AN APPLICATION OF ABSTRACT INTERPRETATION IN SOURCE LEVEL PROGRAM TRANSFORMATION.

Daniel De Schreye, Maurice Bruynooghe
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3030 Heverlee, Belgium.

Abstract. We describe an application of abstract interpretation within the field of source-to-source program transformation for pure Horn clause logic programs. Using a very concrete setting, we aim to provide a comprehensible introduction to the technique of abstract interpretation, particularly suited for the novice in the field. Also, we argue that abstract interpretation is not only suited for applications in code optimization, but provides an excellent tool to support techniques in source level program transformation.

1. Introduction.

If one aims to prove general properties of programs, it is of crucial importance to have the ability of performing some kind of data abstraction. Although the runtime behavior observed during a concrete execution of a program may provide an example to support our expectations on the presence or absence of certain properties, a technique for interpreting the program using abstract data is needed to be able to prove these properties in general.

P.Cousot and R.Cousot in [10] were the first to thoroughly describe a general mechanism for the abstract interpretation of imperative programs. Very recently, several successful efforts were made to adapt their technique to logic programming. General reformulations have been presented by C.S.Mellish [22], N.D.Jones and H.Sondergaard [20],T.Kanamari and T.Kawamura [21] and M.Bruynooghe [6], applications in code optimization for logic programs have been described by S.K.Debray and D.S.Warren [12] and M.Bruynooghe et al. [5]. Also, a first description of the use of abstract interpretation in program transformation - more particularly in program specialization - is presented by J.Gallagher and M.Codish [16].

The increasing attention that abstract interpretation has obtained from researchers active in the field of logic programming can be explained by several reasons. First, there is the desire to obtain a better runtime efficiency for declarative programming languages such as Prolog. Through code optimizations obtained from mode inference, type inference and compile time garbage collection, there is high hope of eventually achieving runtime efficiencies of logic programs that are comparable to those of their imperative equivalents. Also, there is the ease with which abstract interpretation can be described and implemented within the setting of logic programming. This is partially due to the fact that a language such as Prolog contains its own meta-language, increasing the ability of writing various types of interpreters for the language. A second reason is that, because of its high declarativity and its data structuring facilities, the language is particularly well suited for symbol manipulation.

From the currently available literature on the abstract interpretation of programs, a general impression that one obtains, is that the theory is quite complex and hard to comprehend. Contributions to the field, such as [12], [5] and, most notably, [6], have improved the accesibility of the subject to some extend. However, most authors prefer to introduce the theory in a setting covering a wide range of potential applications. In doing so, they increase the level of abstraction of their presentation and end up giving the non-expert a hard time figuring out how this most powerful technique can be applied to solve his problem at hand.

This paper mainly addresses the novice in the field. It does not contribute any essential new features to the theory of [10], nor does it improve on the reformulation of abstract interpretation within the setting of logic programming by M.Bruynooghe [6]. It is merely an attempt to illustrate the type of considerations and the degree of creativity that are required to make abstract interpretation work for you.

In order to do so, we start with a concrete problem selected from the field of source-to-source transformation for logic programs. This application was thoroughly described in [3], [4] and [13] and deals with the compilation of ideal control rules into existing declarative Horn clause programs. It was selected for various reasons. First, it seemed appropriate not to focus on a problem within the field of code optimization, to illustrate that the applicability of the technique is not limited to this field of research. Secondly, several topics in program transformation, such as loop detection [2], [25], partial evaluation [16], fold/unfold [7], [19], [27], and the elimination of redundant computation [14], make use of techniques that are closely related to abstract interpretation, without explicitly refering to it. Finally, the selected problem involves a combination of mode-, structure- and aliasing inference and is composed of different layers where abstract interpretation is of use, so that even with one concrete application, we can illustrate most of the power of the technique.

We start off with some preliminaries on Horn clause logic and a first, high level introduction to the different steps that are encountered in building an application of abstract interpretation in section 2. Section 3, introduces the example application in program transformation and provides more details on the different steps that are encountered in a first layer solution for the application. In section 4, a more complete, second layer solution to the same problem is provided. Finally, we end with a discussion of the wide range of potential applications for the technique within the field of program transformation.

2. Preliminaries and high level approach.

The language for which both the transformation technique and the abstract interpretation are discussed is that of *pure* Horn clause logic with the SLD-resolution mechanism of Prolog. In this language, a program is a finite set of Horn clauses which are of the form:

$$A \leftarrow B_1, B_2, \cdots, B_n, \ n \geqslant 0,$$

and a goal clause or query

$$\leftarrow Q_1, Q_2, \cdots, Q_n, \ n \geqslant 1,$$

where A, B_1, B_2, $\cdots$, B_n and Q_1, Q_2, $\cdots$, Q_n are atomic formulae of the form

$$P(t_1, t_2, \cdots, t_m) \ , \ m \geqslant 0,$$

with an m-ary predicate symbol P and terms t_1, t_2, $\cdots$, t_m. Terms are either variables, constants or are constructed from functors whose arguments are again terms.

The query activates the program. The computation rule selects a subgoal Q_i from the query; the search rule selects a clause $A \leftarrow B_1, B_2, \cdots, B_n$ whose left hand side (head), A, has the same predicate symbol as Q_i. An attempt is made to find a most general unifier (mgu) θ such that $Q_i\,\theta = A\,\theta$ (for the clause a fresh set of variables is supplied). The query is replaced by the new goal

$$\leftarrow Q_1\,\theta, \cdots, Q_{i-1}\,\theta, B_1\,\theta, \cdots, B_n\,\theta, Q_{i+1}\,\theta, \cdots, Q_r\,\theta\,.$$

if unification succeeds. For some predicates (such as arithmetic operations) the predicate's definitions are built into the system. With a depth first strategy, the computation always proceeds with the most recently generated goal for which there are untried clauses matching the selected subgoal Q_i. The original query succeeds when the empty goal is derived and the composition of all mgu's applied on the variables of the original query yields the answer. All solutions are generated when all selected subgoals have exhausted their candidate clauses.

The order in which the subgoals in the query are selected for unification is determined by the computation rule. The standard computation rule of Prolog selects the subgoals in the query from left to right.

For a given query, we can represent the execution of a program under such a rule in a proof tree. The different goals obtained during the execution are the nodes in the tree. The proof tree contains an arc from one goal to another for every successful resolution step. The mgu is added as a label on the arc (for simplicity we will only specify the substitutions caused on the variables of the subgoal). The sequence of resolution steps performed by the theorem prover is found by tracing the tree depth-first, left-to-right.

Before giving an example to illustrate this, we sum up some conventions. Variable names start with a lower case character; constants, functors and predicate names with an upper case character. The infix notation x.y is used to denote a list with head x and tail y.

The example program is *Permutation_sort*, consisting of the following Horn clauses:

Sort(x, y) ← Perm(x, y), Ord(y) .
Perm(Nil, Nil) ← .
Perm(x.y, u.v) ← Del(u, x.y, w), Perm(w, v) .
Del(x, x.y, y) ← .
Del(x, y.u.t, y.v) ← Del(x, u.t, v) .
Ord(Nil) ← .
Ord(x.Nil) ← .
Ord(x.y.z) ← x⩽y, Ord(y.z) .

This program sorts a list of numbers by first permuting it and next testing whether the permuted list is ordered. With an initial query ← Sort(2.1.Nil, x), we get the proof tree of Fig.1 .

Along with the procedural interpretation of logic programs described above, there is a declarative one. Informally, in this second interpretation a Horn clause

$$A \leftarrow B_1, B_2, \cdots, B_n , \; n \geqslant 0,$$

expresses that for all possible assignments of values (taken from some predefined universe) to the variables occurring in A, B_1, B_2, $\cdots$ and B_n , the instantiated predicate A will hold if the conjunction of all the instantiated predicates $B_1, B_2, \cdots, B_n$ holds. A query

$$\leftarrow Q_1, Q_2, \cdots, Q_n , \; n \geqslant 1,$$

states that no value assignment for the variables in Q_1, Q_2, $\cdots$ and Q_n can exist, such that the conjunction of instantiated predicates Q_1, Q_2, $\cdots$, Q_n holds. Augmenting the program with a query can be viewed as adding the hypothesis that no solutions for Q_1, Q_2, $\cdots$ and Q_n can be deduced from the given set of Horn clauses in the program. The computation uses unification and the *modus tolens* deduction rule from mathematical logic, to derive a new hypothesis

$$\leftarrow Q_1\,\theta, \cdots, Q_{i-1}\,\theta, B_1\,\theta, \cdots, B_n\,\theta, Q_{i+1}\,\theta, \cdots, Q_r\,\theta .$$

again expressing that no further substitutions for variables in these predicates can exist such that the conjunction holds. The ultimate goal is to obtain the empty query (←.), which is a notation for contradiction. At this point, a proof by contradiction is completed for the initial conjunction of goals Q_1, Q_2, $\cdots$, Q_n, and the composition of all parameter substitutions yields a set of values such that the conjunction holds.

In this sense, a Prolog computation can be viewed as proving a theorem. For the Permutation_sort example above, the query ← Sort(2.1.Nil, x) activates a proof for the fact that x/1.2.Nil is a substitution for which Sort(2.1.Nil, 1.2.Nil) is true. This is the main reason why Prolog is an excellent programming environment for performing abstract interpretation. The same theorem prover that is used to compute the results of programs, can - with some adaptation - be used to prove theorems concerning the program's behavior. As an easy example: if we redefine the builtin predicate $\leqslant/2$ to succeed for any two arguments, then by activating the program with a query ← Sort(x.y.Nil, z) we prove that if the first argument of Sort/2 is a list of length 2 then the second argument of Sort/2 must also have this structure. Here, we made abstraction of the explicit content and type of the arguments x and y in the list. Therefore, the resolution mechanism had to be altered in such a way that any explicit reference to those values or types is omited. So, $\leqslant/2$ was redefined.

However, in this easy example, the abstractions x and y still constitute Prolog variables and the unification mechanism of Prolog can deal with them precisely in the way we expect it to. This is not always the case. If we would be interested in the mode behavior of our program for instance, then we could assign meaning to ← Sort(Ground, Var) as being an abstract representation for the mode pattern we wish to investigate our program's behavior for. But without adaptation, the

Prolog unification algorithm, is unable to perform even one single resolution step starting from this query. Thus, along with the abstract representation we wish to use, the unification algorithm will be redefined.

In general, there are six different types of activities involved in building a concrete application using abstract interpretation:

1. Chosing the desired **level of data-abstraction.** What type of information do we wish to make abstraction of and what (other) information has to remain explicit during the interpretation. This step is highly dependent on the application at hand. It determines the class of abstract substitutions that are allowed for each predicate occurring in the program (the *abstract domain*).

2. Closely related to the choice of the abstraction level is the selection of a **representation** for the abstract substitutions.

3. The **unification mechanism** has to be redefined to be able to cope with resolution within the predefined abstract domain and its representation.

4. The behavior of **builtin predicates** has to be specified explicitly. More specifically, for each builtin predicate and each possible abstract pattern with which it may be invoked during the interpretation, the outcoming abstract substitution (and therefore the effect on the current set of pending goal statements) has to be made explicit.

5. A **special purpose interpreter** has to be written to support the abstract interpretation. This interpreter is not only intended to incorporate the new unification mechanism and builtin behavior, but should also serve the purpose of avoiding infinite loops, which will occur more frequently in an abstract interpretation than in a corresponding concrete execution due to the data abstraction.

6. Some applications require that the abstract interpretation behaves in a deterministic way. In particular, for code optimization problems, we are not interested in type- or mode inference within each branch in the search tree separately, but we want to deduce the strongest possible statement which holds for all possible solutions for the given procedure. This means that whenever nondeterminism occurs, a **least generalization** of all the computed abstract output substitutions has to be made.

Each of these six steps will be further explained and illustrated in the following sections.

3. A simple application : building abstract proof trees.

The problem discussed in [4] and [13] is as follows. Suppose that a highly declarative - but inefficient - Horn clause program is given, together with a special computation rule (different from the standard computation rule of Prolog, described above) for this program. From these two, synthesize a new Horn clause program that has the same computational behavior under the standard computation rule as the old program has under the special rule.

The solution for this problem, proposed in [4] and developed in more detail in [13], consists of two parts: first build a symbolic (abstract) proof tree for the execution of the original program under the new computation rule; then generate new Horn clauses that synthesize all the observed transitions in the obtained tree. Here, we focus on the first of these objectives. It gives rise to a first layer within the problem, where abstract interpretation is of use.

In this phase of the transformation method, an abstract query and an appropriate computation rule for the given program and query are specified. Then, the abstract proof is constructed. It traces the computation that will occur for any concrete query matching the abstract query pattern. Therefore - but also to distinguish it from a concrete proof tree -, we will refer to it as the trace tree in what follows.

3.1 Selecting the abstraction level.

The process of generating the trace tree must be guided by the computation rule. Therefore, it is essential to determine what kind of information the computation rule will need to perform this task. Several researchers have worked on the control of logic programs and have proposed mechanisms for expressing and enforcing new control rules. Some suggest the use of meta interpreters [17], [26] leading to languages with extremely rich control features. Others propose logic programming environments which include a separate control language to enforce the appropriate goal- or clause selection at run time [18], [8], [23], [30], [9]. A third group describe methods to compile control rules through source-to-source program transformation [19] , [27], [4], [24].

All the annotations, declarations or meta-predicates suggested in these papers aim at controlling the execution of programs on the basis of one of the following criteria:

- A goal is selected for expansion if it is sufficiently instantiated. This type of condition is quite simple to verify; it merely requires the ability to test whether some variable is either free (uninstantiated) or ground (fully instantiated). We call this type of control information 'static'.
- A goal is selected for expansion if certain of its variables will not become further instantiated due to this additional expansion. Here the condition is more complicated since a further resolution step has to be performed to detect whether a variable becomes instantiated. We call it control of the 'dynamic' type.

As in [4], [13] we use 'static' information to describe our computation rules, since the idea of selecting subgoals on the basis of their obtained instantiation pattern is quite natural and easy to understand and support. Since the computation rule will select a subgoal from the most recently obtained state (node) in the abstract trace tree, the abstraction level that we will use within the tree should contain information on whether a variable is free or ground. As an example, an appropriate abstract query pattern for the Permutation_sort program could be

$\leftarrow$ Sort(x , y) , where x is ground and y is free.

A second piece of information in our data of which we do not want to make abstraction are the functors appearing in the arguments of the goals. With the clauses for the predicate Ord/1 in the

Permutation_sort program, it seems quite appropriate to expand

← Ord(x) , where x is ground,

and

← Ord(x.y.z) , where x and y are ground and z is free

but inappropriate to expand goals of type

← Ord(x) , where x is free

and

← Ord(x.y) , where x is ground and y is free.

In the second of these four abstract queries, the two first members in the list are instantiated and therefore the test $x \leqslant y$ is ready to be performed. Obviously, this is not the case in query three and four.

Finally, we want to include in the abstraction level, all information concerning bindings between (free) variables. The reason why this is needed should be clear from the following example. Suppose that at some point during the computation we obtain a state of type

← A(x, y) , B(y, z) , C(z) , where x, y and z are free.

If the program contains a clause such as

A(0, 0).

then selecting the goal A(x, y) for expansion will lead to a new state

← B(y, z) , C(z) , where y is ground and z is free.

However, without the information on the binding between the second parameter of A/2 and the first of B/2, this result could not be obtained and the information on instantiation in the trace tree would be incorrect. In fact, it is very unlikely that there exists an application of abstract interpretation which does not rely on binding or sharing between variables at all.

3.2 Representing the data-abstraction.

A general way to describe the abstract pattern which a predicate has obtained at some point during the computation uses abstract substitutions [6]. An abstract substitution is a high level description of the set of concrete substitutions for the arguments of the predicate. Through such sets of substitutions we can express abstract patterns. As an example, an abstract instantiation pattern for the predicate Ord/1

Ord(x.y.y) , where x is ground and y is free

could be represented as the pair

```
( Ord( x),
        { x ← y.z.t ,
          y ← ground ,
          z ← free ,
          binding( t, z) } ) .
```

Here, the second entry in the pair is a description of the set of all concrete substitutions for x which, during a concrete execution of the program, may still occur. In principle, there is no restriction on the syntax that is used to describe these abstract substitutions. The user is free to define them in any way he wants. Only, he must ensure that each syntactic expression occurring in the abstract substitutions is properly supported by his new definition of the unification algorithm and of the effect of calls to builtin predicates. As another example, the description

{ x, y, possible_share(x, y), type(x, Int) }

is a typical abstract substitution of a type-inference application.

However, since the choice of syntax is free, we will use a more compact representation to describe the instantiation patterns of predicates in our application. The example pattern for Ord/1 given above, will be represented as

Ord($G.V_1.V_1$) ,

where G is a constant representing any ground term and V_i, $i \in \mathbb{N}$ is a constant representing a particular free variable. Bindings between variables are made explicit by using the same index i for all occurrences of the variable throughout the state.

Although this representation is elegant in the sense that it does not introduce complex new notations, it suffers from the inconvenience that different abstract patterns in this representation refer to the same state. Renumbering the indices i in the V_i's amounts to an equivalent representation and the replacement of a term including no V_i's by G results in a pattern which describes the state as well, e.g.

← Perm(G.G.G, V_1), Ord(V_1).

could also be represented as

← Perm(G, V_2), Ord(V_2).

In order to overcome this, one must define a *canonical* abstract representation of a state. Here, it is introduced as a representation of the above type, in which the first occurrences of each V_i are numbered starting from 0 and ascending from the left hand to the right hand side in the state and where each term containing no V_i's is replaced by G.

3.3 Redefining the unification algorithm.

Obviously, some redefinition is needed since the unification for calls of type

$\leftarrow$ Ord(G).

or

$\leftarrow$ Ord(V_1).

with the clauses for Ord/1 using the Prolog unification mechanism will simply fail, which does not correspond to our intentions. This problem can easily be solved if the abstract interpretation is itself described as a Horn clause logic program. If this is the case, then the following steps can be taken:

1. Replace every occurrence of G in the abstract state by a fresh free variable and each different occurrence of a V_i as well (using the same variable for all occurrences of V_i with the same index i). At the same time, build a list containing all new variables associated to a G and a second list containing corresponding pairs of new variables and their associated V_i's (see the example below for a concrete illustration).

2. Perform a resolution step with the selected goal from the newly obtained state and a clause from the program, using the Prolog unification mechanism.

3. Now, because of the resolution step, the appropriate substitution has been applied to the free variables - not only in the state itself, but also in the two lists expressing the correspondences. What remains to be done is to reconvert the obtained state into the proper abstract form. It is for this purpose that the two correspondence lists are kept. They are used in three steps:

   - Replace all ground terms occurring in the first list by G and instantiate every free variable in it to G.
   - Replace all ground terms occurring in the first argument of a pair in the second list by G and unify each first argument of a pair which is still uninstantiated with the corresponding second argument.
   - Finally, instantiate all remaining free variables in the new goal list by V_j's, where the j's are fresh indices.

This leads to a program scheme of the form:

```
Expand_selected( selected_goal, other_goals, new_goals) ←

        Build_free_state( selected_goal.other_goals,
                          free_goal.free_others,
                          variables_for_Gs, pairs_of_variables_and_Vi),
        Clause( free_goal , goals_from_body),
        Append( goals_from_body, other_goals, new_goals),
        Instantiate_ground_list( variables_for_Gs),
        Instantiate_variables_list( pairs_of_variables_and_Vi),
        Instantiate_new_variables( goals_from_body).
```

and for an example query such as

← Expand_selected(Perm(G, V_1), Ord(V_1).Nil, new_goals).

and through unification with the second clause for Perm, this results in the following computation

- Build_free_state/4:
  free_goal.free_others ← *Perm(x, y).Ord(y)* ,
  variables_for_Gs ← *x.Nil* ,
  pairs_of_variables_and_Vi ← (V_1, *y).Nil*
- Clause/2:
  x ← *z.t* , y ← *v.w*
  goals_from_body ← *Del(v, z.t, u).Perm(u, w).Nil*

  Observe that these first two substitutions also cause the instantiations
  variables_for_Gs ← *z.t.Nil* , and
  pairs_of_variables_and_Vi ← (V_1, *v.w).Nil*
- Append/3:
  new_goals ← *Del(v, z.t, u).Perm(u, w).Ord(v.w).Nil*
- Instantiate_ground_list/1:
  z ← *G* , t ← *G*
- Instantiate_variables_list/1 performs no instantiations
  (pairs_of_variables_and_Vi = (V_1, *v.w).Nil* contains no ground terms nor free variables in a second argument of a pair)
- Instantiate_new_variables/1:
  v ← V_2 , w ← V_3 , u ← V_4

Thus, this results in the new abstract state:

← Del(V_2, G.G, V_4), Perm(V_4, V_3), Ord(V_2.V_3)

3.4 Redefining the builtin predicates.

Again, the problem is that presenting abstract calls to builtin predicates causes failure (or runtime errors) for most cases and results in an undesired behavior for others. Typical examples in our setting are:

← V_1 = G , (fails, where it should instantiate V_1)

← V_1 = V_2 , (fails, where it should cause a binding)

← V_1 ⩽ V_2 , (succeeds, where it should fail)

The only way it can be handled is by predefining the desired effect for calls to builtin predicates for each possible call-pattern. This could, for instance, be done with a predicate Abstract_builtin/2, for which we define - among many others - the following Horn clauses, in

order to deal with the examples above:

Abstract_builtin($V_1 = G$, (V_1/G.Nil).Nil).

Abstract_builtin($V_1 = V_2$, (V_2/V_1.Nil).Nil).

Abstract_builtin($V_1 \leqslant V_2$, Nil).

The first argument of Abstract_builtin/2 contains a *canonical* abstract call-pattern to a builtin predicate. The second is a list of lists, containing one sublist for each different abstract output substitution with which the given abstract call can succeed. These abstract substitutions are needed, since they must be applied to all goals occurring in the state where the abstract call to the builtin predicate was selected from.

In the clauses stated above, we express that the call-pattern $V_1 = G$ can only succeed with one possible output substitution, namely $V_1 \leftarrow G$. This is also the case for $V_1 = V_2$, with $V_2 \leftarrow V_1$. The third example deals with a failing call-pattern.

3.5 The interpreter for solving an abstract goal.

The techniques described in the previous subsections are sufficient to be able to develop an abstract trace tree for Permutation_sort with the abstract query pattern ← Sort(G, V_1) and a computation rule that differs only from the standard computation rule in its eagerness to expand any call to Ord/1 with instantiation

Ord(G) or Ord(G.G.V1).

A finite part of the resulting trace tree is very similar to the one displayed in Fig.2, except that it contains additional branches originating from the expansion of the Del(V_i, G.G, V_j) goals.

The special way in which the expansion in the trace tree of Fig.2 deals with the Del(V_i, G.G, V_j) goals is due to the following pragmatic - but for large-size program transformations essential - observation. Often, a program consists of two types of procedures: those that need additional control directives - mostly coroutining - in order to become efficient, and those that already behave efficiently under the standard computation rule. In the example, calls to Del(V_i, G.G, V_j) do not play an active role in the coroutining process between Perm/2 and Ord/1 and are therefore of the second type. If we were to use the abstract interpretation to expand all goals, even when most of them do not need a transformation, then for real-life problems, the size of the resulting trace trees would become unacceptably large.

The way we will deal with goals for which transformation is unnecessary is very similar to the way we approach builtin predicates. The only information regarding them that should be made explicit in the trace tree consists of the abstract pattern of the goal before the call and the outcomming substitution obtained from completely solving it with the standard computation rule. This substitution must then be applied to the remaining goals in the state. In fact, we could make the approach completely similar to that of the builtin predicates by adding a fact of type Builtin_pattern/2 for each such call and abstract pattern. However, this would mean that new

facts of this type would have to be produced by the user for each new transformation session. Instead, we will use the abstract interpretation itself to determine the abstract substitutions resulting from completely solving a given abstract goal. This is where some new and more interesting aspects regarding the design of the application turn up.

Solving an abstract goal is different from building (a finite part of) its abstract trace tree because recursive programs have infinite abstract trace trees. Thus, during the expansion we need criteria to determine whether or not additional output substitutions, different from the ones we already obtained, exist. In other words: abstractly solving a goal usually leads to an infinite number of success nodes, even if each corresponding concrete goal - each instance of the abstract pattern - results in a finite computation. The reason is that although certain calls occurring in a concrete execution contain different data, their abstractions may become identical and therefore the interpretation may infinitely loop.

The problem of infinite looping turns up when the recursive expansion of an abstract goal A_1 and the goals descending from it, eventually leads to the expansion of a descendent goal A'_1, such that A_1 and A'_1 have identical canonical patterns. A good way to formally describe and solve the problem is by representing the computation by means of an abstract AND-OR tree, since this type of tree is very explicit in representing the ancestor-descendent relation between the subgoals of the states.

An AND-OR tree is similar to the trace trees we have used so far, but instead of representing a conjunction of goal statements (what we called a *state*) as a node in the graph, we represent

- each subgoal individually (OR-node),
- for each clause which can be applied to the query in the OR-node, a descending node containing the conjuction of all the subgoals obtained from the body of the clause, after the unification of the query with the head has been performed (AND-node).
- for each AND-node, we represent all the subgoals in this node a second time, as the OR-nodes descending from it.

In Fig.3 we have drawn the - abstract - AND-OR-tree which describes the infinite loop situation in abstract interpretation.

To avoid entering infinite loops and still guarantee that all solutions be computed, the following technique is built into the abstract interpreter:

- We do not expand the descendent goal A'_1, but instead we freeze and record the entire computation (including the resulting substitutions) leading from the ancestor A_1 over all intermediate AND- and OR- nodes to A'_1.
- Then, the computation is continued as if the query $\leftarrow A'_1$ has failed. It backtracks to the latest OR-node in the tree for which there are untried clauses in the program.
- When eventually an output substitution for the goal A_1 has been computed, a proper renaming of the V_i's occurring in this output substitution yields an output substitution for the suspended OR-nodes A'_1 as well, because A_1 and A'_1 have identical canonical forms. Using the renaming as the output pattern that would result from solving A'_1, we can reactivate the suspended AND-OR-subtree and continue its computation in an attempt to derive additional output patterns for A_1.

- This process is continued until saturation occurs, i.e. until all output patterns obtained for A_1 have been used for A_1 and no further new patterns for A_1 can be derived in this way.

This method involves the following potential disadvantages:

1. If the computation for A_1 terminates without generating at least one output substitution for A_1, or if it does not terminate at all - due to an infinite subtree descending from some OR-node which is not of the loop type described above - then none of the suspended computations can ever be reactivated. Thus, we could possibly has lost some output patterns for A_1.
2. For some programs, infinitely many different output instantiation patterns exist, so that the point of saturation is never reached.

First, we discuss the reactivation of suspended goals. Assuming that the application has been built correctly - see [6] for criteria and proofs on correctness -, the AND-OR-tree for a concrete query with instantiation pattern $\leftarrow A_1$ is obtainable by taking a subtree from the abstract AND-OR-tree and replacing the abstract goals by their appropriate concrete instances. For the problem case with a terminating computation, this implies that the concrete AND-OR-tree will not produce a success node either. For a nonterminating computation the matter is more problematic. Here, it is possible that, although the abstract interpretation - using a depth-first search on the AND-OR-tree - can never continue its search for further solutions outside the infinite subtree, a corresponding concrete computation may fail at some point within the infinite branch and eventually lead to more solutions when backtracking.

Such infinite subtrees (different from the ones we suspend in the algorithm) can only occur in applications involving an infinite abstract domain. This is clear, since the OR-nodes in any descending path of a branch in the abstract AND-OR-tree must be either mutually distinct or suspended. This is why most applications of abstract interpretation make use of finite domains. This ensures termination of the interpretation and in most cases a finite domain is a necessary condition for completeness (see [6] for more details).

In our application, this is difficult to realize, because in principle recursive data structures can create functors with arbitrary complex instantiation patterns. The way that it is dealt with in [13], is by introducing an additional constant A to describe the abstract patterns. This constant takes its place among the abstract terms G and V_i and is used as an abstraction for any term, whatever its instantiation pattern. However, it is not allowed to occur within the actual abstract resolution, but only within some predefined declarations concerning certain positions within certain predicates. Its use is to express an upper bound beyond which the instantiation of a (recursively defined) term within a predicate becomes irrelevant to the computation.

For instance, a declaration of the form Ord(G.G.G.A) expresses that we are not interested in computations that provide instantiations of Ord(x) beyond the first three ground members of the list x.

Obviously, such declarations must be provided by the user, since they require knowledge regarding the semantics of the program. Their sole purpose is to interrupt possible infinite

computations and to force the abstract interpretation to backtrack within the AND-OR-tree. In some sense, it can be regarded as an artificial way to reduce the abstract domain to a finite size.

Although the second problem, concerning the generation of infinitely many output patterns for a suspended query pattern has a quite different origine, the solution is identical. The declarations we have just described, will also cause the termination of these infinite processes.

3.6 Generalization.

Computing the output pattern obtained from solving an abstract goal is a non deterministic application of abstract interpretation. It generates a number of different solutions. Each different resulting pattern will give rise to a new branch in the abstract trace tree we originally set out to construct. For many applications however, it is more interesting to obtain only one abstract substitution that generalizes all those obtained in the different underlying branches. This is the case for all applications regarding code optimization. One implication is that the generalization of a set of abstract substitutions must be possible. To ensure this, the abstract domain of the application should be designed with the structure of a partially ordered set with a largest element. In order to obtain powerful results, a lattice is preferable. So, for each set of abstract substitutions (which are representations for sets of concrete substitutions themselves), there should be a least general abstract substitution, representing a set of concrete substitutions containing all concrete substitutions in the original abstract ones. A second implication is that the interpreter has to be developed with a breath-first approach instead of the depth-first approach we used in the previous subsection. The reason is that for every OR-node, all different solution should first be obtained, then generalized and finally the generalized solution should be passed to the ancestor nodes. With our previously described method, each individual solution to an OR-node is passed on to the ancestors.

We will not go into more detail on the matter of generalization, since it is our experience that in applications of program transformation there is very little need for it.

4. A second layer of abstract interpretation: completeness.

In the previous section we showed how the solutions of an infinite AND-OR-tree are computed by a special algorithm. However, if recursive clauses are expanded, the abstract tree for the original query is also infinite. After a finite number of steps, the expansion has to be stopped. The problem is to determine at which point sufficient information regarding the computation has been gathered in the tree, in order to be able to synthesize a new program from it, equivalent to the old one. Again, abstract interpretation is an adequate tool for performing this task.

The idea is as follows. The original program contains only a finite number of clauses. If we assume that the new computation rule is finitely expressable in terms of a finite number of different instantiation patterns which may occur during the computation (see [13] for a formal definition of *instantiation based computation rules*), then the number of essentially different transitions occurring in the trace tree is finite as well. With *essentially different* we mean that either

- a different instantiation atom was selected from the initial state, or

- a different clause from the program was used for the expansion, or
- the changes in the instantiation patterns of the remaining subgoals in the state caused by the expansion are different (i.e. the bindings between the selected subgoal and the other goals in the initial state are different).

The synthesis algorithm of [13] generates a new Horn clause for every equivalence class of mutually *similar* (not essentially different) transitions observed in the trace. The problem left to the abstract interpretation is to determine, at which point during the expansion of the tree, all the equivalence classes have been obtained.

In order to accomplish this, we introduce a further abstraction on nodes (states) in the abstract trace tree. This abstraction will be refered to as a *state description*. To introduce them on an informal basis, we recall some observations made in the manual completeness proof for the Sieve of Eratosthenos example in section 3 of [4].

The program is defined by the following set of Horn clauses:

Primes(n,p) ← Integers(2,i), Sift(i,p), Length(p,n).
Integers(n,Nil) ←.
Integers(n,n.i) ← m is n+1, Integers(m,i).
Sift(n.i,n.p) ← Filter(n,i,f), Sift(f,p).
Sift(Nil,Nil) ←.
Filter(n,m.i,f) ← Divides(n,m), Filter(n,i,f).
Filter(n,m.i,m.f) ← not Divides(n.m), Filter(n,i,f).
Filter(n,Nil,Nil) ←.
Length(Nil,0) ←.
Length(h.t,n) ← n > 0, m is n - 1, Length(t,n).

The completeness proof in [4] considers a number of *similar* states from a finite subtree of the abstract trace tree, e.g.

- Integers(G,V_1), Filter(G,G.V_1,V_2), Sift(V_2,V_3), Length(V_3,G)
- Integers(G,V_1), Filter(G,G.V_1,V_2), Filter(G,V_2,V_3), Sift(V_3,V_4), Length(V_4,G)
- Integers(G,V_1), Filter(G,V_1,V_2), Filter(G,G.V_2,V_3), Sift(V_3,V_4), Length(V_4,G)
- Integers(G,V_1), Filter(G,V_1,V_2), Filter(G,G.V_2,V_3), Filter(G,V_3,V_4), Sift(V_4,V_5), Length(V_5,G)

and associates to them a general description of instantiation patterns and bindings occurring in them:

Integers(G,V_1), Filter(G,V_1,V_2), $\cdots$, Filter(G,V_{i-1},V_i), Filter(G,G.V_i,V_{i+1}),
Filter(G,V_{i+1},V_{i+2}), $\cdots$,Filter(G,V_{n-1},V_n), Sift(V_n,V_{n+1}), Length(V_{n+1},G)

The pattern is that each example state contains precisely one goal of the types Integers(G,V_1), Filter(G,G.V_i,V_{i+1}), Sift(V_n,V_{n+1}) and Length(V_{n+1},G). Also, it contains 0 or more goals of type Filter(G,V_i,V_{i+1}). The binding pattern is a chain connecting the Integer/2 goal to 0 or more Filter(G,V_i,V_{i+1}) goals, then to the more instantiated Filter/3, which is again connected to 0 or more Filter(G,V_i,V_{i+1}) goals and finally to the pair Sift(V_n,V_{n+1}), Length(V_{n+1},G).

Different patterns of the above type are generated, so that eventually every state in the finite part of the trace tree is an instance of a state description. Then, it is proved that starting from any of these descriptions and performing a resolution step expanding the subgoal selected by the computation rule, an instance of another state description is obtained.

When for a given finite part of the abstract trace tree this result can be proved, it implies that all relevant transitions have been gathered from the trace and that the abstract expansion is in some particular sense (depending on the abstraction in the descriptions) *complete*.

4.1 A short review of the six steps.

We aim to perform resolution on the state descriptions. Therefore, these descriptions should at least contain the information retained for the previous application: the instantiation patterns of the goals within the state, functors within the arguments of those goals and bindings between their variables.

These leave little space for further abstraction. As shown in the example description for the Sieve of Eratosthenos program, what we make abstraction of is the *number* of goals that occur in chains of identically instantiated goals with identical bindings linking them together.

As an example, the state description in the Sieve of Eratosthenos program contains two such chains, both with base block

Filter(G, V_i, V_{i+1}),

which is linked to a previous block Filter(G, V_{i-1}, V_i) through its second argument and to the next Filter(G, V_{i+1}, V_{i+2}) through its third argument.

In order to deal with general states, a state description usually will contain chains, of which the base blocks reveal a much more complicated structure than the ones in the example. Often base blocks are sets of abstract goals (with connecting bindings) themselves. In some cases they even take the form of chains. We do not discuss these technical observations in more detail, but simply stress the fact that abstraction must be made of the frequency with which repeating structures occur within the state.

As for most applications, representing the abstraction can be performed in many different ways. The unification mechanism and the behavior for builtin predicates can be dealt with similarly as in the previous section. We do not discuss them in further detail.

In the interpreter, special care must be taken of the selection of subgoals and the generation of new descriptions. Both must be dealt with during each resolution step. A resolution step now consists of

- selecting a appropriate goal from the description - using the new computation rule -,
- expanding or solving the abstract goal using the mechanism of section 3,
- applying the resulting abstract substitution to the entire state description
- building a new description that represents the newly obtained state. This may include a case study, distinguishing between the presence of one or more base blocks in a chain. It also includes verifying whether or not newly created goals can be combined with existing goals to obtain new chains in the description.

After each resolution step, the obtained state description must be compared to all previously obtained descriptions (not just to the ancestors). If its canonical form is equal to that of a previous description, then no further resolution for that branch of the abstract trace tree is needed.

5. Discussion.

The main objective of this paper was to present an informal introduction to the topic of abstract interpretation with the aim of reducing the effort which is required from a novice in the field in order to be able to develop his application. Using an example from the field of source-to-source program transformation we illustrated the main activities involved in building such applications. The solution to the problem was split up into two different layers, with increasing difficulty, such that the combination of the two is both correct and complete. We deliberately did not provide rigorous proofs of the termination nor of the correctness of the method, since we are convinced that they would decrease the comprehensibility of the paper. For more details on these, within a general framework on abstract interpretation, we refer to [6]. In the more specific framework of program transformation, the topic is dealt with in [16].

We conclude by discussing the applicability of abstract interpretation as a basic tool for different kinds of program transformation. The motivation is simple: most program transformation methods start off with an analysis of the runtime behavior of the input program to detect which inefficiencies may occur. This is obviously the case for techniques for loop-detection and -avoidance (e.g. [2], [25]) as well as for those dealing with the detection and elimination of redundant computations [14]. Here, abstract interpretation will provide an automated way to detect these inefficiencies, together with sufficient information on the computational history to perform the transformation which is adequate.

Next, there are the methods based on the synthesis of programs from traces of example computations. Research in this field has been initiated by A. Biermann in [2]. Using abstract interpretation to construct the traces, significantly increases the power of these methods, because

1. more computational paths can be covered with a single (abstract) top level query,
2. completeness of the synthesized program can be ensured in an automated way.

Both these advantages where illustrated in our example application. [4] and [13] describe a technique of the above type aiming at the conversion of any generate and test program into a more efficient - but logically equivalent - one, involving a coroutining control regime. V.Turchin proposes a similar technique for functional languages in [28], majorly aiming at a further automation of the *Unfold/Fold* method.

A promising extension of the work presented in [4], again based on synthesizing example traces, is the conversion of generate and test algorithms into equivalent forward checking algorithms. Here, it will not be sufficient to abstractly interpret a given program following a new computation rule as a basis for synthesis, but minor transformations on the logical component of the program will be needed as well. P.Van Hentenryck in [29] has thoroughly studied the relationship between these two control mechanisms and in our further work we aim to develop a transformation scheme on the basis of his results.

The *Unfold/Fold* technique of Burstall and Darlington [7] offer another application of abstract interpretation. In [19] S.Gregory illustrates through his many examples that the mechanism of unfolding (abstract expansion) and folding (recognition of fixed points within an abstract execution trace) are strongly related to abstract interpretation. In a more recent paper [11], J.Darlington recasts the original ideas on *Unfold/Fold* within a setting based on symbolic execution.

Closely related to code optimization is the automatic generation of mode declarations using abstract interpretation. This source level transformation method has been successfully developed and implemented at our research centre as an auxiliary result of the interpretation described in the previous sections.

A final, but certainly not less important application domain is situated in the field of partial evaluation. Practical use of partial evaluation as an independent technique in program transformation has lost of its credibility in the last years, because no halting condition for the expansion of recursive clauses can be formulated. A possible solution to the problem is the use of automatically generated *wait*-declarations, as proposed by L.Naish in [23]. However, the range of applicability for this technique has not clearly been established. More convincing is the idea to derive partially evaluated programs for given top level query patterns, where abstract interpretation guides the evaluation of recursive clauses. This idea was elaborated by J.Gallagher and M.Codish in [16] for the case of program specialization. This paper recasts the theoretical framework of abstract interpretation within the field of program transformation in a rigorous manner and is therefore highly recommended as further reading. Possibly, the synthesis of the specialized program can be slightly improved using the flowchart analysis technique described in [3]. At this time, several other researchers in the field (e.g. H.Fujita [15]) are exploring the possibilities of combined partial evaluation and abstract interpretation. We are convinced that these efforts will lead to a further breakthrough of both.

6. Acknowledgement.

We are indebted to G.Janssens, A.Callebout, B.Demoen and A.Marien for communications on their early work in the field. Also thanks to B. Mignon for his implementation of a mode declaration generator. D.De Schreye is supported by the Belgian I.W.O.N.L.-I.R.S.I.A. under contract number 4856. M.Bruynooghe is supported as research associate by the Belgian National Fund for Scientific Research.

References.

[1] Biermann A., On the inference of Turing machines from sample computations, Artificial Intelligence, Vol. 3, 1972.

[2] Brough D.R., Walker A., Some practical properties of logic programming interpreters, in Proc. FGCS conference, 1984, pp. 149-156.

[3] Bruynooghe M., De Schreye D. and Krekels B., Compiling Control, Proc.Third International Symposium on Logic Programming, 1986, pp. 70-78.

[4] Bruynooghe M., De Schreye D. and Krekels B., Compiling Control, J.Logic Programming, to appear.

[5] Bruynooghe M., Janssens G., Callebout A., Demoen B., Abstract interpretation: towards the global optimisation of Prolog programs, Proc. Fourth International Symposium on Logic Programming, 1987.

[6] Bruynooghe M., A framework for the abstract interpretation of logic programs, report CW62, 1987, K.U.Leuven.

[7] Burstall R.M. and Darlington J., A transformation system for developing recursive programs, JACM, 24, 1977, pp. 44-67.

[8] Clark K.I., McCabe F.G., Gregory S., IC-Prolog language features, Logic programming, ed. Clark/Tarnlund, 1982, pp. 254-266.

[9] Colmerauer A., Prolog II, manuel de reference et modele theoretique, Marseille, 1982.

[10] Cousot P., Cousot R., Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints, in Proc. 4th ACM POPL symposium, 1977, pp. 238-252.

[11] Darlington J., Pull H., A program development methodology based on a unified approach to execution and transformation, in Proc.workshop on partial evaluation and mixed computation, 1987, Denmark.

[12] Debray S.K., Warren D.S., Automatic mode inferencing for Prolog programs, Proc.Third International Symposium on Logic Programming, 1986, pp. 78-88.

[13] De Schreye D., Bruynooghe M. On the transformation of logic programs with instantiation based computation rules, J.Symbolic Computation, to appear.

[14] Fronhofer B., Double work as a reason for inefficiency of programs, Technical report T.U.M. Munchen, 1987.

[15] Fujita H., Abstract interpretation and partial evaluation of prolog programs, ICOT technical report, 1986.

[16] Gallagher J., Codish M., Specialisation of Prolog and FCP programs using abstract interpretation, in Proceedings of the workshop on partial evaluation and mixed computation, 1987, Denmark.

[17] Gallaire H. and Laserre C., A control meta language for logic programming, in *LogicProgramming*, eds. Clark L. and Tarnlund S.A., Academic Press, 1982, pp. 173-185.

[18] Genesereth M.R. and Ginsberg M.L., Logic Programming, CACM 28(9), Sept. 1985, pp. 933-941.

[19] Gregory S., Towards the compilation of annotated logic programs, Res.Report DOC80/16, June 1980, Imperial College.

[20] Jones N.D., Sondergaard H., A semantics based framework for the abstract interpretation of Prolog, in Abstract interpretation of declarative languages, eds. Abramsky S. and Hankin C. , Ellis Horwood, in print.

[21] Kanamari T., Kawamura T., Analyzing succes patterns of Logic programs by abstract hybrid interpretation, ICOT technical report, TR 279, 1987.

[22] Mellish C.S., Abstract interpretation of prolog programs, in Proc. 3rd International Conference on Logic Programming, 1986, LNCS 225, Springer-Verlag, 1986, pp. 463-474.

[23] Naish L., Automating control for logic programs, J. Logic Programming 2, 1985, pp. 167-183.

[24] Narain S., A technique for doing lazy evaluation in Logic, J.Logic Programming 3 (3), 1986, pp. 259-276.

[25] Pelhat S., Analysis and control of recursivity in Prolog programs, Technical report CRIL, Universite de Paris-sud, 1987.

[26] Pereira L.M., Logic control with logic, in Implementations of Prolog, ed. Cambell, Ellis, Horwood, 1984, pp.177-193.

[27] Sato T., Tamaki H., Transformational logic program synthesis, FGCS '84, Tokyo, 1984.

[28] Turchin V.F., The concept of a supercompiler, ACM Transactions on Programming Languages and Systems 8 (3), 1986, pp. 292-325.

[29] Van Hentenryck P., Consistency techniques in logic programming, Ph.D. thesis FUNDF, Namur, Belgium, 1987.

[30] Warren D.H.D., Coroutining facilities for Prolog, implemented in Prolog, DAI working paper, Edinburgh, 1979.

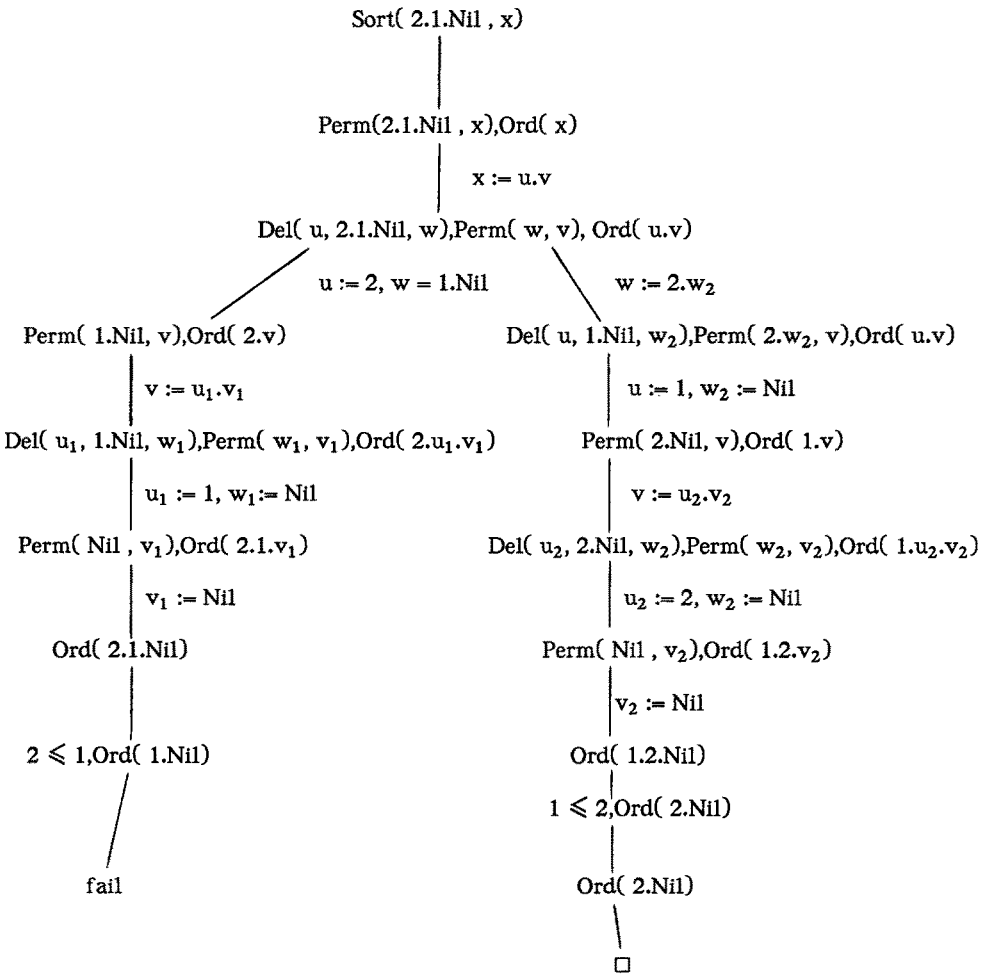

Fig. 1
Proof tree for ← Sort(2.1.Nil) under the standard computation rule of Prolog.

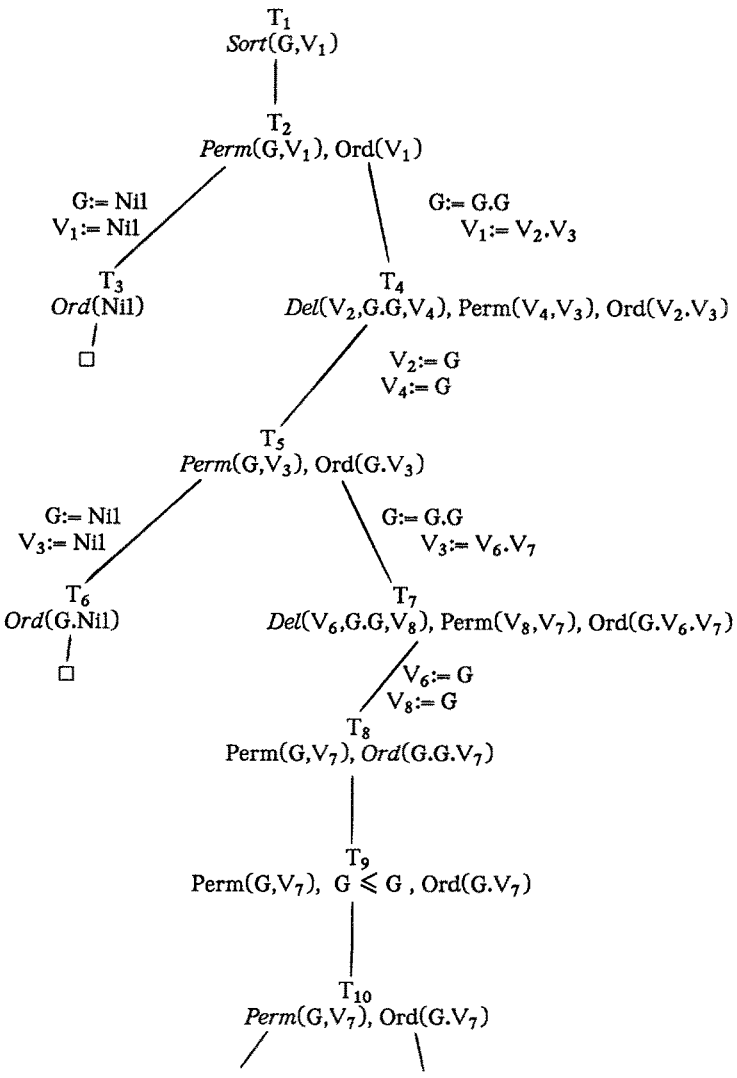

Fig. 2
Abstract trace tree of Slowsort
The selected subgoal is in Italic (e.g.*Perm*(G,V_1)).

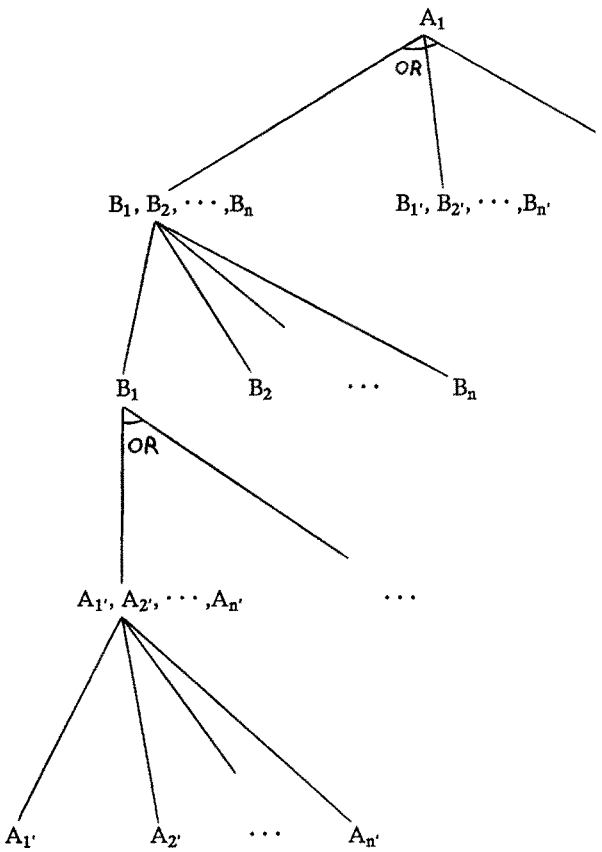

Fig.3 Abstract AND-OR-tree.