A PRACTICAL FRAMEWORK FOR THE ABSTRACT INTERPRETATION OF LOGIC PROGRAMS

MAURICE BRUYNOOGHE*

▷ There are numerous papers concerned with the compile-time derivation of certain run-time properties of logic programs, e.g. mode inferencing, type checking, type synthesis, and properties relevant for AND-parallel execution. Most approaches have little in common, they are developed in an ad hoc way, and their correctness is not always obvious. We develop a general framework which is suited to develop complex applications and to prove their correctness. All states which are possible at run time can be represented by an infinite set of proof trees (AND trees, SLD derivations). The core idea of our approach is to represent this infinite set of AND trees by a finite abstract AND-OR graph. We present a generic abstract interpretation procedure for the construction of such an abstract AND-OR graph and formulate conditions which allow us to construct a correct one in finite time. ◁

1. INTRODUCTION

Declarative semantics is one of the biggest assets of logic programming. It gives programs a meaning independent of any execution mechanism. However, the procedural semantics is also important. It helps the programmer to determine for which class of queries a procedure terminates and for which class the computation is sufficiently efficient. Procedural information helps the implementer to optimize the generated code by specializing it for the set of calls which can really occur at run time. Mode information is the best known example of procedural information. However, the information concerning possible procedure calls is not available to the implementer; it is in most cases an unacceptable burden for the programmer to provide it, and moreover, there would still be the problem of verifying the asserted statements.

*Supported as research associate by the Belgian National Fund for Scientific Research.
Address correspondence to Maurice Bruynooghe, Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3030 Heverlee, Belgium.
Received May 1988; accepted February 1989.

THE JOURNAL OF LOGIC PROGRAMMING

©Elsevier Science Publishing Co., Inc., 1991
655 Avenue of the Americas, New York, NY 10010 0743-1066/91/$3.50

So the relevant information about possible procedure calls must be collected or at least verified at compile time by means of a global analysis of the program. In recent years, numerous papers have appeared describing some kind of global analysis, for example mode inferencing, type checking, type synthesis, the collection of information allowing for compile-time garbage collection, and the collection of information relevant for the AND-parallel execution of programs. A lot of the work is in an ad hoc setting, the analysis is geared towards the application on hand, and its correctness is not always obvious. Recently, there have been several attempts to develop a more systematic approach based on abstract interpretation [1, 4, 10, 11, 15]. All these works are heavily influenced by the seminal work of Cousot and Cousot [7] on the abstract interpretation of programs in imperative languages. This paper is another such attempt. It contains the material described in [2], but substantially reorganized; irrelevant details obscuring the main theme have been removed, and the notion of correctness is more precise.

The purpose of abstract interpretation is to collect properties about the run-time behavior of programs, usually properties about the set of calls which are possible for each predicate. Obviously, which calls can occur at run time depends on the computation rule. So when performing abstract interpretation, one has to settle for a specific computation rule. Throughout the paper, we consider the top-down execution of logic programs under the depth-first left-to-right computation rule of PROLOG. Adaptation of the framework to other computation rules seems feasible. We ignore the search rule and the effects of cuts, so there might be calls which are considered possible but will never occur at run time. Also the undecidability of the halting problem contributes to this: a procedure call occurring at the right of a call which never succeeds is never executed at run time. According to the halting problem, some calls which never succeed cannot be recognized at compile time.

Now, the problem can be formulated as follows: given a program and a set of queries for a predicate Q (or, because such a set is often infinite, the properties of interest of the set of queries), determine the (properties of interest of the) set of calls which can occur at run time for each predicate of the program. The problem is solved, at least in principle, if one knows the set of proof trees (AND trees, SLD derivations) which can occur at run time. We say "in principle" because the set is usually infinite and even individual proof trees can be infinite (proof trees with a loop).

We define the concept of an abstract AND-OR graph, which represents a set of AND trees. We give a procedure to construct such an abstract AND-OR graph starting from a given set of queries and give conditions which are sufficient to guarantee that the constructed abstract AND-OR graph represents all AND trees which can appear at run time while executing a query in the given set. We show how to obtain a finite abstract AND-OR graph and give conditions which are sufficient to guarantee termination of the construction procedure.

In Section 2, we recall the concept of AND tree and develop the concept of an abstract AND-OR tree. We give the abstract AND-OR tree a semantics by defining a mapping from an abstract AND-OR tree into a set of AND trees. To facilitate the formulation of the semantics, we also introduce two intermediate structures: the generalized AND tree and the abstract AND tree.

Our framework for abstract interpretation is developed in Section 3. Our purpose is to develop a procedure to construct a correct and precise abstract

AND-OR tree (Section 3.1). In Section 3.2, we introduce the building blocks: procedure entry, procedure exit, and abstract interpretation of built-ins; also we formulate and prove conditions for their correctness. Because recursive procedures cause the abstract AND-OR tree to be infinite, we introduce in Section 3.3 the abstract AND-OR graph, which can be kept finite. Conditions for termination of the construction procedure are developed in Section 3.4. The results are summarized in Section 3.5.

As an example, mode inferencing is developed and proved correct for a simple class of programs in Section 4.

In Section 5 we conclude with a discussion of our achievements and of related work.

2. PRELIMINARIES

2.1. *The Concrete* AND *Tree*

Notation. We use σ, θ, and τ for (concrete) substitutions. A substitution is a set $\{V_1 \leftarrow t_1, \ldots, V_n \leftarrow t_n\}$ with the V_i distinct variables and the t_i terms. We sometimes say t_i is the *value* of V_i in the substitution. The set $\{V_1, \ldots, V_n\}$ is the *domain* of the substitution; the *range* is the set of variables appearing in $t_1, \ldots, t_n$. We can restrict our attention to idempotent substitutions, i.e. substitutions for which the intersection of domain and range is empty [12]. By $\mathrm{var}(t)$ we denote the set of variables occurring in t, by $\mathrm{var}(\sigma)$ the union of the domain and range of σ. Composition of two substitutions σ and θ is denoted by $\sigma\theta$.

Let us recall the basics of the top-down execution of logic programs under the standard (PROLOG) computation rule. At each stage, the current goal statement can be represented by $\leftarrow (A_1, \ldots, A_n)\sigma_i$, where $\sigma_i = \theta_1 \cdots \theta_i$ is the composition of the substitutions applied so far (the *accumulated* substitution). For the initial goal statement (the query) we have $\sigma_0 = \epsilon$, the empty substitution.

To perform a logical inference, the computation rule selects the leftmost subgoal $A_1\sigma_i$. The search rule selects a clause $H \leftarrow B_1, \ldots, B_m$, renames it, and unifies the head H with $A_1\sigma_i$. If the unification is successful with mgu θ_{i+1}, then the goal statement $\leftarrow (B_1, \ldots, B_m, A_2, \ldots, A_n)\sigma_{i+1}$ is derived with $\sigma_{i+1} = \sigma_i\theta_{i+1}$. (Notice that $B_j\sigma_i = B_j$ and thus $B_j\sigma_{i+1} = B_j\theta_{i+1}$ due to the renaming.) The new state is the *immediate successor* of the previous one; the old state is the *immediate predecessor* of the new one. Because a subgoal can match different clause heads, a goal statement can have several immediate successors. Transitive closure yields the successor and predecessor relations. A *sequence* $\ldots t_i, t_j, \ldots$ of AND trees is such that t_j is the immediate successor of t_i.

It is common to represent goal statements as proof trees (we call them AND trees in the sequel). The calls in the goal statement are the leaves of the tree. The AND tree also shows the origin of the calls. A possible representation is shown in Figure 1, where a set of successive AND trees is shown. Notice that we also include the clause heads in the AND tree. Later on, this will turn out to be convenient. In each AND tree, the accumulated substitution $\sigma_i = \theta_1 \cdots \theta_i$ appears everywhere in the tree. This is reflected in PROLOG implementations by the fact that each

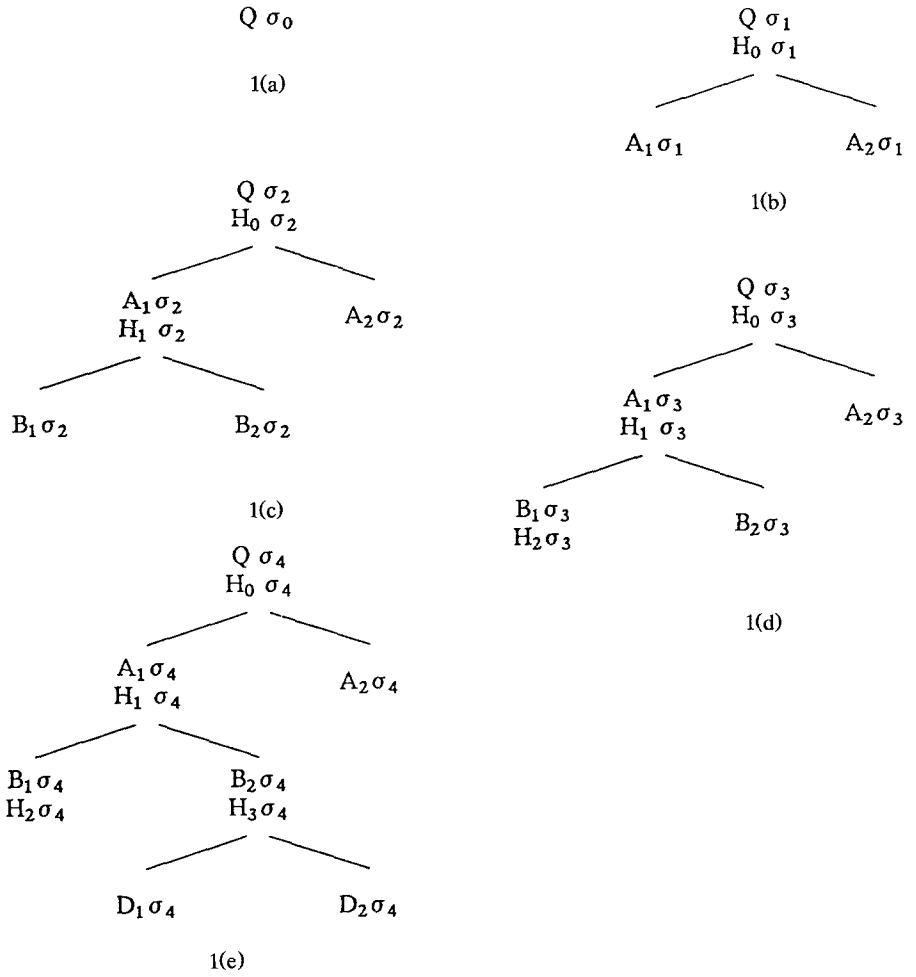


FIGURE 1. Some successive AND trees constructed while solving a query $\leftarrow Q$. (a) Initial query $\leftarrow Q$ ($\sigma_0 = \epsilon$). (b) The query is solved with a clause $C_0 = H_0 \leftarrow A_1, A_2$; the mgu of $Q\sigma_0$ and H_0 is θ_1, $\sigma_1 = \sigma_0\theta_1 = \theta_1$. (c) The call $A_1\sigma_1$ is solved with a clause $C_1 = H \leftarrow B_1, B_2$; the mgu of $A_1\sigma_1$ and H_1 is θ_2, $\sigma_2 = \sigma_1\theta_2$. (d) $B_1\sigma_2$ unifies with the fact $C_2 = H_2$; the mgu is θ_3, $\sigma_3 = \sigma_2\theta_3$. (e) $B_2\sigma_3$ is solved with the clause $C_3 = H_3 \leftarrow D_1, D_2$; the mgu of $B_2\sigma_3$ and H_3 is θ_4, $\sigma_4 = \sigma_3\theta_4$.

variable has a unique associated memory cell where one can access its current value. Knowing the computation rule, one can derive the shape of all predecessors of an AND tree, so, except for the intermediate substitutions, an AND tree represents an SLD derivation [12].

The set of all AND trees which can originate from a given set of queries specifies completely the procedural behavior of a program for that set of queries. Indeed, each AND tree shows precisely the instance of the call to be executed in the next inference step. For example, in Figure 1(e) the call to be executed is D_1; the instance is $D_1\sigma_4$. In addition, some AND trees also show the instance of a call

immediately after its successful completion: for example, in Figure 1(d), the call B_1 is completed. The instance of the call immediately after its completion is $B_1\sigma_3$.

It is convenient to assume that queries consist of a single call as in Figure 1; this does not incur any loss of generality.

2.2. *The Generalized* AND *Tree*

An AND tree contains more information than needed for our purpose; it shows the whole state of the computation. To characterize the procedural behavior, it suffices to know the instance of procedure calls immediately before their execution (and immediately after their completion). This suggests representing a sequence of successive AND trees by a *generalized* AND *tree.*

Definition.

A tree with a call Q as simple node, with that call adorned on the left with a substitution τ, and with the domain of τ a subset of var(Q) is an (initial) generalized AND tree. τ is called the *call substitution* of Q. [Figure 2(a) shows an example.]

Given a generalized AND tree with a call P as leaf such that P is adorned on the left with the call substitution τ_i and $H \leftarrow B_1, \ldots, B_n$ is a properly renamed clause defining P. A new generalized AND tree is obtained by pairing P with H and adding the calls $B_1, \ldots, B_n$ as children of P; B_1 is adorned on the left with a call substitution τ_{i+1}. The domain of τ_{i+1} is a subset of var($H \leftarrow B_1, \ldots, B_n$). [Figure 2(b), (c), and (f) are examples for $n > 0$.] With $n = 0$, τ_{i+1} adorns an empty body [Figure 2(d)] and is also the *success substitution* of the body. The operation extending the tree is named *procedure entry.*

Given a generalized AND tree with a call P as leaf such that P is adorned on the left with the call substitution τ_i and P is a built-in. A new generalized AND tree is obtained by adorning P on the right with a substitution τ_{i+1}. The domain of τ_{i+1} is a subset of var(clause containing P). τ_{i+1} is called the success substitution of P. With P the last call of its clause, τ_{i+1} is also the success substitution of the body; otherwise it is the call substitution of the next call. The operation extending the tree is named *interpretation of a built-in.*

Given a generalized AND tree with a call P adorned on the left with a call substitution but not adorned with a substitution on the right, such that P is the father of a clause body with success substitution τ_i. A new generalized AND tree is obtained by adorning P on the right with a substitution τ_{i+1}. The domain of τ_{i+1} is a subset of var(clause containing P). τ_{i+1} is called the success substitution of P. With P the last call of its clause (the query), τ_{i+1} is also the success substitution of the body (the query); otherwise it is the call substitution of the next call. The operation extending the tree is named *procedure exit.* [Figure 2(e) is an example.]

This definition is constructive and allows us to use the notions of *immediate successor*, *immediate predecessor*, and *sequence* for generalized AND trees as we did for AND trees.

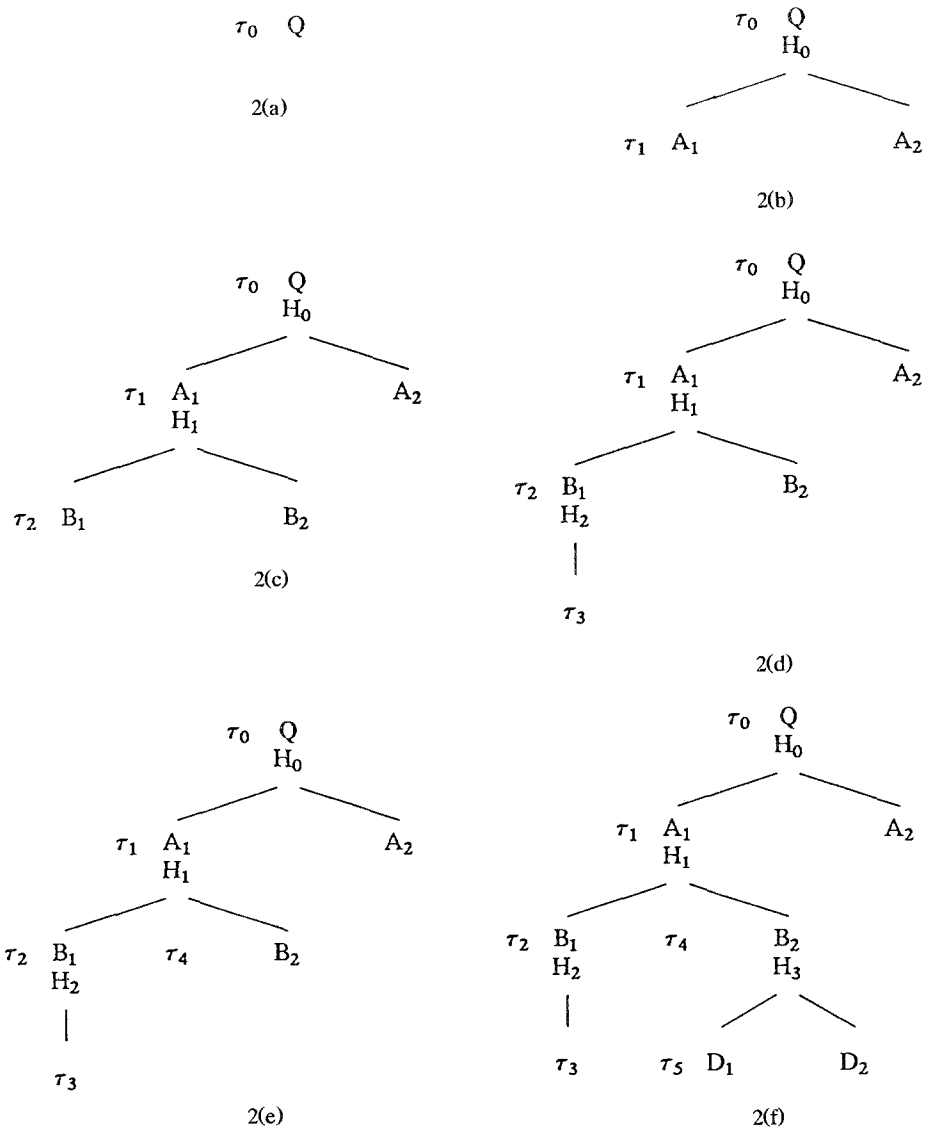


FIGURE 2. Generalized AND trees. Each tree is an extension of the previous one. (a) Initial tree; $Q\tau_0$ is the instance of the call. (b) Procedure entry; the clause $C_0 = H_0 \leftarrow A_1, A_2$ is used. τ_1 is computed; $A_1\tau_1$ is the instance of the call to be executed. (c) Procedure entry; $B_1\tau_2$ is the next call. (d) Procedure entry: τ_3 is computed; the body of the used clause is empty. (e) Procedure exit: τ_4 is computed; $B_1\tau_4$ is the instance of the completed call; $B_2\tau_4$ the instance of the next call. (f) Procedure entry: τ_5 is computed.

There is a strong similarity between a sequence of AND trees and a sequence of generalized AND trees (e.g. Figures 1 and 2). At each point, only one of the three extension operations is possible, and the construction mimics the computation rule. However, notice that the definition does not require the substitutions τ_i to be related to the substitutions occurring during execution of a query. In particular, the execution can fail at a certain point, while the generalized AND tree can be further extended. Of course the idea is that a useful generalized AND tree captures the relevant procedural information present in a sequence of AND trees. This is the case when applying a call substitution to the call it adorns yields the instance of the call immediately before its execution and applying a success substitution to the call it adorns yields the instance of the call immediately after its successful completion. To achieve this, it suffices that the substitutions appearing in the generalized AND tree are restrictions of the corresponding accumulating substitutions—for example, in Figures 1 and 2, when τ_0 is the restriction of σ_0, τ_1 of $\sigma_1, \ldots$. Notice that τ_3 and τ_4 are different restrictions of the same accumulating substitution σ_3.

One might wonder why we impose the requirement on the domain of substitutions in the generalized AND tree. Allowing accumulated substitutions would result in domains with an unbounded number of variables. Our requirement avoids this. Also, it becomes easy to compare substitutions adorning different instances of the same clause, something which will be essential for the treatment of recursive calls. There is a price to be paid: procedure exit must recover a different restriction of the same accumulating substitution.

So, given a sequence of AND trees, it is straightforward to obtain a generalized AND tree corresponding to it. For example, with the above-indicated choices for the τ_i, Figures 2(f) corresponds to the sequence of AND trees shown in Figure 1. We say it *represents* the sequence, because it conveys the important information regarding the instances of the calls. However, it will become necessary to work the other way round: Given a generalized AND tree (which can be considered as being constructed by a sequence of operations), which set of (successive) AND trees does it represent? This is the purpose of the following definition.

Definition. The set of AND trees represented by a generalized AND tree consists of the sequence of trees defined below and their renamings:

Base case: The first subtree in the sequence of subtrees resulting in the given generalized AND tree consists of the root only: a call Q and its call substitution τ_0. It represents the initial AND tree for the query $Q\tau_0$. This initial AND tree is called the *active* AND *tree* of the initial generalized AND tree. For example, the AND tree of Figure 1(a) corresponds to and is represented by the generalized AND tree of Figure 2(a) when $Q\sigma_0 = Q\tau_0$, i.e. when the variables of var(Q) have the same value in σ_0 and τ_0.

Recursion: The set of AND trees represented by a generalized AND tree consists of (1) the set of AND trees represented by its immediate predecessor, (2) its *active* AND *tree* (if existing). It remains to define the active AND tree of a generalized AND tree.

Case 1: The generalized AND tree is derived from its predecessor by the procedure-entry operation. If (1) the predecessor has an active AND tree

representing the goal statement $\leftarrow (A_1, \ldots, A_n)\sigma_i$, (2) the procedure-entry operation has applied a clause $H \leftarrow B_1, \ldots, B_m$ and has computed a substitution τ, (3) using the same clause to solve $A_1\sigma_i$ results in a new AND tree representing the goal statement $\leftarrow (B_1, \ldots, B_m, A_2, \ldots, A_n)\sigma_j$, (4) $(H \leftarrow B_1, \ldots, B_m)\tau = (H \leftarrow B_1, \ldots, B_m)\sigma_j$ (the variables of the clause have the same value in τ and σ_j), then the new AND tree is the active AND tree. There is no active AND tree if the four conditions are not satisfied. For example, Figure 1(c) represents the active AND tree of the generalized AND tree of Figure 2(c) iff $(H_1 \leftarrow B_1, B_2)\tau_2 = (H_1 \leftarrow B_1, B_2)\sigma_2$, and Figure 1(b) represents the active AND tree of the generalized AND tree of Figure 2(b).

Case 2: The generalized AND tree is derived from its predecessor by the interpretation of a built-in B_i which is part of a clause $H \leftarrow B_1, \ldots, B_m$. If (1) the predecessor has an active AND tree with σ_i as accumulated substitution and B_i as call to be executed, (2) the success substitution of B_i in the generalized AND tree is τ, (3) solving $B_i\sigma_i$ succeeds and yields a new AND tree with accumulating substitution σ_j, (4) $(H \leftarrow B_1, \ldots, B_m)\tau = (H \leftarrow B_1, \ldots, B_m)\sigma_j$, then the new AND tree is the active AND tree of the generalized AND tree.

Case 3: The generalized AND tree is derived from its predecessor by the procedure-exit operation. The procedure-exit operation computes the success substitution τ of a call B_i. This call is part of a clause $H \leftarrow B_1, \ldots, B_m$ (or is the initial query). If (1) the predecessor of the generalized AND tree has an active AND tree containing the accumulated substitution σ, (2) $(H \leftarrow B_1, \ldots, B_m)\tau = (H \leftarrow B_1, \ldots, B_m)\sigma$, then this same AND tree is also the active AND tree of the given generalized AND tree. For example, Figure 1(d) represents the active AND tree of the generalized AND tree of Figure 2(e) iff $(H_1 \leftarrow B_1, B_2)\tau_4 = (H_1 \leftarrow B_1, B_2)\sigma_3$, and Figure 1(d) represents the active AND tree of the generalized AND tree of Figure 2(d).

A generalized AND tree *representing a sequence* of AND trees is a kind of visualization of the trace used by Mellish in [15].

Lemma. Given a generalized AND *tree having an active* AND *tree. A clause which is properly renamed to be used for a procedure-entry operation on the generalized* AND *tree is also properly renamed to be used for the corresponding procedure call in the active* AND *tree.*

PROOF. Variables occurring in the active AND tree also occur in the generalized AND tree. This property holds for the initial AND tree consisting of $Q\sigma_0$. Indeed, because $Q\sigma_0 = Q\tau_0$, the variables of $Q\sigma_0$ occur in the initial generalized AND tree. New variables introduced by procedure calls in the AND tree originate from the applied clause. So, by induction, the properly renamed clause used for procedure entry in the generalized clause is also properly renamed to be used for the corresponding procedure call in the active AND tree. □

2.3. *Abstract* AND-OR *Tree*

Let us consider a set of queries $Q\sigma_1, \ldots, Q\sigma_n$ instead of a single query. Some of the generalized AND trees will differ only in the substitutions they contain. We can

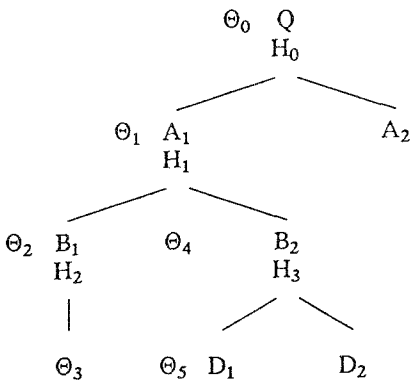


FIGURE 3. Abstract AND tree. An abstract AND tree represents a set of generalized AND trees.

represent such AND trees at once by replacing single substitutions with sets of substitutions. We call the resulting tree an *abstract* AND *tree*. We use the word *abstract* in anticipation of the fact that the sets will often be infinite and that we can only represent them by making abstraction of certain properties.

Notation. We use the symbol Θ to represent a set of substitutions, and $\perp$ to represent the empty set. $t\Theta$ represents the set $\{t\theta \mid \theta \in \Theta\}$.

An example of an abstract AND tree is shown in Figure 3. An abstract AND tree represents a set of generalized AND trees. An element of the set is obtained by selecting a substitution from every set of substitutions. All notions of generalized AND trees can be carried over: successor, predecessor, and sequence remain, as well as the operations procedure entry and procedure exit; interpretation of built-ins is renamed into abstract interpretation of built-ins; call and success substitution are renamed into abstract call and abstract success substitution; and the application order of the operations still mimics the computation rule. Finally, the active AND tree becomes a *set* of active AND trees. Members of the set are active AND trees of generalized AND trees represented by the abstract AND tree. Notice that every generalized AND tree represented by an abstract AND tree represents at least one concrete AND tree (a query) but does not necessarily have an active AND tree (the selected substitutions can be incompatible).

In a final generalization, we cope with the nondeterminism. A predicate can be defined by several clauses. So we introduce the *abstract* AND-OR *tree* as a generalization of the abstract AND tree. We associate an OR node with each call as shown in Figure 4. There can be a branch for every clause defining the predicate of the call. An abstract AND-OR tree represents a set of abstract AND trees; an element of the set is obtained by selecting one branch at each OR node. For example, the branches drawn in Figure 4 correspond to the abstract AND tree of Figure 3. The notions of abstract call and abstract success substitution can be carried over. However, extending a given abstract AND-OR tree becomes nondeterminate in absence of a search rule. Indeed, operations can be possible at different OR branches.

The notion of a set of active AND trees can be carried over; however, members of the set no longer have the same shape: different clauses can have been used to

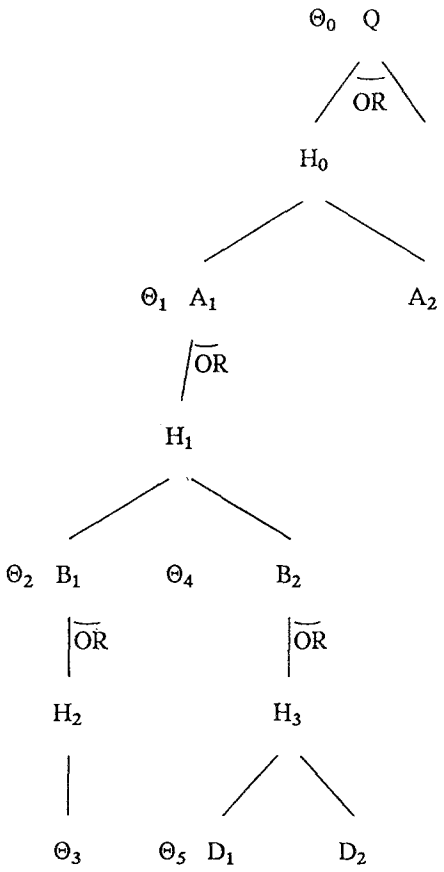


FIGURE 4. Part of an abstract AND-OR tree. An abstract AND-OR tree represents a set of abstract AND trees.

construct different members of the set. We can partition the set of active AND trees according to the call selected by the computation rule. Members in the same partition class have equivalent paths from root to selected call: the same sequence of calls $Q_0, \ldots, Q_n$; and for all i, the same clause is used to solve Q_i. This sequence of calls and clauses is also present in the abstract AND-OR tree, so we call such a partition class the *set of active* AND *trees of the node* Q_n. For example, with appropriate values for the Θ_i, the AND tree of Figure 1(e) corresponds to the node D_1 of the abstract AND-OR tree of Figure 4. (The path consists of the calls Q, A_1, B_2).

Other elements of the class could be obtained either by using another element of Θ_0 or by using other clauses for solving B_1.

By now, we have introduced abstract AND-OR trees, defined the set of AND trees they represent, and defined the set of active AND trees of a node which is ready for execution.

3. ABSTRACT INTERPRETATION

3.1. Purpose

Now, we can define the goal of abstract interpretation in the following concrete terms: Given a set of queries $Q\Theta$, construct an abstract AND-OR tree which is

correct. Correctness means it describes all concrete AND trees which can occur during execution of a query in $Q\Theta$.

The abstract AND-OR tree should also be as precise as possible, i.e., the sets of substitutions Θ_i should be as small as possible. We say "as small as possible" because in practice, the Θ_i are always overestimations of reality.

A correct (and sufficiently precise) abstract AND-OR tree contains all the information one can hope for about instances of calls immediately before their execution and immediately after their successful completion. In the sequel we use the term *call specification* for the set of terms in $P\Theta$ where P is a call and Θ its abstract call substitution. Similarly, $P\Theta$ is the *success specification* when Θ is the abstract success substitution of P.

3.2. *Construction of a Correct Abstract* AND-OR *Tree*

What we describe below is in fact a collecting interpretation. Usually, the sets Θ are infinite, so one has to use an abstraction of them. Again we use the word *abstract* in anticipation of this. Also, the procedure described below does not terminate for recursive programs: the point where the success specification of a recursive call is computed is never reached. We ignore this problem for the time being.

In this section we develop a procedure (abstract interpretation procedure) to construct an abstract AND-OR tree for a call specification $Q\Theta$ of a given set of queries. The procedure basically uses primitive operations: procedure entry, procedure exit, and abstract interpretation of built-ins. The definition of these primitives is application dependent; however, we can formulate conditions for them which are sufficient to guarantee that the final AND-OR tree is correct. So we can replace the global conditions for correctness—the final abstract AND-OR tree must represent all concrete AND trees—by conditions which are local to the primitive operations.

To prove the global correctness of the abstract AND-OR tree, we maintain an invariant at each stage of the construction.

Invariant. AND trees which can occur at run time either are represented by the (partial) abstract AND-OR tree or are successors of an *active* AND tree represented by the abstract AND-OR tree.

3.2.1. Initialization. Given a query with call specification $Q\Theta$, the abstract AND-OR tree is initialized with a root representing the call Q and having an abstract call substitution Θ. This abstract AND-OR tree represents all initial concrete AND trees for queries in the set $Q\Theta$. Obviously, this establishes the invariant.

3.2.2. Procedure Entry. Figure 5 illustrates the procedure-entry operation on a call B_i with abstract call substitution Θ. Assuming that B_i is defined by m clauses, the procedure-entry operation creates an OR node with m branches. Figure 5(b) shows the jth branch in detail. An AND node with n children is created, and Θ_1, the call substitution of D_1 is computed.

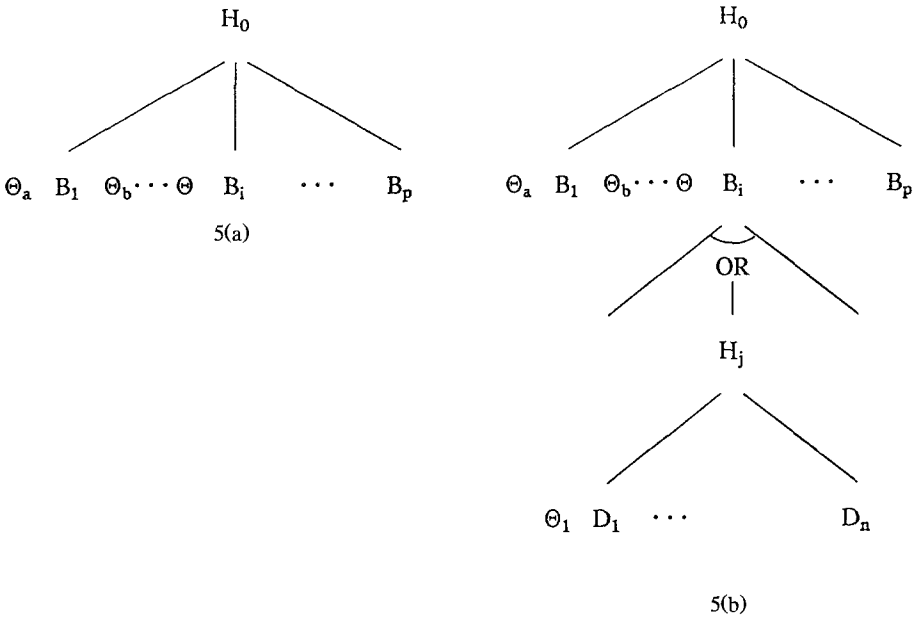


FIGURE 5. Procedure entry. (a) Part of an abstract AND-OR tree under construction. Call B_i with abstract call substitution Θ is ready for execution. (b) Effect of procedure-entry operation on call B_i: detail of jth branch. The abstract call substitution Θ_1 for the call D_1 is computed.

Proposition. A sufficient condition to maintain the invariant is the following:

if $\tau \in \Theta$, $B_i\tau$, *and* H_j *have mgu* θ_j
then $(H_j \leftarrow D_1, \ldots, D_n)\theta_j \in (H_j \leftarrow D_1, \ldots, D_n)\Theta_1$.

PROOF. The invariant holds for the situation of Figure 5(a), so AND trees which can appear at run time either are represented by the abstract AND-OR tree or are successors of active AND trees of the abstract AND-OR tree.

First, consider the active AND trees of nodes other than B_i. (Recall that an abstract AND-OR tree can have different calls ready for execution.) They are still active AND trees of the abstract AND-OR tree of Figure 5(b).

Remain the active AND trees of node B_i and their successors. Let Figure 6(a) be such an active AND tree. Because it is represented by the abstract AND-OR tree of Figure 5(a), there exists a generalized AND tree representing it. Let Figure 6(b) be that generalized AND tree, so $\tau \in \Theta$ and $B_i\sigma = B_i\tau$. Figure 6(c) shows an immediate successor of the AND tree of Figure 6(a). It is obtained by applying the clause $H_j \leftarrow D_1, \ldots, D_n$ on $B_i\sigma$. This AND tree is the active AND tree of a generalized AND tree which is an extension of Fiure 6(b) as shown in Figure 6(d) and which satisfies $(H_j \leftarrow D_1, \ldots, D_n)\tau_j = (H_j \leftarrow D_1, \ldots, D_n)\sigma_j$. The abstract AND-OR tree of Figure 5(b) represents such a generalized AND tree: one makes the same choices as one did to obtain the generalized AND tree of Figure 6(b), and selects the jth branch and a $\tau_j \in \Theta_1$ satisfying $(H_j \leftarrow D_1, \ldots, D_n)\tau_j = (H_j \leftarrow D_1, \ldots, D_n)\sigma_j$. Such a τ_j exists

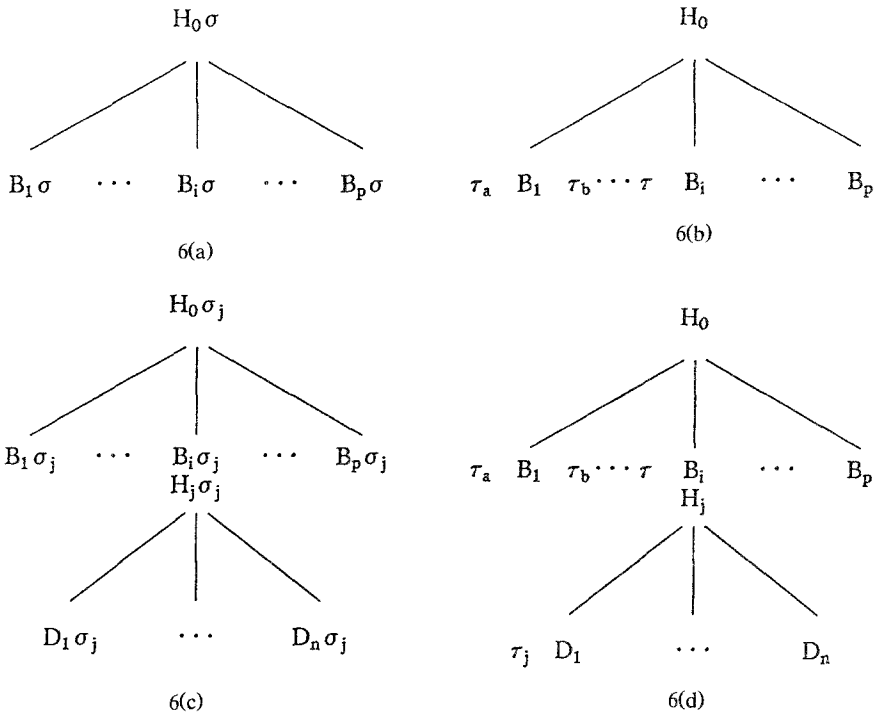


FIGURE 6. AND trees related to procedure entry. (a) Part of an active AND tree of node B_i of the abstract AND-OR tree of Figure 5(a). (b) Part of the generalized AND tree representing the AND tree of (a): $(H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\sigma = (H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\tau$. (c) Part of an AND tree which is a successor of the AND tree of (a). $\theta_j = \text{mgu}(B_i\sigma, H_j)$, $\sigma_j = \sigma\theta_j$. (d) Part of the generalized AND tree representing the AND tree of (c): $(H_j \leftarrow D_1, \ldots, D_i, \ldots, D_n)\tau_j = (H_j \leftarrow D_1, \ldots, D_i, \ldots, D_n)\sigma_j$.

because:

$\tau \in \Theta$ [the AND tree of Figure 6(a) is represented by the generalized AND tree of Figure 6(b), which in turn is represented by the abstract AND-OR tree of Figure 5(a)].

$B_i\sigma$ and H_j have mgu θ_j [Figure 6(c) is a successor of Figure 6(a)] and $B_i\sigma = B_i\tau$, so also $B_i\tau$ and H_j have mgu θ_j.

The premises of the sufficient condition are satisfied thus:

$$(H_j \leftarrow D_1, \ldots, D_n)\theta_j \in (H_j \leftarrow D_1, \ldots, D_n)\Theta_1;$$

in other words, Θ_1 contains a τ_j such that

$$(H_j \leftarrow D_1, \ldots, D_n)\theta_j = (H_j \leftarrow D_1, \ldots, D_n)\tau_j.$$

Now $(H_j \leftarrow D_1, \ldots, D_n)\sigma\theta_j = (H_j \leftarrow D_1, \ldots, D_n)\theta_j$. (The variables of the clause $H_j \leftarrow \cdots$ do not occur in σ.) So $(H_j \leftarrow D_1, \ldots, D_n)\tau_j = (H_j \leftarrow D_1, \ldots, D_n)\sigma_j$, and the generalized AND tree of Figure 6(d) represents the active AND tree of Figure 6(c).

This means that the active AND tree of Figure 6(a) is represented by the old abstract AND-OR tree [Figure 5(a)], and thus also by the new one [Figure 5(b)]. Its

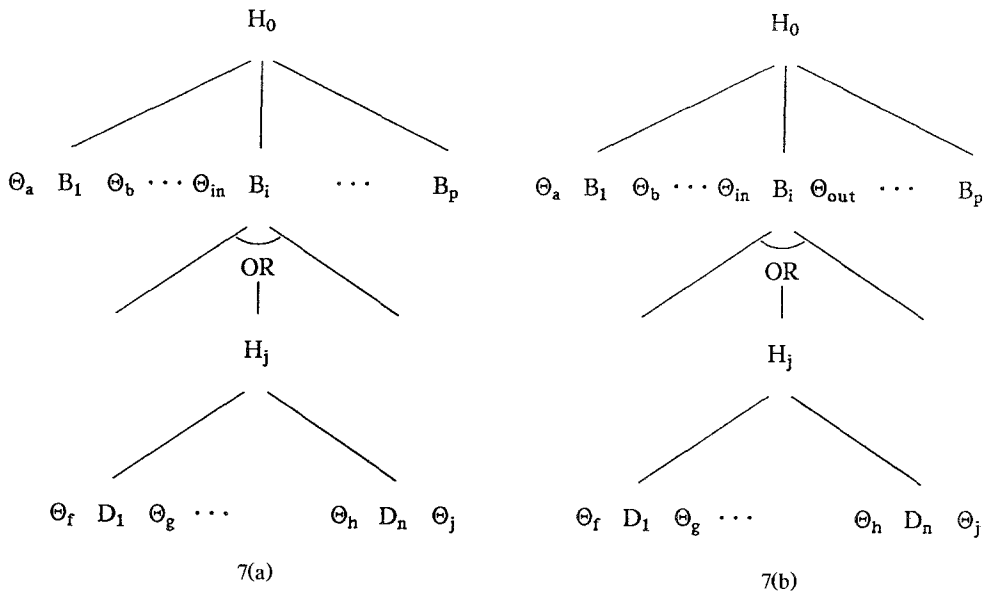


FIGURE 7. Procedure exit. (a) For each of the m clauses defining B_i, the final abstract substitution Θ_j has been obtained. (b) The procedure-exit operation computes Θ_{out}, the abstract success substitution of B_i.

immediate successors are active AND trees of the new abstract AND-OR tree, while the other successors are successors of these new active AND trees. Thus the invariant is maintained.

3.2.3. Procedure Exit. Figure 7 illustrates the procedure-exit operation. The abstract success substitution Θ_{out} of a call B_i can be computed when the m clauses defining the predicate B_i have been completely processed and the abstract success substitution Θ_j of the body of each defining clause has been obtained. Figure 7(a) shows the jth branch in detail. Θ_{out} is added in Figure 7(b).

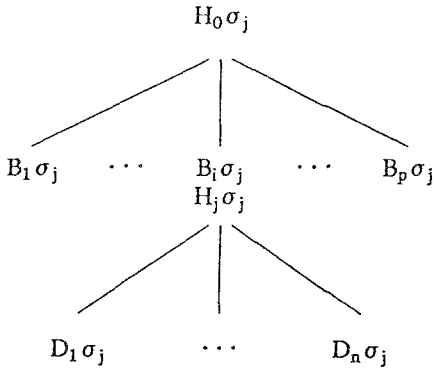
Proposition. A sufficient condition to maintain the invariant is the following: for all j in the range 1 *to m:*

if $\tau_j \in \Theta_j$ *and there exists a pair of substitutions* τ_{in}, θ *such that* $\tau_{in} \in \Theta_{in}$, $B_i\tau_{in}\theta = H_j\tau_j$, *and the only variables of* $(H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\tau_{in}$ *occurring in* θ *also occur in* $B_i\tau_{in}$ [$X \in \text{var}(\theta)$ *and* $X \in \text{var}((H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\tau_{in})$ $\rightarrow X \in \text{var}(B_i\tau_{in})$ *in words:* θ *can be the result of executing* B_i *with properly renamed clauses*],

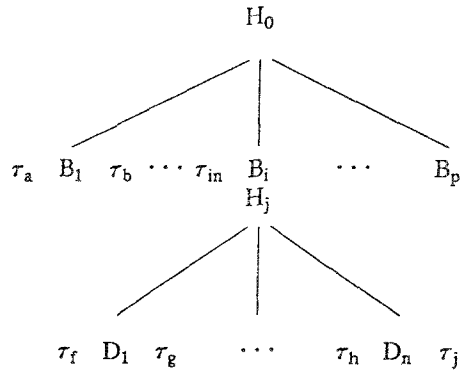
then $(H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\tau_{in}\theta \in (H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\Theta_{out}$.

The proof is very similar to the previous one and can be skipped if desired.

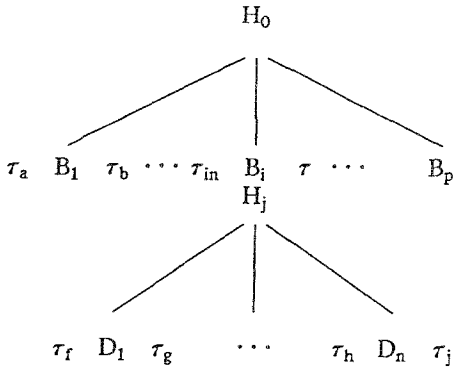
PROOF. The invariant holds for the situation of Figure 7(a). To maintain the invariant, it suffices that the abstract AND-OR tree of Figure 7(b) represent the same set of active AND trees as the abstract AND-OR tree of Figure 7(a).



8(a)



8(b)



8(c)

FIGURE 8. AND trees related to procedure exit. (a) Part of an active AND tree representing the state of the computation immediately after completing D_n. (b) Part of the generalized AND tree representing the active AND tree of (a): $(H_j \leftarrow D_1, \ldots, D_n)\sigma_j = (H_j \leftarrow D_1, \ldots, D_n)\tau_j$. (c) This generalized AND tree is an immediate successor of the tree in (b). It has the same active AND tree iff $(H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_n)\sigma_j = (H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_n)\tau$.

The only problem is with active AND trees as shown in Figure 8(a). The clause $H_j \leftarrow D_1, \ldots, D_n$ has been used to solve B_i, and the AND tree represents the state of the computation immediately after completing B_i. Because of the invariant, there exists a generalized AND tree representing the active AND tree of Figure 8(a) and represented by the abstract AND-OR tree of Figure 7(a). Let Figure 8(b) be that generalized AND tree, so $\tau_j \in \Theta_j$. Figure 8(c) shows a successor of the generalized AND tree of Figure 8(b). Its active AND tree is also the tree shown in Figure 8(a) iff $(H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\sigma_j = (H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\tau$.

The abstract AND-OR tree of Figure 7(b) represents such a generalized AND tree: one makes the same choices as one did to obtain the generalized AND tree of Figure 8(b) and selects $\tau \in \Theta_{out}$ such that $(H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\sigma_j = (H_0 \leftarrow$

$B_1, \ldots, B_i, \ldots, B_p)\tau$. Such a τ exists because:

σ_j can be written as $\sigma_{\text{in}}\theta$ with σ_{in} the accumulated substitution immediately before the execution of B_i.

Because Figure 8(b) is a generalized AND tree represented by the abstract AND-OR tree of Figure 7(a) and because Figure 8(a) is the active AND tree of that generalized AND tree and the invariant holds for the abstract AND-OR tree of Figure 7(a), we have

$$\tau_j \in \Theta_j, \qquad \tau_{\text{in}} \in \Theta_{\text{in}},$$

and

$$(H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\sigma_{\text{in}} = (H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\tau_{\text{in}}.$$

$B_i\tau_{\text{in}}\theta = B_i\sigma_{\text{in}}\theta = B_i\sigma_j = H_j\sigma_j = H_j\tau_j$.

θ is the accumulated substitution due to the execution of $B_i\sigma_{\text{in}}$, so it only contains variables occurring in $B_i\sigma_{\text{in}}$ or in the clauses used to solve $B_i\sigma_{\text{in}}$. In other words, it does not contain variables occurring in $(H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\sigma_{\text{in}}$ other than those of $B_i\sigma_{\text{in}}$.

So τ_j, τ_{in}, and θ satisfy the premises of the condition; thus

$$(H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\tau_{\text{in}}\theta \in (H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\Theta_{\text{out}}.$$

Thus Θ_{out} contains a substitution τ such that

$$(H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\tau = (H_0 \leftarrow B_1, \ldots, B_i, \ldots, B_p)\tau_{\text{in}}\theta.$$

$(H_0 \leftarrow B_1, \ldots, B_p)\tau_{\text{in}} = (H_0 \leftarrow B_1, \ldots, B_p)\sigma_{\text{in}}$; thus $(H_0 \leftarrow B_1, \ldots, B_p)\tau_{\text{in}}\theta = (H_0 \leftarrow B_1, \ldots, B_p)\sigma_{\text{in}}\theta$, because θ does not contain other variables of $(H_0 \leftarrow B_1, \ldots, B_p)\sigma_{\text{in}}$ than those occurring in $B_i\sigma_{\text{in}}$ and $B_i\sigma_{\text{in}}\theta = B_i\tau_{\text{in}}\theta$.

$\sigma_{\text{in}}\theta = \sigma_j$; thus $(H_0 \leftarrow B_1, \ldots, B_p)\tau = (H_0 \leftarrow B_1, \ldots, B_p)\sigma_j$ and τ is the substitution we were looking for.

In other words, the condition is sufficient to guarantee that the new abstract AND-OR tree represents the same active AND trees as its predecessor, so the invariant is maintained. □

3.2.4. Abstract Interpretation of Built-ins. Certain predicates are not defined by a set of clauses but are primitives of the execution mechanism. To construct the abstract AND-OR tree, one must be able to compute their abstract success substitution given their abstract call substitution. Again, it is an application problem to do so, but we can formulate a condition which is sufficient to guarantee that the invariant is maintained.

Figure 9 illustrates the situation; B_i is a built-in with abstract call substitution Θ_{in}.

Proposition. A sufficient condition to maintain the invariant is the following:

if $\tau_{\text{in}} \in \Theta_{\text{in}}$ *and executing* $B_i\tau_{\text{in}}$ *yields* θ,

then $(H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\tau_{\text{in}}\theta \in (H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\Theta_{\text{out}}$.

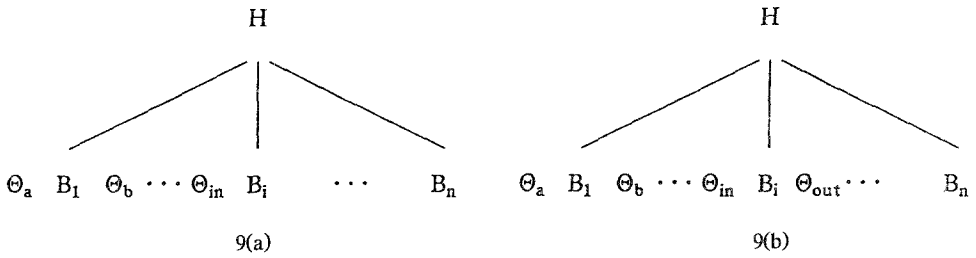
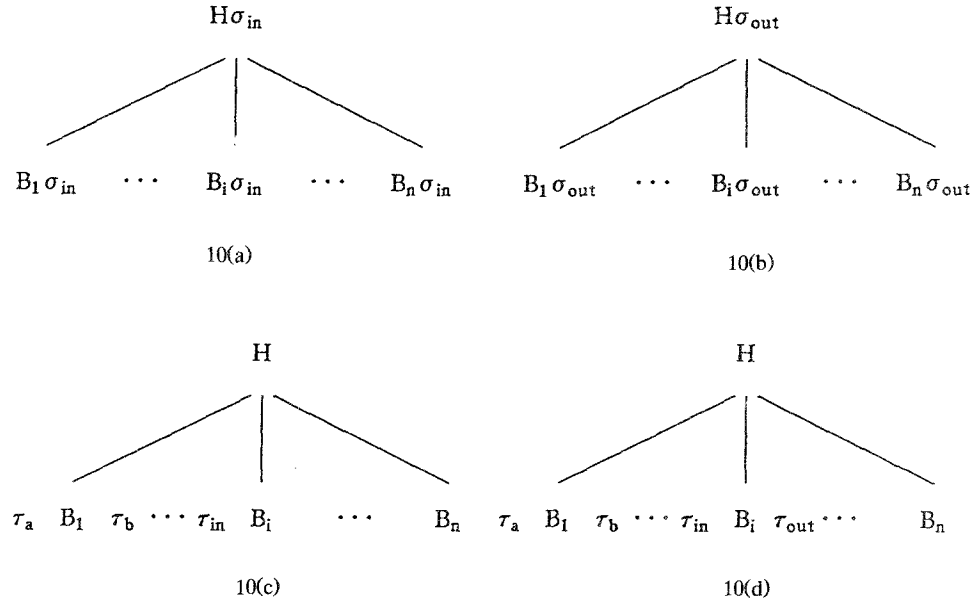


FIGURE 9. Abstract interpretation of built-in. (a) B_i is a built-in with abstract call substitution Θ_{in}. (b) The abstract success substitution Θ_{out} is computed.

Again, the proof is similar and can be skipped if desired.

PROOF. The invariant holds for the situation shown in Figure 9(a). The active AND trees represented by Figure 9(a) are the same as those represented by Figure 9(b) except for those of node B_i. Figure 10(a) shows such an active AND tree. Because of the invariant, the abstract AND-OR tree of Figure 9(a) represents a generalized AND tree representing the active AND tree. Let Figure 10(c) show this generalized AND tree: $(H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\sigma_{in} = (H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\tau_{in}$. Figure 10(b) shows the immediate successor of the AND tree of Figure 10(a). Figure 10(d) shows a successor of Figure 10(c), and its active AND tree is the AND tree shown in Figure 10(b) iff $(H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\sigma_{out} = (H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\tau_{out}$. The abstract

FIGURE 10. AND trees related to execution of built-in. (a) Part of an active AND tree of node B_i. (b) Executing $B_i\sigma_{in}$ yields θ; $\sigma_{out} = \sigma_{in}\theta$. (c) Part of the generalized AND tree representing the active AND tree of (a). (d) This generalized AND tree is the successor of the one shown in (c).



AND-OR tree of Figure 9(b) represents such a generalized AND tree: one makes the same choices as one did to obtain the generalized AND tree shown in Figure 10(c) and selects $\tau_{out} \in \Theta_{out}$ such that

$$(H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\tau_{out} = (H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\sigma_{in}\theta.$$

Such a τ_{out} exists because:

$\tau_{in} \in \Theta_{in}$ and $(H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\tau_{in} = (H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\sigma_{in}$. The abstract AND-OR tree of Figure 9(a) represents the generalized AND tree of Figure 10(c); this generalizes AND tree represents the active AND tree of Figure 10(a).

Executing $B_i\sigma_{in} = B_i\tau_{in}$ yields θ.

The premises of the sufficient condition are satisfied; thus $(H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\tau_{in}\theta \in (H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\Theta_{out}$; thus Θ_{out} contains a τ_{out} such that

$$(H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\tau_{out} = (H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\tau_{in}\theta.$$

θ is the result of executing $B_i\sigma_{in} = B_i\tau_{in}$, so it can only instantiate variables of $B_i\sigma_{in}$; thus

$$(H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\sigma_{in}\theta = (H \leftarrow B_1, \ldots, B_i, \ldots, B_n)\tau_{in}\theta.$$

Thus τ_{out} is the substitution we are looking for.

So the active AND tree of Figure 10(a) is represented by the old abstract AND-OR tree and thus also by the new one; its immediate successor(s) is an active AND tree of the new abstract AND-OR tree, while the other successors are successors of the new active AND tree. Thus the invariant is maintained. □

The idea underlying the different conditions is to replace the global requirement "the set of substitutions in a point must contain the restriction of the accumulated substitution of every computation reaching that point" by local conditions which are much easier to prove.

3.3. Towards a Finite Representation

Our procedure, as it stands, is quite useless. For recursive programs, it attempts to construct an infinite tree, obviously a nonterminating activity. Even worse, it never comes to the point where it can compute the abstract success substitution of a recursive call.

Another problem is the representation of the sets of substitutions Θ which can be infinite. The latter problem is solved by abstraction. We use abstract substitutions instead of substitutions. They are finite expressions in some language (an example is in Section 4). They are given a semantics by means of a concretization function γ. γ maps an abstract substitution into a set of substitutions. Using the symbol β to represent an abstract substitution, we have $\gamma(\beta) = \Theta$. An abstract substitution has a domain: the set of variables the elements of Θ can replace. So, for $\theta \in \gamma(\beta)$, $\mathrm{dom}(\theta) \subseteq \mathrm{dom}(\beta)$. $\perp$ denotes the empty abstract substitution: $\gamma(\perp) = \varnothing$. The meaning of γ can be extended: $\gamma(t\beta) = \{t\theta \mid \theta \in \gamma(\beta)\}$. We can define an

order relation on abstract substitutions: $\beta_1 \geq \beta_2$ iff $\gamma(\beta_1) \supseteq \gamma(\beta_2)$, $\beta_1 \equiv \beta_2$ iff $\beta_1 \leq \beta_2$ and $\beta_2 \leq \beta_1$; and an upper bound with the property $\beta = \text{up}(\beta_1, \beta_2) \rightarrow \gamma(\beta) \supseteq \gamma(\beta_1) \cup \gamma(\beta_2)$.

To tackle the main problem, we will use an idea close to the notion of rational trees as introduced in the logic-programming community by Colmerauer [6]. Although a rational tree can be infinite, it has a finite graph representation because it has only a finite number of distinct subtrees.

Using the same idea, we can also represent an infinite AND-OR tree by a finite graph if the tree contains, up to renaming, only a finite number of distinct subtrees. However, usually the number of distinct subtrees will be infinite. That means there is at least one branch containing an infinite number of calls to some predicate P. Moreover, there are an infinite number of distinct subtrees rooted at these calls P. As there is an OR branch for every clause defining P, all these trees have the same shape. The difference is in the abstract substitutions. Replacing an abstract substitution by a larger one cannot reduce the represented set of AND trees. However, it can make finite the number of distinct subtrees on a branch.

This idea is used in the procedure described below for the construction of a finite graph which represents a *rational* abstract AND-OR tree. After describing the algorithm, we will show that it is correct (on terminating, it represents all possible AND trees) and we will develop conditions for termination of the procedure.

3.4. *Abstract Interpretation Procedure*

3.4.1. Initialization. Given $Q\Theta$ representing a set of queries. The abstract AND-OR graph is initialized with a root representing the call Q and having an abstract call substitution β, such that $\gamma(Q\beta) \supseteq Q\Theta$. This is followed by the *abstract interpretation of the call Q with abstract call substitution β.*

3.4.2. Procedure Entry Operation. Applied to a call P with abstract call substitution β_{in}, procedure entry extends the abstract AND-OR graph as shown in Figure 11(a) and computes $\beta_1^{\text{in}}, \ldots, \beta_m^{\text{in}}$. β_j^{in} is a function of $P\beta_{\text{in}}$ and the clause $H_j \leftarrow B_1^j, \ldots, B_{j_n}^j$; its domain is $\text{var}(H_j \leftarrow B_1^j, \ldots, B_{j_n}^j)$.

A sufficient condition for correctness is: If $\tau \in \gamma(\beta_{\text{in}})$, and $P\tau$ and H_j have mgu θ, then

$$\left(H_j \leftarrow B_1^j, \ldots, B_{j_n}^j\right)\theta \in \gamma\left(\left(H_j \leftarrow B_1^j, \ldots, B_{j_n}^j\right)\beta_j^{\text{in}}\right).$$

3.4.3. Procedure Exit Operation. Having processed the bodies of the clauses defining P, procedure exit can be applied. It is convenient to compute β_{out} in two steps [Figure 11(b)]. First an abstract substitution β with the variables of P as domain is computed from $H_1, \beta_1^{\text{out}}, \ldots, H_m, \beta_m^{\text{out}}$. Finally, β_{out} is a function (the *extension*) of β and β_{in}; its domain is $\text{var}(H \leftarrow B_1, \ldots, P, \ldots, B_n)$.

A sufficient condition for correctness is (for all j):

if $\tau_j \in \gamma(\beta_j^{\text{out}})$ and there exists a pair of substitutions τ_{in}, θ such that $\tau_{\text{in}} \in \gamma(\beta_{\text{in}})$ and $P\tau_{\text{in}}\theta = H_j\tau_j$ and the only variables of $(H \leftarrow B_1, \ldots, P, \ldots, B_n)\tau_{\text{in}}$ occurring in θ also occur in $P\tau_{\text{in}}$,

then $(H \leftarrow B_1, \ldots, P, \ldots, B_n)\tau_{\text{in}}\theta \in \gamma((H \leftarrow B_1, \ldots, P, \ldots, B_n)\beta_{\text{out}})$.

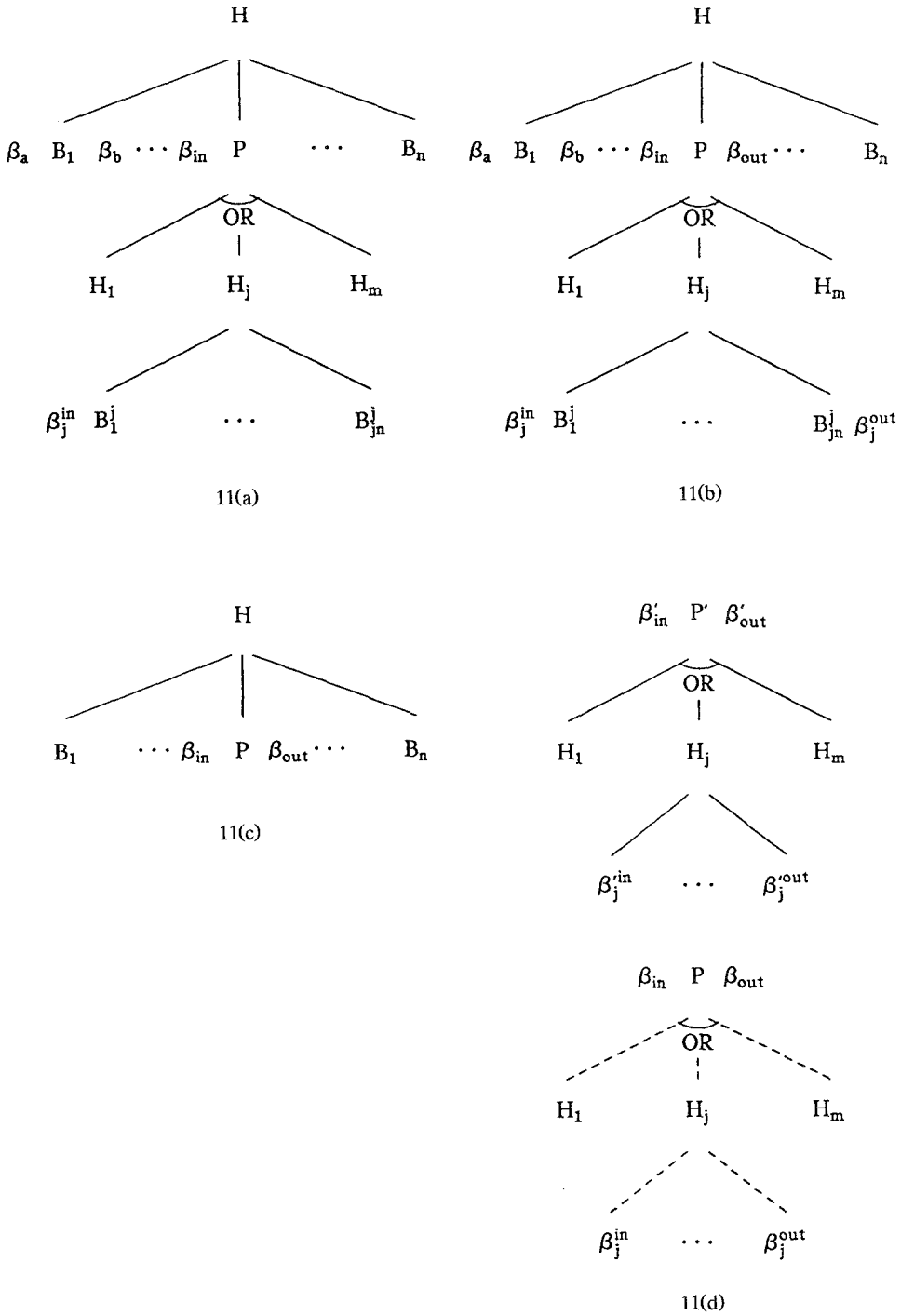


FIGURE 11. The primitive steps of the abstract interpretation procedure. (a) Procedure entry extends the abstract AND-OR graph and computes $\beta_1^{in}, \ldots, \beta_m^{in}$. (b) The procedure exit computes β_{out}. (c) Abstract interpretation of built-in. (d) Recursion. β_{out} is computed from β_{in} and renamings of $\beta_j'^{out}$.

When proving the correctness of applications, it is convenient to use the intermediate abstract substitution β over variables of P (corresponding to the input of the extension) and to replace the condition by the following two conditions:

I. *If* $\tau_j \in \gamma(\beta_j^{\text{out}})$ and there exists a pair of substitutions τ_{in}, θ such that $\tau_{\text{in}} \in \gamma(\beta_{\text{in}})$ and $P\tau_{\text{in}}\theta = H_j\tau_j$ and the only variables of $(H \leftarrow B_1, \ldots, P, \ldots, B_n)\tau_{\text{in}}$ occurring in θ also occur in $P\tau_{\text{in}}$,
then there exists a $\tau \in \gamma(\beta)$ such that $P\tau = P\tau_{\text{in}}\theta$.

II. If $\tau_{\text{in}} \in \gamma(\beta_{\text{in}})$ and there exists a substitution θ such that $P\tau_{\text{in}}\theta \in \gamma(P\beta)$ and the only variables of $(H \leftarrow B_1, \ldots, P, \ldots, B_n)\tau_{\text{in}}$ occurring in θ also occur in $P\tau_{\text{in}}$,
then $(H \leftarrow B_1, \ldots, P, \ldots, B_n)\tau_{\text{in}}\theta \in \gamma((H \leftarrow B_1, \ldots, P, \ldots, B_n)\beta_{\text{out}})$.

These two conditions imply the original one.

3.4.4. Abstract Interpretation of Built-ins. Applied to a built-in P with abstract call substitution β_{in}, abstract interpretation of the built-in computes the success-substitution β_{out} [Figure 11(c)]. Again, it is convenient to distinguish the intermediate substitution β with the variables of P as domain.

A sufficient condition for correctness is:

If $\tau_{\text{in}} \in \gamma(\beta_{\text{in}})$ and executing P yields θ
then $(H \leftarrow B_1, \ldots, P, \ldots, B_n)\tau_{\text{in}}\theta \in \gamma((H \leftarrow B_1, \ldots, P, \ldots, B_n)\beta_{\text{out}})$.

Also this condition is implied by the following two (the last one is the same as for procedure exit):

I. *If* $\tau_{\text{in}} \in \gamma(\beta_{\text{in}})$ and executing $P\tau_{\text{in}}$ yields θ,
then $P\tau_{\text{in}}\theta \in \gamma(P\beta)$.

II. *If* $\tau_{\text{in}} \in \gamma(\beta_{\text{in}})$ and there exists a substitution θ such that $P\tau_{\text{in}}\theta \in \gamma(P\beta)$ and the only variables of $(H \leftarrow B_1, \ldots, P, \ldots, B_n)\tau_{\text{in}}$ occurring in θ also occur in $P\tau_{\text{in}}$ (i.e., θ can be the result of executing $P\tau_{\text{in}}$),
then $(H \leftarrow B_1, \ldots, P, \ldots, B_n)\tau_{\text{in}}\theta \in \gamma((H \leftarrow B_1, \ldots, P, \ldots, B_n)\beta_{\text{out}})$.

3.4.5. Abstract Interpretation of Call P with Abstract Call Substitution β_{in}. To obtain β_{out}, the abstract success-substitution of P with the same domain as β_{in}, we distinguish four cases:

Case 1. P is a built-in. Apply abstract interpretation (Section 3.4.4).

Case 2. P has no ancestor node with a call for the same predicate.

Apply procedure entry (Section 3.4.2).

Apply abstract interpretation (Section 3.4.5) on the calls of the created bodies, always working on a call with known abstract call substitution.

Apply procedure exit (Section 3.4.3).

Case 3. P has an ancestor node with a call P' to the same predicate [Figure 11(d)]. Applying procedure entry yields β_j^{in}. For all j, it is the case that $\beta_j^{\text{in}} \equiv \beta_j'^{\text{in}}$

(up to renaming) with $\beta_j'^{\text{in}}$ the result of applying procedure entry on P' [Figure 11(d)].

Procedure exit is used to obtain a value for β_{out}; as value for β_j^{out} one uses the renaming of $\beta_j'^{\text{out}}$ [Figure 11(d)] or, if that value is not computable, $\perp$.

This finishes the call P. However, at some point, all values $\beta_j'^{\text{out}}$ become available. If, for some j, it is not the case that $\beta_j'^{\text{out}} \leq \beta_j^{\text{out}}$, then one starts over with the procedure-exit operation computing β_{out}. As new value for β_j^{out} one uses $\text{up}(\beta_j'^{\text{out}}, \beta_j^{\text{out}})$.

The node P refers back to the ancestor P'.

Case 4. P has an ancestor with a call P' to the same predicate, but case 3 does not apply. If the depth of recursion is not beyond a predefined threshold, then proceed as in case 2. Otherwise, apply procedure entry. If $\beta_j^{\text{in}} \leq \beta_j'^{\text{in}}$ for all j, then proceed as in case 3.

Otherwise, for those j such that $\beta_j^{\text{in}} \leq \beta_j'^{\text{in}}$ does not hold, recompute the jth subtree of P' using $\text{up}(\beta_j'^{\text{in}}, \beta_j^{\text{in}})$ as $\beta_j'^{\text{in}}$.

3.5. *Partial Correctness*

Assuming termination of the procedure, we obtain a finite abstract AND-OR graph which represents a rational abstract AND-OR tree. This tree usually is infinite for recursive programs. The desired property is that it represents any AND tree arising during execution of a query. More formally, we have to prove:

Proposition. The rational abstract AND-OR *tree represented by the finite abstract* AND-OR *graph as constructed for* $Q\beta_0$ *by the procedure of Section* 3.4 *represents all concrete* AND *trees that can arise during the execution of a query* $Q\sigma_0 \in \gamma(Q\beta_0)$.

Notice that this result is stronger than for the (partial) abstract AND-OR trees constructable by the procedure of Section 3.2. There, all possible concrete AND trees were proved either to be represented or to be a successor of a represented active AND tree. The stronger result is due to the fact that the rational abstract AND-OR tree represents for any given concrete AND tree an abstract AND tree with the same shape. This abstract AND tree is obtained by selecting the appropriate branch at each OR node.

PROOF. For the proof, we consider the sequence of concrete AND trees leading to the given one and the corresponding sequence of generalized AND trees. The proof is by induction on the latter sequence.

Base. The initialization of the abstract AND-OR (Section 3.4.1) assures that the initial generalized AND tree, which has the initial AND tree as active AND tree, is represented.

Step. Given a generalized AND tree GT and its active AND tree AT, which are represented by the abstract AND-OR tree. We have to show that the successor of GT, SGT, is also represented by the abstract AND-OR tree. We distinguish three cases.

Procedure entry. SGT is derived from GT by applying procedure entry on a call P, using a clause $H_j \leftarrow B_1^j, \ldots$. Assume β_{in} and β_j^{in} are the abstract call substitutions of P and B_1^j, respectively, in the abstract AND-OR tree. Inspecting the procedure of Section 3.4 reveals:

In case 2 of Section 3.4.5 [Figure 11(a)], procedure entry is used to obtain β_j^{in}.

In case 4 of Section 3.4.5 [Figure 11(d)] the value of $\beta_j'^{in}$ can be enlarged.

In cases 3 and 4 [Figure 11(d)], the call P in the abstract AND-OR graph refers back to P'. In the infinite abstract AND-OR tree, the value of β_j^{in} is a renaming of the value of $\beta_j'^{in}$. However, this renaming is identical to (case 3) or at least as large as (case 4) the value one would obtain by applying procedure entry.

So, in all cases, β_j^{in} is at least as large as the value obtained by applying procedure entry on the call P with abstract call substitution β_{in}. GT is represented, so the correctness of procedure entry assures that SGT is also represented by the abstract AND-OR tree. The active AND tree of SGT is the successor of AT.

Interpretation of built-in. SGT is derived from GT by interpretation of a built-in P. Assume β_{in} and β_{out} are respectively the abstract call substitution and abstract success substitution of P in the abstract AND-OR tree. β_{out} is derived from β_{in} by abstract interpretation of built-in P [Section 3.4.4, Figure 11(c)]. GT is represented, so the correctness of abstract interpretation of the built-in assures that SGT is also represented by the abstract AND-OR tree. The active AND tree of SGT is the successor of AT.

Procedure exit. SGT is derived from GT by applying procedure exit on a call P. P is solved with the clause $H_j \leftarrow B_1^j, \ldots$. Assume β_{in} is the abstract call substitution of P in the abstract AND-OR tree. Assume β_j^{out} and β_{out} are the abstract success substitutions of $B_{j_n}^j$ and P respectively, in the abstract AND-OR tree. Inspecting the procedure of Section 3.4 reveals:

In case 2 of Section 3.4.5 [Figure 11(b)], procedure exit is applied, using β_j^{out} and the values of the other branches to obtain β_{out}.

In case 3 [Figure 11(d)], the call P in the abstract AND-OR graph refers back to P'. In the abstract AND-OR tree, the value of β_j^{out} is a renaming of the value of $\beta_j'^{out}$. According to case 3, procedure exit has been used to obtain β_{out} using a value which is at least as large as the renaming of $\beta_j'^{out}$. The correctness of procedure exit assures that β_{out} is at least as large as the value one would obtain by applying procedure exit on the values β_j^{out} which are in the infinite AND-OR tree. (The set $H_j\tau_j$ is at least as large.)

So GT is represented, and in both cases the value for β_{out} is at least as large as the value one would obtain by applying procedure exit on the values β_j^{out} which are in the infinite AND-OR tree. Thus SGT is also represented by the abstract AND-OR tree because of the correctness of procedure exit. SGT and GT have the same active AND tree. □

3.6. Termination

Whether the procedure terminates is determined by the behavior of the iterations in case 3 and case 4 of Section 3.4.5. In case 3, each iteration starts, at least for one j, with β_j^{out} strictly larger than in the previous iteration. In case 4, each iteration starts, at least for one j, with $\beta_j'^{\text{in}}$ strictly larger than in the previous iteration. $\gamma(\beta)$ is a set of substitutions; their domains are subsets of the variables occurring in the clause β annotates. So, for the iterations to terminate, it suffices that there be an abstract substitution $\top$ such that $\gamma(\top) = \{$all substitutions whose domain is a subset of the variables in the annotated clause$\}$ and that $\top$ be reached after a finite number of iterations. This is the case if the set of possible abstract substitutions for β_j^{out} ($\beta_j'^{\text{in}}$) (all having the same domain) is a partially ordered set without ascending chains (without infinite sequences $\alpha_1 < \alpha_2 \ldots$). Notice that we have not to impose restrictions on abstract substitutions adorning other calls of the abstract AND-OR graph.

In the sequel, the set of possible values for an abstract substitution is called the abstract domain.

3.7. Remarks

(1) When developing an application, one has to devise a language for abstract substitutions which is rich enough to express the properties of interest. On the other hand, the language must be restricted enough to be able

to define correct realizations of procedure entry, procedure exit, and abstract interpretation of built-ins;

to enforce the ascending chain condition on the abstract domain, at least for the points in the abstract AND-OR graph which require it;

to define $\equiv$, $\leq$, and upper-bound operations.

For example, with a sufficiently restricted language, $\equiv$ can be realized by a test for syntactical identity.

(2) Many authors require that the abstract domain be a lattice and that the operations be monotone. As shown, this is not essential. The point is that the *collecting* operations working on sets of solutions are inherently monotone, while sets of substitutions form a lattice with intersection as glb and union as lub. As the abstract operations should not only be correct but also as precise as possible, a lattice structure for the abstract domain and monotonicity for the operations are valuable properties. They assure the existence of a most precise outcome for every operation.

(3) With monotonicity for all operations, the successive values for β_j^{out} and $\beta_j'^{\text{in}}$ (cases 3 and 4 in Section 3.4.5) are increasing sequences. Iteration terminates when two successive values are equivalent. Actually, there is an interesting variant for the computation of β_{out}. One starts with $\top$. Due to the monotonicity, one gets a decreasing sequence which terminates when two successive values are equivalent. Actually, one can stop at any point; indeed, any value in the sequence is correct. Also, one can start for β_{out} with an application dependent value for which one can prove that it initiates an increasing or a decreasing sequence (that the second value is either

greater or smaller). With monotonicity, the iterations are reminiscent of the fixpoint computations as used by many authors.

(4) One does not need to apply procedure entry on a call with abstract call substitution $\perp$. Indeed, $\perp$ is a correct (and the most precise) abstract success substitution.

(5) A call P with abstract call substitution β can appear in different branches of the abstract AND-OR graph. Of course, an implementation takes care not to compute the same subtree twice and uses for example an extension table [7] to obtain the correct abstract success substitution immediately.

(6) With the previous optimization, an abstract AND-OR graph can still contain several occurrences of the same clause (the calls having different abstract call substitutions). A compiler using this abstract AND-OR graph as input has the option to generate code for several specialized versions of the clause. Of course, care must be taken that each call uses the appropriate version. In case one definitely wants only one version, one can modify the abstract-interpretation procedure and decide not to expand a call P as soon as there is already an expanded call P' for that predicate: either one enlarges β_j^{in} until $\beta_j^{\text{in}} \equiv \beta_j'^{\text{in}}$, or one can redo P' with $\text{up}(\beta_j^{\text{in}}, \beta_j'^{\text{in}})$ as input for the procedure-entry operation. In this case, one can build the skeleton of the graph representation of the entire abstract AND-OR graph before computing the abstract substitutions. There is a close correspondence between this case and the Jones-Sondergaard approach [10].

(7) One can use a symbolic value for the abstract success substitution β_j^{out} [Figure 11(d)] of an unexpanded call. In some cases (typically tail recursion) one can obtain the new value for β_j^{out} without making any assumption on the initial concrete value of β_j^{out}. One obtains an inequality which can be solved with the method described in [17]. This avoids having to iterate several times. (The same holds for the iterations between $\beta_j'^{\text{in}}$ and β_j^{in} with the additional inequality γ(initial value for $\beta_j'^{\text{in}}$) $\subseteq \gamma$(symbolic value for $\beta_j'^{\text{in}}$).

(8) Consider clause bodies $b1 = P, Q, R$ and $b2 = P, Q, P, R$. In b2, the call P is repeated. Consider abstract interpretation of $b1$ and $b2$ just before abstract execution of R. Let β_1 be the abstract call substitution of R in $b1$, and β_2 the same for $\beta 2$. Every active AND tree for $b1$ has a corresponding active AND tree for $b2$ where the subtree rooted at P appears twice. (The reverse does not hold: it is possible to have an active AND tree for $b2$ where the two subtrees of P are different.) This implies that β_2 is a *correct* value for β_1 and validates a *repeat-previous-call* strategy: At any point in the abstract interpretation of a program where one has derived an abstract substitution β, one can repeat a previous call P (a call to the left of the program point containing β) with abstract call substitution β to obtain a new value for β (the subtree rooted at P is not modified). For example, P has an argument with principal functor f or g; Q shares the same argument and fails for the case with principal functor g. At this point, it can be useful to redo the abstract interpretation of P. Now the principal functor of the argument under consideration can only be f, and one can expect a more

precise result for the abstract substitution of the program point between Q and R. Of course, this approach has a computational cost, but it can be a worthwhile alternative for a refinement of the abstract domain.

4. MODE INFERENCING

We illustrate the application of our framework with a simple mode-inferencing system. The derived modes are not very precise; however, they are nontrivial. The application is developed for a restricted class of programs. A more sophisticated system, integrating mode and type inferencing, is described in [3].

4.1. Abstract Domain

An abstract substitution over a set $\{X_1, \ldots, X_n\}$ of variables is either $\perp$ or a set containing

an assignment of the form $X_i := m_i$ for each variable, where $m_i \in \{a, f, g\}$ (a, f, g are abbreviations for respectively *any*, *free*, and *ground*);

a number (≥ 0) of sets $\{X_i, X_j\}$ $(i \neq j)$.

The set $\{X_1, \ldots, X_n\}$ is the domain of the abstract substitution. It is convenient to represent in a concrete substitution a free variable by $X_i \leftarrow X_i$.

An idempotent $\theta = \{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\} \in \gamma(\beta)$ iff

(domain of θ) = (domain of β);

if $X_i := g \in \beta$ then t_i is a ground term;

if $X_i := f \in \beta$ then t_i is a variable;

if $X_i \leftarrow t_i$, $X_j \leftarrow t_j \in \theta$, and $\text{var}(t_i) \cap \text{var}(t_j) \neq \emptyset$, then $\{X_i, X_j\} \in \beta$.

The assignments give the mode of the variables; the sets $\{X_i, X_j\}$ express that X_i and X_j *possibly* share a variable. Of course, $\gamma(\perp) = \emptyset$.

We define an order relation $\leq$ between abstract substitutions over the same domain: $\beta_1 \leq \beta_2$ iff

if $X_i := f \in \beta_1$ then $X_i := f \in \beta_2$ or $X_i := a \in \beta_2$;

if $X_i := g \in \beta_1$ then $X_i := g \in \beta_2$ or $X_i := a \in \beta_2$;

if $X_i := a \in \beta_1$ then $X_i := a \in \beta_2$;

if $\{X_i, X_j\} \in \beta_1$ then $\{X_i, X_j\} \in \beta_2$.

In fact, we obtain a lattice structure: $\perp$ is the bottom element, and the top element is the abstract substitution containing $X_i := a$ for every variable in the domain and $\{X_i, X_j\}$ for every pair of variables in the domain.

The least upper bound of two abstract substitutions β_1 and β_2 is defined as follows:

if $\beta_1(\beta_2) = \perp$ then $\text{lub}(\beta_1, \beta_2) = \beta_2(\beta_1)$; else

the mode of X_i in $\text{lub}(\beta_1, \beta_2)$ is derived from the modes in β_1 and β_2 by the

following table:

	f	g	a
f	f	a	a
g	a	g	a
a	a	a	a

if $\{X_i, X_j\} \in \beta_1$ or $\{X_i, X_j\} \in \beta_2$ then $\{X_i, X_j\} \in \text{lub}(\beta_1, \beta_2)$.

The following properties are obvious:

$\beta_1 \leq \beta_2$ iff $\gamma(\beta_1) \subseteq \gamma(\beta_2)$.

$\gamma(\text{lub}(\beta_1, \beta_2)) \supseteq \gamma(\beta_1) \cup \gamma(\beta_2)$.

$\beta_1 \equiv \beta_2$ iff $\beta_1 = \beta_2$.

The lattice of abstract substitutions over the same domain is a finite lattice.

4.2. *Operations*

To keep the example simple, we assume procedure headings and calls (except built-ins) are in the form $P(X_1, \ldots, X_n)$ with all X_i distinct variables. By adding the built-ins $X = Y$ and $X = f(Y_1, \ldots, Y_n)$ ($X, Y, Y_1, \ldots, Y_n$ variables, f a functor), we retain the same expressivity.

4.2.1. Procedure Entry. Given a call $P(Y_1, \ldots, Y_n)$, an abstract substitution β_{in} with domain $\{Y_1, \ldots, Y_n, X_1, \ldots, X_m\}$, and a clause $P(Z_1, \ldots, Z_n) \leftarrow B_1, \ldots, B_p$ containing variables $\{Z_1, \ldots, Z_n, Z_{n+1}, \ldots, Z_q\}$. The procedure entry operation computes β_1, the abstract call substitution of B_1 with domain $\{Z_1, \ldots, Z_q\}$, as follows: The assignments of β_1 are obtained by taking the assignments $Y_i := v_i$ of β_{in}, replacing Y_i by Z_i, and adding $Z_{n+1} := f, \ldots, Z_q := f$. For the sharing, the sets $\{Y_i, Y_j\}$ are selected in β_{in}; replacing $\{Y_i, Y_j\}$ by $\{Z_i, Z_j\}$ yields the sets for β_1.

Theorem. Procedure entry satisfies the correctness condition: if $\tau \in \gamma(\beta_{\text{in}})$, *and if* $P(Y_1, \ldots, Y_n)\tau$ *and* $P(Z_1, \ldots, Z_n)$ *have mgu* θ, *then*

$$\big(P(Z_1, \ldots, Z_n) \leftarrow B_1, \ldots, B_p\big)\theta \in \gamma\Big(\big(P(Z_1, \ldots, Z_n) \leftarrow B_1, \ldots, B_p\big)\beta_1\Big).$$

PROOF. Assume $\tau = \{Y_1 \leftarrow t_1, \ldots, Y_n \leftarrow t_n, X_1 \leftarrow s_1, \ldots, X_m \leftarrow s_m\} \in \gamma(\beta_{\text{in}})$. Then

$$P\tau = P(t_1, \ldots, t_n).$$

$\theta = \{Z_1 \leftarrow t_1, \ldots, Z_n \leftarrow t_n\}$ is a mgu of $P(t_1, \ldots, t_n)$ and $P(Z_1, \ldots, Z_n)$.

Let σ be $\{Z_1 \leftarrow t_1, \ldots, Z_n \leftarrow t_n, Z_{n+1}\ Z_{n+1}, \ldots, Z_q \leftarrow Z_q\}$. Obviously, $(P(Z_1, \ldots, Z_n) \leftarrow B_1, \ldots, B_p)\theta = (P(Z_1, \ldots, Z_n) \leftarrow B_1, \ldots, B_p)\sigma$, so it suffices to show that $\sigma \in \gamma(\beta_1)$.

The variables $Z_{n+1}, \ldots, Z_q$ have mode f in β_1; this mode allows the values they have in σ.

For the variables $Z_1, \ldots, Z_n$: Z_i in σ has the same value as Y_i in τ. By construction of β_1, the mode of Z_i in β_1 is the same as of Y_i in β_{in}, so the modes of the Z_i in β_1 are correct.

The variables $Z_{n+1},\ldots,Z_q$ do not occur in $t_1,\ldots,t_n$, so sharing is only possible between a pair of terms t_i, t_j. If so, the sharing also exists in τ, $\tau \in \gamma(\beta_{\text{in}})$; thus $\{Y_i, Y_j\} \in \beta_{\text{in}}$. By construction $\{Z_i, Z_j\} \in \beta_1$ and the sharing is allowed by β_1.

Thus $\sigma \in \gamma(\beta_1)$.

4.2.2. Procedure Exit. Given a call $P(Y_1,\ldots,Y_n)$ which is part of a clause $H \leftarrow B_1,\ldots,P(Y_1,\ldots,Y_n),\ldots,B_p$; the clause contains the variables $\{Y_1,\ldots,Y_n,X_1,\ldots,X_m\}$. P is defined by a set of clauses $P(Z_1^j,\ldots,Z_n^j) \leftarrow D_1^j,\ldots,D_{q_j}^j$ with variables $\{Z_1^j,\ldots,Z_n^j,Z_{n+1}^j,\ldots,Z_{r_j}^j\}$. β_j^{out} is the abstract success substitution of $D_{q_j}^j$. β_{in} is the abstract call substitution of $P(Y_1,\ldots,Y_n)$.

The procedure-exit operation computes β_{out}, the abstract success substitution of $P(Y_1,\ldots,Y_n)$. The computation is done in three steps:

1. β_j' is an abstract substitution over the variables $Y_1,\ldots,Y_n$. $Y_i := v_i \in \beta_j'$ iff $Z_i^j := v_i \in \beta_j^{\text{out}}$. β_j' contains a pair $\{Y_i, Y_k\}$ iff β_j^{out} contains the pair $\{Z_i^j, Z_k^j\}$.
2. β is the lub of $\{\ldots,\beta_j',\ldots\}$.
3. β_{out} is derived from β_{in} and β as follows (this step computes the *extension* of β_{in} and β:
   $Y_i := v_i \in \beta_{\text{out}}$ iff $Y_i := v_i \in \beta$.
   The assignment of X_i in β_{out}:
   if $X_i := g$ or $X_i := a$ in β_{in}, then the same assignment in β_{out};
   if $X_i := f$ in β_{in}, then
   if $\exists Y_j(\{X_i, Y_j\} \in \beta_{\text{in}}$ and the mode of Y_j in β is not $f)$
   then X_i has mode a in β_{out}
   else X_i has mode f in β_{out}.
   If $\{Y_i, Y_j\} \in \beta$ then $\{Y_i, Y_j\} \in \beta_{\text{out}}$.
   If $\{X_i, X_j\} \in \beta_{\text{in}}$ then $\{X_i, X_j\} \in \beta_{\text{out}}$.
   If $\{X_i, Y_j\} \in \beta_{\text{in}}$ and the mode of Y_j in β is not g, then $\{X_i, Y_j\} \in \beta_{\text{out}}$.
   If $\{X_i, Y_k\} \in \beta_{\text{in}}$ and $\{Y_j, Y_k\} \in \beta$, then $\{X_i, Y_j\} \in \beta_{\text{out}}$.
   If $\{X_i, Y_k\} \in \beta_{\text{in}}$, $\{X_j, Y_k\} \in \beta_{\text{in}}$, the mode of Y_k in β_{in} is a, and the mode of Y_k in β is a,
   then $\{X_i, X_j\} \in \beta_{\text{out}}$ (if Y_k has mode f and X_i and X_j both share a variable with Y_k, then X_i shares a variable with X_j and $\{X_i, X_j\} \in \beta_{\text{in}}$).
   If $\{X_i, Y_k\} \in \beta_{\text{in}}$, $\{X_j, Y_1\} \in \beta_{\text{in}}$, and $\{Y_k, Y_1\} \in \beta$, then $\{X_i, X_j\} \in \beta_{\text{out}}$.

Theorem. Procedure exit satisfies the correctness conditions:

I. *If* $\tau_j \in \gamma(\beta_j^{\text{out}})$ *and there exists a pair of substitutions* τ_{in}, θ *such that* $\tau_{\text{in}} \in \gamma(\beta_{\text{in}})$ *and* $P\tau_{\text{in}}\theta = H_j\tau_j$, *then there exists a* $\tau \in \gamma(\beta)$ *such that* $P\tau = P\tau_{\text{in}}\theta$.

II. *If* $\tau_{\text{in}} \in \gamma(\beta_{\text{in}})$ *and there exists a substitution* θ *such that* $P\tau_{\text{in}}\theta \in \gamma(P\beta)$ *and the only variables of* $(H \leftarrow B_1,\ldots,P,\ldots,B_p)\tau_{\text{in}}$ *occurring in* θ *also occur in* $P\tau_{\text{in}}$, *then*

$$(H \leftarrow B_1,\ldots,P,\ldots,B_p)\tau_{\text{in}}\theta \in \gamma\big((H \leftarrow B_1,\ldots,P,\ldots,B_p)\beta_{\text{out}}\big).$$

PROOF OF I. Assume

$$\tau_j = \left\{Z_1^j \leftarrow t_1, \ldots,\ Z_n^j \leftarrow t_n, \ldots,\ Z_{r_j}^j \leftarrow t_{r_j}\right\} \in \gamma\left(\beta_j^{\text{out}}\right).$$

Assume

$$\tau_{\text{in}} = \{Y_1 \leftarrow s_1, \ldots,\ Y_n \leftarrow s_n,\ X_1 \leftarrow r_1, \ldots,\ X_m \leftarrow r_m\} \in \gamma(\beta_{\text{in}}).$$

Assume θ such that

$$P\left(Z_1^j, \ldots, Z_n^j\right)\tau_j = P(Y_1, \ldots, Y_n)\tau_{\text{in}}\theta.$$

This implies $t_1 = s_1\theta, \ldots,\ t_n = s_n\theta$.

Let τ be $\{Y_1 \leftarrow s_1\theta, \ldots,\ Y_n \leftarrow s_n\theta\}$. Obviously, $P(Y_1, \ldots, Y_n)\tau_{\text{in}}\theta = P(Y_1, \ldots, Y_n)\tau$, so it suffices to show that $\tau \in \gamma(\beta)$.

The variables Y_i have the same value ($t_i = s_i\theta$) in τ as Z_i^j in τ_j. The mode of Z_i^j in β_j^{out} has been combined with other modes by the lub operator to obtain the mode of Y_i in β. So the value t_i is allowed by the mode of Y_i.

Suppose Y_i and Y_k share a variable in τ. That means $s_i\theta = t_i$ and $s_k\theta = t_k$ share a variable. But $\tau_j \in \gamma(\beta_j^{\text{out}})$; thus $\{Z_i^j, Z_k^j\} \in \beta_j^{\text{out}}$. By construction of β, $\{Y_i, Y_k\} \in \beta$, so the sharing in τ is allowed by β. □

PROOF OF II. Assume

$$\tau_{\text{in}} = \{Y_1 \leftarrow s_1, \ldots,\ Y_n \leftarrow s_n,\ X_1 \leftarrow r_1, \ldots,\ X_m \leftarrow r_m\} \in \gamma(\beta_{\text{in}}).$$

Assume θ satisfies the condition and $P(Y_1, \ldots, Y_n)\tau_{\text{in}}\theta \in \gamma(P(Y_1, \ldots, Y_n)\beta)$. This implies there exists a $\tau \in \gamma(\beta)$ such that $\tau = \{Y_1 \leftarrow s_1\theta, \ldots,\ Y_n \leftarrow s_n\theta\}$. Let σ be $\{Y_1 \leftarrow s_1\theta, \ldots,\ Y_n \leftarrow s_n\theta,\ X_1 \leftarrow r_1\theta, \ldots,\ X_m \leftarrow r_m\theta\}$. Then $(H \leftarrow B_1, \ldots, B_p)\tau_{\text{in}}\theta = (H \leftarrow B_1, \ldots, B_p)\sigma$, so it suffices to show that $\sigma \in \gamma(\beta_{\text{out}})$.

The variables Y_i have the same value in σ as in τ. By construction Y_i has the same mode in β_{out} as in β.

The variables X_i have value $r_i\theta$ in σ. Obviously, $r_i\theta$ is allowed when X_i has mode a in β_{out}. The mode of X_i in β_{out} differs from a in only two cases:

(1) X_i has mode g in β_{out} iff it has mode g in β_{in}. But that means r_i, the value of X_i in τ_{in}, is ground. Thus also $r_i\theta$ is ground, and the mode is correct.

(2) X_i has mode f in β_{out} iff it has mode f in β_{in} and, for all Y_j such that $\{X_i, Y_j\} \in \beta_{\text{in}}$, Y_j has mode f in β_{out}. X_i has mode f in β_{in}; thus r_i is a variable. It is a variable in σ, and the mode in β_{out} is correct unless $r_i\theta$ is not a variable. Is this possible? With the given assumption, θ mentions r_i only if $r_i \in \text{var}(s_1, \ldots, s_n)$. So $r_i\theta$ can only differ from r_i when there is some $Y_j \leftarrow s_j$ with $r_i \in \text{var}(s_j)$ in τ_{in}. But that means $\{X_i, Y_j\} \in \beta_{\text{in}}$. In this case, X_i has mode f in β_{out} only if Y_j has mode f in β. But that means $s_j\theta$ is a variable, so also s_j is a variable. So in τ_{in} we have in fact $X_i \leftarrow s_j$, $Y_j \leftarrow s_j$ with s_j a variable. Finally, $s_j\theta$ is a variable, so mode f is correct for X_i in β_{out}.

Finally, we have to consider the sharing of variables in σ. We distinguish three cases:

(1) Suppose Y_i and Y_j share a variable in σ. That means $s_i\theta$ and $s_j\theta$ share a variable. $\tau \in \gamma(\beta)$; thus $\{Y_i, Y_j\} \in \beta$. By construction, $\{Y_i, Y_j\} \in \beta_{\text{out}}$, so this sharing in σ is allowed by β_{out}.

(2) Suppose X_i and Y_j share a variable in σ. That means $r_i\theta$ and $s_j\theta$ share a variable. We distinguish two cases:
   (a) r_i and s_j share a variable. $\tau_{\text{in}} \in \gamma(\beta_{\text{in}})$; thus $\{X_i, Y_j\} \in \beta_{\text{in}}$. Moreover, $s_j\theta$ is nonground; thus by construction $\{X_i, Y_j\} \in \beta_{\text{out}}$.
   (b) r_i and s_j do not share a variable. Under the given assumption for θ, $r_i\theta$ and $s_j\theta$ can only share a variable when $\text{var}(r_i) \cap \text{var}(s_1, \ldots, s_n) \neq \emptyset$. Suppose r_i shares a variable with s_k. That implies that $\{X_i, Y_k\} \in \beta_{\text{in}}$. Moreover, sharing between $r_i\theta$ and $s_j\theta$ can only result if also $s_j\theta$ and $s_k\theta$ share a variable. $\tau \in \gamma(\beta)$; thus by construction $\{Y_j, Y_k\} \in \beta$, and the extension step of the construction adds $\{X_i, Y_j\}$ to β_{out}.

(3) Suppose X_i and X_j share a variable in σ. That means $r_i\theta$ and $r_j\theta$ share a variable. We can distinguish two cases:
   (a) r_i and r_j share a variable. In this case we have $\{X_i, X_j\} \in \beta_{\text{in}}$; by construction $\{X_i, X_j\}$ belongs also to β_{out}.
   (b) r_i and r_j do not share a variable. With the given assumptions for θ, $r_i\theta$ and $r_j\theta$ can share a variable only in the following cases:
   There exists a Y_k such that r_i and s_k share a variable, r_j and s_k share *another* variable, and s_k and $s_k\theta$ are nonground. But that means that $\{X_i, Y_k\} \in \beta_{\text{in}}$, $\{X_j, Y_k\} \in \beta_{\text{in}}$, the mode of Y_k in β_{in} is a, and the mode of Y_k in β_{out} is not ground and thus also is a. By construction $\{X_i, X_j\}$ is added to β_{out} under these conditions.
   There exists a pair Y_k, Y_1 such that r_i and s_k share a variable, r_j and s_1 share a variable, and $s_k\theta$ and $s_1\theta$ share a variable. But that means that $\{X_i, Y_k\} \in \beta_{\text{in}}$, $\{X_j, Y_1\} \in \beta_{\text{in}}$, and $\{Y_k, Y_1\} \in \beta$. By construction $\{X_i, X_j\}$ is added to β_{out} under these conditions.

Thus $\sigma \in \gamma(\beta_{\text{out}})$. □

4.2.3. Abstract Interpretation of $X = Y$. Given β_{in}, the abstract call substitution over a domain $\{X, Y, Z_1, \ldots, Z_n\}$. Abstract interpretation of the built-in computes β_{out}. The computation is done in two steps:

1. β, the abstract substitution over $\{X, Y\}$ after evaluation of $X = Y$ is computed as follows: if $X := g \in \beta_{\text{in}}$ or $Y := g \in \beta_{\text{in}}$, then $\beta = \{X := g, Y := g\}$, else if $X := f \in \beta_{\text{in}}$ and $Y := f \in \beta_{\text{in}}$, then $\beta = \{X := f, Y := f, \{X, Y\}\}$, else $\beta = \{X := a, Y := a, \{X, Y\}\}$
2. The *extension* of β_{in} and β yields β_{out} (see procedure exit).

The correctness conditions for a built-in P are:

I. If $\tau_{\text{in}} \in \gamma(\beta_{\text{in}})$ and $P\tau_{\text{in}}$ succeeds with substitution θ, then there exists a $\tau \in \gamma(\beta)$ such that $P\tau = P\tau_{\text{in}}\theta$.

II. If $\tau_{\text{in}} \in \gamma(\beta_{\text{in}})$ and there exists a substitution θ such that $P\tau_{\text{in}}\theta \in \gamma(P\beta)$ and the only variables of $(H \leftarrow B_1, \ldots, P, \ldots, B_n)\tau_{\text{in}}$ occurring in θ also occur in $P\tau_{\text{in}}$, then $(H \leftarrow B_1, \ldots, P, \ldots, B_n)\tau_{\text{in}}\theta \in \gamma((H \leftarrow B_1, \ldots, P, \ldots, B_n)\beta_{\text{out}})$.

II is the condition for the extension operator, it has already been proved correct. I is so obvious that we omit the proof.

4.2.4. Abstract Interpretation of $X = f(Y_1, \ldots, Y_n)$. Given β_{in} the abstract call substitution over a domain $\{X, Y_1, \ldots, Y_n, Z_1, \ldots, Z_m\}$. The computation is again in two steps:

1. β, the abstract substitution over $\{X, Y_1, \ldots, Y_n\}$ after evaluation of $X = f(Y_1, \ldots, Y_n)$, is computed as follows:
   if $X := g \in \beta_{in}$ then $\beta = \{X := g, Y_1 := g, \ldots, Y_n := g\}$,
   else if $Y_1 := g$ and ... and $Y_n := g$ in β_{in} then $\beta = \{X := g, Y_1 := g, \ldots, Y_n := g\}$,
   else if $X := f \in \beta_{in}$ then
   ($X := a \in \beta$,
   if $Y_i := m_i \in \beta_{in}$ then $Y_i := m_i \in \beta$,
   if $Y_i \neq g$ in β_{in} then $\{X, Y_i\} \in \beta$,
   if $\{Y_i, Y_j\} \in \beta_{in}$ then $\{Y_i, Y_j\} \in \beta$),
   else $X := a \in \beta_{in}$)
   ($X := a \in \beta$,
   if $Y_i := g \in \beta_{in}$ then $Y_i := g \in \beta$,
   if $Y_i \neq g$ in β_{in} then $Y_i := a \in \beta$ and $\{X, Y_i\} \in \beta$,
   if $Y_i \neq g$ in β_{in} and $Y_j \neq g$ in β_{in} then $\{Y_i, Y_j\} \in \beta$).
2. The *extension* of β_{in} and β yields β_{out}.

The correctness proof is similar to the proof for $X = Y$ and is omitted.

The abstract call substitution of a recursive call is initialized with $\perp$. It is possible to show that all defined operations are monotone.

Example.

$P(X,Y) \leftarrow Q(X,Y), R(X), S(Y)$
$Q(X,Y) \leftarrow X = Y$
$R(X) \leftarrow X = a$
$S(X) \leftarrow$
$\leftarrow P(X,Y)$

The abstract call substitution of $P(X,Y)$ is $\{X := f, Y := f\}$. Abstract interpretation of $Q(X,Y)$ yields $\{X := f, Y := f, \{X,Y\}\}$. The abstract interpretation of R yields $\{X := g, Y := a\}$. This is correct although imprecise. More precision can be obtained (1) by applying the repeat-previous-call strategy [another abstract interpretation of $Q(X,Y)$ will yield $\{X := g, Y := g\}$], (2) by refining the abstract domain, (3) by defining the procedure-entry and procedure-exit operations for normal Horn clauses and reformulating the program.

5. DISCUSSION

We have developed a novel framework for the abstract interpretation of logic programs. What novel is:

The use of an abstract AND-OR graph to store the collected information.

The mapping between abstract AND-OR graphs and sets of AND trees (proof trees), which provides a clear and concise semantics and allows us to formulate correctness conditions for the application dependent operations.

The close correspondence between AND trees and SLD derivations [12] provides a simple correctness criterion for abstract interpretation. The abstract AND-OR graph must represent all SLD derivations of interest. In our case, the SLD derivations of interest are those achievable by a standard PROLOG interpreter. We have developed a procedure to construct the abstract AND-OR graph and conditions to guarantee the correctness of the constructed tree and termination of the procedure. We have shown that the framework is a good basis for the development of applications by developing a simple case of mode inferencing (a more complex case is described in [3]).

The framework is extendible. One could add attributes to the abstract AND-OR graph. For example, one could adorn each clause of the abstract AND-OR graph with an attribute specifying the "liveness" properties of its variables immediately after procedure entry. This requires one to extend the mapping to concrete AND trees and to adapt the correctness conditions of procedure entry and procedure exit. This allows a clean and rigorous correctness proof of the compile-time garbage-collection application described in [4, 5].

Our interest is in all SLD derivations, so our abstract interpretation procedure mimics quite closely a standard PROLOG interpreter. In certain applications, the interest is restricted to the *SLD refutations*. This makes it feasible to consider other procedures for the construction of the abstract AND-OR graph. For example, one could use a bottom-up strategy as in [13]. Still, developing a semantics by defining a mapping to the set of successful AND trees (SLD refutations), and developing correctness conditions for the primitive operations which instantiate the general mechanism to a particular application, seems to be a worthwhile undertaking which would greatly facilitate the development of a correct application.

The use of abstract interpretation in computer science can be traced back [19] to Peter Naur [16], who used it for type checking in Algol compilers. The work of Sintzoff [18] was also an important milestone. Finally, the seminal paper of Cousot and Cousot provided a (complex) theoretical foundation for the abstract interpretation of programs in imperative languages.

Global analysis of programs also attracted the interest of researchers in the area of logic programming. One of the first and best-known examples is the work of Mellish on automatic generation of modes. It is reported in [14]. The early attempts were very pragmatic. Inspired by [7], one looked for a more formal basis. Debray developed a system for mode inferencing [8]; its power compares well with our example in Section 4. However, the "safety criterion" is not very elegant.

Mellish [15] uses the idea of a trace to summarize the execution of a query. A trace is quite similar to a generalized AND tree. However, when making the abstraction, the sets of call substitutions of different calls to the same predicate are lumped together in a single set input; similarly the sets of success substitutions are lumped together in the set output.

Jones and Sondergaard [10] also have a generic procedure, their *core* semantics, which must be augmented with application dependent *auxiliary functions* similar to our operations. These auxiliary functions operate on an abstract domain consisting of appropriate approximations of the *collecting* semantics. They distinguish between different call instances. However, there is only one instance of every clause, so substitutions originating from different call instances are lumped together. This

corresponds to a special version of our algorithm as sketched in remark (6) of Section 3.7.

Besides allowing for a more detailed analysis (we can distinguish between different instances of the same clause), we believe our main contribution is to formulate precise correctness conditions for the operations making up an application. As shown in Section 4 and in [3, 5], these conditions are suited to develop correct applications.

The approach of Kanamori and Kawamura [11] is quite close to ours. Where we construct an AND-OR graph with individual calls as nodes, they construct an OR tree with goal statements as nodes. The information about call and success substitutions is stored in an extension table. Their formalism is geared towards a set representation of the abstract domain. The upper-bound operation used to combine solutions originating from different clauses is the set union. Compared with our approach, this seems to be a restriction, although not fundamental. If the call selected for execution from a goal statement is already in the extension table, then they do not develop an OR branch for each defining clause, but an OR branch for each solution in the current set of solutions stored in the extension table. This corresponds to the case where we do not expand a call, but use the results from the ancestor. Each solution is used as a fact to solve the call. When later on other solutions are found, then additional OR branches are added. This corresponds to the case where we start another iteration, using the upper bound of solutions obtained so far. As the upper bound is always set union in their case, this reduces to adding additional branches.

Finally, it is interesting to compare our framework with the inductive assertion method of Drabent and Maluszynski [9]. Assertions are somewhat similar to abstract substitutions. One important difference is that output assertions can refer back to the corresponding input assertion. This allows one to distinguish cases depending on the input. Our use of the abstract AND-OR graph gives exactly the same expressive power in this regard. Indeed, one obtains different instances of calls to the same predicate. Each abstract success substitution gives a specification of the output for the corresponding input.

There is a more fundamental difference between the two approaches. With abstract interpretation, one has to devise a language for abstract substitutions, to define primitive operations, and to prove correctness of the primitive operations (using only local information). Once this is done, one can automatically obtain correct abstract substitutions for all programs. The inductive assertion method is a manual method. There is no limitation on the expressiveness of assertions. Also, in proving their correctness, one is not limited to the information which is locally available. However, the work has to be done for each program.

This work has grown out of an attempt to provide correctness arguments for an engineering effort towards improved PROLOG compilers. We felt existing methods were too complex and not suited as a basis for the desired correctness proofs. The current framework and the proofs emerged after many iterations. We are indebted to many people for discussions, patience, support, encouragement, and insistence on more rigor. Most notable are A. Callebaut, B. Demoen, D. De Schreye, W. Drabent, G. Janssens, F. Kluzniak, B. Martens, K. Musumbu, H. Sondergaard, and P. Weemeeuw. Also, comments by the referees were very helpful.

REFERENCES

1. Abramsky, S. and Hankin, C. (eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.
2. Bruynooghe, M., A Framework for the Abstract Interpretation of Logic Programs, Report CW62, Dept. of Computer Science, Katholieke Univ. Leuven, 1987.
3. Bruynooghe, M. and Janssens, G., An Instance of Abstract Interpretation Integrating Type and Mode Inferencing, in: *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, MIT Press, 1988, pp. 669–683.
4. Bruynooghe, M., Janssens, G., Callebaut, A., and Demoen, B., Abstract Interpretation: Towards the Global Optimization of PROLOG Programs, in: *Proceedings of the Fourth Symposium on Logic Programming*, San Francisco, IEEE Society Press, 1987, pp. 192–204.
5. Bruynooghe, M., Mulkers, A., and Winsborough, W., Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs. To appear in *Proceedings of the Seventh International Conference on Logic Programming*, Jerusalem, MIT Press, 1990.
6. Colmerauer, A., Prolog and Infinite Trees, in: K. L. Clark and S. A. Tarnlund (eds.), *Logic Programming*, Academic, 1982, pp. 231–251.
7. Cousot, P. and Cousot, R., Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, 1977, pp. 238–252.
8. Debray, S. K. and Warren, D. S., Automatic Mode Inferencing for Prolog Programs, in: *Proceedings of the Third Symposium on Logic Programming*, Salt Lake City, IEEE Society Press, 1986, pp. 78–88; *J. Logic Programming* 5:207–229 (1988).
9. Drabent, W. and Maluszynski, J., Inductive Assertion Method for Logic Programs, in: *Proceedings of TAPSOFT* 87, Pisa, Springer-Verlag, 1987, pp. 167–181.
10. Jones, N. D. and Sondergaard, H., A Semantics Based Framework for the Abstract Interpretation of Prolog, in: S. Abramsky and C. Hankin (eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987, pp. 123–142.
11. Kanamori, T. and Kawamura, T., Analyzing Success Patterns of Logic Programs by Abstract Hybrid Interpretation, ICOT Technical Report TR279, 1987.
12. Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, 1984.
13. Marriott, K. and Sondergaard, H., Bottom-Up Abstract Interpretation of Logic Programs, in: *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, MIT Press, 1988, pp. 733–748.
14. Mellish, C. S., Some Global Optimizations for a Prolog Compiler, *J. Logic Programming* 2:43–66 (1985).
15. Mellish, C. S., Abstract Interpretation of Prolog Programs, in: *Proceedings of the Third International Conference on Logic Programming*, London, Springer-Verlag, 1986, pp. 463–474; revised in: S. Abramsky and C. Hankin (eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.
16. Naur, P., The Design of the Gier Algol Compiler, Part II, *BIT* 3:145–166 (1963).
17. O'Keefe, R. A., Finite Fixed-point Problems, in: *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, MIT Press, 1987, pp. 729–743.
18. Sintzoff, M., Calculating Properties of Programs by Valuation on Specific Models, in: *Proceedings of the ACM Conference on Proving Assertions about Programs*, New Mexico (*Sigplan Notices* 7, No. 1), 1972, pp. 203–207.
19. Sondergaard, H., Abstract Interpretation of Logic Programs, DIKU Project 86-7-10, Univ. of Copenhagen, 1987.