_sons(L N)
     assert_father(L N);
hybrid_([H | T] N L) <-
     evidence_oracle(N L _)
     no_sons_evidence(L N)
     assert_father(L N)
```

```
    hybrid_(T H N);
hybrid_([H | T] N L) <-
    evidence_oracle(N L _)
    attach_sons(L N)
    assert_father(L N)
    hybrid_(T H L);
hybrid_(S N L) <-
    not(evidence_oracle(N L _))
    father(X L)
    hybrid_(S N X);
%
% This succeedes only if the rightmost child of L is not evidence for
% N, or if L has no children.
%
no_sons_evidence(L _) <-
    not(father(L _));
no_sons_evidence(L N) <-
    rightmost_child(L Son)
    not(evidence_oracle(Son N _));
%
% This predicate will remove the rightmost child of L. If it has no
% children, then it succeedes automatically. Otherwise the assertions
% making the sentence the rightmost child of L are removed, and the next
% to rightmost (if it exists) is made the rightmost.
%
remove_rightmost(L) <-
    not(rightmost_child(L _));
remove_rightmost(L) <-
    rightmost_child(L Son)
    not(left_brother(_ Son))
    delax(usr rightmost_child(L Son))
    delax(usr father(L Son));
remove_rightmost(L) <-
    rightmost_child(L Son)
    left_brother(New_son Son)
    delax(usr left_brother(New_son Son))
    delax(usr rightmost_child(L Son))
    delax(usr father(L Son))
    assert(usr rightmost_child(L New_son) []);
%
%   This will perform any special functions required. The only one
%   so far is outputting the tree as it stands.
%
special('t' X _) <-
    printf("Tree is;0 [])
    nl
    print_tree(dummy 0)
    nl
    printf("(answer y or n)  " [])
    get_first_char(X);
special(X X _);
```

Examples of the Hybrid Algorithm

Example 1

```
[1]% wup super_hybrid
Waterloo Unix Prolog
?   analyse([[the,city,is,a,mess],
[the,playground,area,is,run,down],
[the,sandboxes,are,dirty],
[the,swings,are,broken],
[the,parks,are,a,disaster],
[the,highway,system,needs,revamping]]);

The argument is;
[the,city,is,a,mess]
[the,playground,area,is,run,down]
[the,sandboxes,are,dirty]
[the,swings,are,broken]
[the,parks,are,a,disaster]
[the,highway,system,needs,revamping]

Is
[the,playground,area,is,run,down]
evidence for
[the,city,is,a,mess]
(answer y or n)  y

Is
[the,sandboxes,are,dirty]
evidence for
[the,playground,area,is,run,down]
(answer y or n)  y

Is
[the,swings,are,broken]
evidence for
[the,sandboxes,are,dirty]
(answer y or n)  n

Is
[the,swings,are,broken]
evidence for
[the,playground,area,is,run,down]
(answer y or n)  y

Is
[the,sandboxes,are,dirty]
evidence for
[the,swings,are,broken]
(answer y or n)  n

Is
[the,parks,are,a,disaster]
evidence for
[the,swings,are,broken]
(answer y or n)  n

Is
[the,parks,are,a,disaster]
evidence for
[the,playground,area,is,run,down]
(answer y or n)  n

Is
[the,parks,are,a,disaster]
evidence for
[the,city,is,a,mess]
(answer y or n)  y

Is
[the,playground,area,is,run,down]
evidence for
[the,parks,are,a,disaster]
(answer y or n)  y

Is
[the,highway,system,needs,revamping]
evidence for
[the,city,is,a,mess]
(answer y or n)  y

Is
[the,parks,are,a,disaster]
evidence for
[the,highway,system,needs,revamping]
(answer y or n)  n

Argument tree;
[the,city,is,a,mess]
 [the,parks,are,a,disaster]
  [the,playground,area,is,run,down]
   [the,sandboxes,are,dirty]
   [the,swings,are,broken]
 [the,highway,system,needs,revamping]
yes
?quit;
```

Example 1 (Annotated)

```
[1]% wup super_hybrid
Waterloo Unix Prolog
?   analyse([[the,city,is,a,mess],
[the,playground,area,is,run,down],
[the,sandboxes,are,dirty],
[the,swings,are,broken],
[the,parks,are,a,disaster],
[the,highway,system,needs,revamping]]);
```

```
The argument is;
[the,city,is,a,mess]
[the,playground,area,is,run,down]
[the,sandboxes,are,dirty]
[the,swings,are,broken]
[the,parks,are,a,disaster]
[the,highway,system,needs,revamping]

Is
[the,playground,area,is,run,down]
evidence for
[the,city,is,a,mess]
(answer y or n)  y

Is
[the,sandboxes,are,dirty]
evidence for
[the,playground,area,is,run,down]
(answer y or n)  y

Is
[the,swings,are,broken]
evidence for
[the,sandboxes,are,dirty]
(answer y or n)  n

Is
[the,swings,are,broken]
evidence for
[the,playground,area,is,run,down]
(answer y or n)  y

% Here the algorithm will check for
% possible reattachment of sons.

Is
[the,sandboxes,are,dirty]
evidence for
[the,swings,are,broken]
(answer y or n)  n

% [the,swings,are,broken] is still
% the last eligible son.

Is
[the,parks,are,a,disaster]
evidence for
[the,swings,are,broken]
(answer y or n)  n

% The previous brother is not
% checked.

Is
[the,parks,are,a,disaster]
evidence for
[the,playground,area,is,run,down]
(answer y or n)  n

Is
[the,parks,are,a,disaster]
evidence for
[the,city,is,a,mess]
(answer y or n)  y

% Re-attachment of sons is done in
% this case.

Is
[the,playground,area,is,run,down]
evidence for
[the,parks,are,a,disaster]
(answer y or n)  y

% [the,parks,are,a,disaster] is no longer
% eligible since it was included in the
% re-attachment.

Is
[the,highway,system,needs,revamping]
evidence for
[the,city,is,a,mess]
(answer y or n)  y

% Re-attaching sons check

Is
[the,parks,are,a,disaster]
evidence for
[the,highway,system,needs,revamping]
(answer y or n)  n

Argument tree;
[the,city,is,a,mess]
 [the,parks,are,a,disaster]
  [the,playground,area,is,run,down]
   [the,sandboxes,are,dirty]
   [the,swings,are,broken]
 [the,highway,system,needs,revamping]
yes
?quit;
```

Example 2

```
[1]% wup super_hybrid
Waterloo Unix Prolog
?    analyse([[its,been,a,good,winter],
```

```
[i,made,progress,on,my,thesis],
[i,found,a,new,apartment],
[it,wasnt,too,cold],
[and,it,didnt,rain,too,much],
[actually,the,weather,was,good]]);

The argument is;
[its,been,a,good,winter]
[i,made,progress,on,my,thesis]
[i,found,a,new,apartment]
[it,wasnt,too,cold]
[and,it,didnt,rain,too,much]
[actually,the,weather,was,good]

Is
[i,made,progress,on,my,thesis]
evidence for
[its,been,a,good,winter]
(answer y or n)  y

Is
[i,found,a,new,apartment]
evidence for
[i,made,progress,on,my,thesis]
(answer y or n)  n

Is
[i,found,a,new,apartment]
evidence for
[its,been,a,good,winter]
(answer y or n)  y

Is
[i,made,progress,on,my,thesis]
evidence for
[i,found,a,new,apartment]
(answer y or n)  n

Is
[it,wasnt,too,cold]
evidence for
[i,found,a,new,apartment]
(answer y or n)  n

Is
[it,wasnt,too,cold]
evidence for
[its,been,a,good,winter]
(answer y or n)  y

Is
[i,found,a,new,apartment]
evidence for
[it,wasnt,too,cold]
(answer y or n)  n

Is
[and,it,didnt,rain,too,much]
evidence for
[it,wasnt,too,cold]
(answer y or n)  n

Is
[and,it,didnt,rain,too,much]
evidence for
[its,been,a,good,winter]
(answer y or n)  y

Is
[it,wasnt,too,cold]
evidence for
[and,it,didnt,rain,too,much]
(answer y or n)  n

Is
[actually,the,weather,was,good]
evidence for
[and,it,didnt,rain,too,much]
(answer y or n)  t

Tree is;

[its,been,a,good,winter]
   [i,made,progress,on,my,thesis]
   [i,found,a,new,apartment]
   [it,wasnt,too,cold]
   [and,it,didnt,rain,too,much]

(answer y or n)  n

Is
[actually,the,weather,was,good]
evidence for
[its,been,a,good,winter]
(answer y or n)  y

Is
[and,it,didnt,rain,too,much]
evidence for
[actually,the,weather,was,good]
(answer y or n)  y

Is
[it,wasnt,too,cold]
evidence for
[actually,the,weather,was,good]
(answer y or n)  y
```

```
Is
[i,found,a,new,apartment]
evidence for
[actually,the,weather,was,good]
(answer y or n)  n

Argument tree;
[its,been,a,good,winter]
 [i,made,progress,on,my,thesis]
 [i,found,a,new,apartment]
 [actually,the,weather,was,good]
  [it,wasnt,too,cold]
  [and,it,didnt,rain,too,much]
yes
?quit;
```

The previous example illustrates how the current argument tree can be printed out. The is done by responding with *t* when asked a question by the evidence oracle.

References

R. Cohen, Investigation of Processing Strategies for the Structural Analysis of Arguments, *Proceedings of ACL Conference*, pp. 71-75 (June, 1981).

R. Cohen, A Computational Model for the Analysis of Arguments, *University of Toronto Technical Report CSRG-151*, (October, 1983).

R. Cohen, A Theory of Discourse Coherence for Argument Understanding, *Proceedings of CSCSI/SCEIO Conference*, pp. 6-10 (May, 1984).