Automatic Binding-related Error Diagnosis in Logic Programs

Paweł Pietrzak[1] and Manuel V. Hermenegildo[1,2]

[1] School of Computer Science, Technical University of Madrid (UPM)
[2] Depts. of CS and ECE, University of New Mexico

Abstract. This paper proposes a diagnosis algorithm for locating a certain kind of errors in logic programs: variable binding errors that result in abstract symptoms during compile-time checking of assertions based on abstract interpretation. The diagnoser analyzes the graph generated by the abstract interpreter, which is a provably safe approximation of the program semantics. The proposed algorithm traverses this graph to find the point where the actual error originates (a reason of the symptom), leading to the point the error has been reported (the symptom). The procedure is fully automatic, not requiring any interaction with the user. A prototype diagnoser has been implemented and preliminary results are encouraging.

1 Introduction

Obtaining a program that satisfies the programmer's intentions is clearly a crucial objective in software development. If the program does not conform to the user's expectations (i.e., if it contains a discrepancy between the program semantics and the specification –a *symptom*) this means that somewhere in the program there is an error which has to be found and corrected. The difficulty in this process comes from the fact that the effects of a given error (the symptoms) propagate from one location in the code to another and manifest themselves far away from the place where the error resides. The process of locating (a piece of code that contains) an error given some symptom is called *diagnosis*. In this paper we address the problem of compile-time automatic diagnosis in untyped (constraint) logic programs, focusing on binding errors, meaning that we locate a variable binding that eventually produces a symptom (making the program erroneous). We aim at designing sound foundations for a practical and useful diagnosis tool that can be used routinely.

Before an error can be diagnosed, its presence has to be detected through a symptom. Our approach relies on the idea of compile-time program verification and error detection based on abstract interpretation [7], as proposed in [13, 22, 14]. Properties expected by the user concern call and success patterns of program predicates and are given in the form of *assertions* [11, 19, 21]. An important characteristic of the approach used is that only a small number of assertions may be present in the program, or even no assertions at all. In the latter case the

system takes advantage of assertions written for built-in and library predicates to detect errors in user programs. Also, the approach is parametric on the abstract domains used, so that a variety of properties can be proved or disproved, based on the set of abstract domains used.

Assertion verification is preceded by static program analysis based on abstract interpretation [7, 3, 20, 15]. The results of the analysis are compared against the assertions. An assertion that can be shown to be false, together with the related program point is called a *symptom*. Such (abstract) symptoms are the starting point of our diagnosis procedure. The static analyzer produces a *program analysis graph* which is essentially a finite representation of all execution paths that may appear at run-time, annotated with the state at the call and exit points of procedures and at each program point. This graph is the fundamental data structure exploited by the diagnoser. The diagnoser traverses the graph from the point of the symptom, against the direction of execution, trying to identify a point (or points) where variable bindings occur which are responsible for the symptom. During the traversal, the abstract operations of the analyzer are executed in order to analyze parts of the graph, and thus come to conclusions regarding corresponding pieces of the code. The proposed procedure is fully automatic, and does not require any user intervention. In the implementation, the initial call to the diagnoser is (optionally) automatically triggered by the assertion checker when a discrepancy with an assertion is detected.

2 Related work

Locating errors is an inherent part of debugging[3] and has attracted significant attention. One of the best-known diagnosis techniques is *declarative* or *algorithmic debugging*, initially proposed in [24]. In this approach the search for the error takes the form of an interactive session with the user, who is required to answer queries about the intended behavior of the program. A drawback of the approach is that the number of questions posed to the user is typically very large. One way to reduce the number of queries and to simplify them is to add partial formal specifications to the program in the form of assertions [11], but the load on the user remains a problem for the practical takeup of this technique. The algorithmic debugging approach is strongly tied to the declarative semantics while our aim is to develop an approach that works also for impure (constraint) logic programs. An additional difference with our approach is that the declarative debugging session concerns the concrete semantics and a single (test) execution only, whereas we are interested in diagnosing errors at compile-time, and for all possible executions.

When the full (abstract) specification of the program is available the method known as *abstract diagnosis* [6] can be applied. This method however requires, in addition to the full specification, again adherence to the declarative semantics (and also makes the assumption that the specification can be linked with the

[3] We prefer to reserve the term "debugging" for a process that involves both locating and removing the bug.

concrete semantics via a *Galois connection*). Other techniques based on applying a verification condition, such as, e.g., [10], can also be used to locate errors. In [10], in particular, descriptive types are used to approximate the operational semantics. A clause on which the inductive proof fails indicates an error.

In the context of strongly-typed languages, the problem of locating type errors, i.e., understanding why an expression cannot be typed, has received much attention. These diagnosis algorithms try to find a reason for the failure in type unification during type inference. The problem was initially attacked in [26] where the steps of the type inference procedure are recorded and later looked up for inconsistencies. Many researchers (e.g., [2]) have followed this line and proposed various improvements. In our case we are dealing with an untyped (logic) language, and with a general class of properties that goes beyond traditional types. Also, as in the typed languages, the error might be placed far away from the expression reported in the type error message. But because in our case there may be only a few assertions in the program, the error may in fact be propagated further and show up much later, even in different functions or modules, and in a way that does not correspond intuitively with the direction of the execution-time data flow.

Our approach has a strong relationship with *slicing* (see e.g.,[12, 23] for slicing in logic programming). Note that in (backward) slicing the goal is to find a piece of program that potentially affects a value of a variable at the point of interest, whatever the value is, whereas we are interested only in values violating the specification. Also, unlike in slicing we do not track dependencies between individual variables, letting an abstract domain and the generic abstract interpreter capture the necessary information.

3 Preliminaries and notation

We assume that the reader is familiar with logic programming (see, e.g., [1, 18]) and abstract interpretation [7]. We will use the standard notions of SLD resolution and SLD derivation with the Prolog computation rule. We use a standard notion of substitution, i.e. mappings from program variables to terms. A substitution will be typically denoted as θ, possibly with sub- or superscripts. We also use $\theta_{|A}$ to denote a projection of θ over variables in an atom A. Let G_k be a *resolvent* of the form $\leftarrow (A_1, \ldots, A_n)\theta_0 \cdots \theta_k$, obtained in the k-th step of the derivation. In step $k+1$ we obtain the resolvent G_{k+1} (denoted $G_k \rightsquigarrow G_{k+1}$) of the form $\leftarrow (B_1, \ldots, B_m, A_2, \ldots, A_n)\theta_0 \cdots \theta_k\theta_{k+1}$ where $B_0 \leftarrow B_1, \ldots, B_m$ is a renamed clause of the program and $\theta_{k+1} = mgu(A_1\theta_0 \cdots \theta_k, B_0)$. Let $\overset{+}{\rightsquigarrow}$ denote a transitive closure of $\rightsquigarrow$. In order to handle program points we annotate every atom A in a derivation by a program-point identifier $\textcircled{p}$, which determines a clause and a position in the clause where A comes from. We write annotated atoms as $A^{\textcircled{p}}$. We say that a program point $\textcircled{p}$ corresponds to a derivation state G iff G is of the form $\leftarrow (A^{\textcircled{p}}, \ldots)\theta$.

Goal-directed abstract interpretation is a technique whose aim is for a given initial call pattern (describing a possibly infinite set of input data), to generate

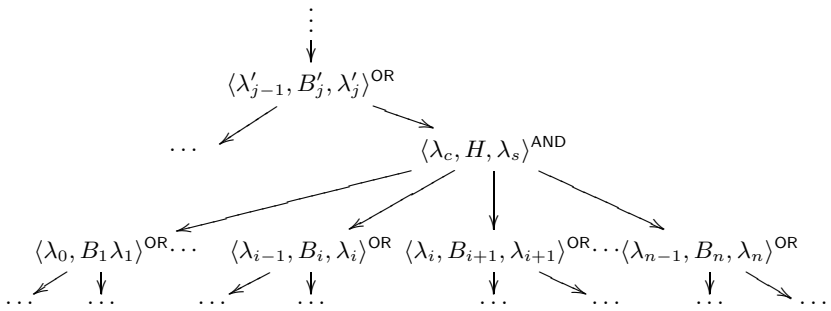

Fig. 1. A fragment of an abstract AND-OR graph.

annotations describing (in an abstract way) all possible run-time variable bindings. The annotations are expressed in an *abstract domain* D_α (a lattice equipped with a partial order $\sqsubseteq$, and standard elements $\top$ and $\bot$, and operations $\sqcup$ and $\sqcap$). In the case of logic programming, the annotations typically take the form of *abstract substitutions* (approximations of concrete substitutions), i.e., mappings from program variables to values in D_α. Abstract domain D_α and concrete domain D are linked to each other by two monotone mappings $\alpha : D \mapsto D_\alpha$ and $\gamma : D_\alpha \mapsto D$, called *abstraction* and *concretization* functions respectively. We do not restrict our attention to any specific abstract domain. However, in the examples we will use for concreteness regular types [9, 25].

Throughout this paper we will use the abstract interpretation framework of Bruynooghe [3], which, with variations and optimizations (in our case [20, 15]), is the basis of a large portion of the practical analyzers for logic programs. In these frameworks the analyzer produces a program analysis graph called an *abstract* AND-OR *graph*. The abstract AND-OR graph is a finite description of the set of (concrete) AND-OR trees that are conceptually traversed during execution of the program. The abstract graph (see Figure 1) has two sorts of nodes: AND-nodes containing (copies of) heads of clauses, which have body atoms as their children[4] and OR-nodes which are body atoms representing calls. Each node (whether AND- or OR-) is adorned with two substitutions, describing the bindings of variables just before entering and just after exiting the respective piece of program code.

Let $vars(E)$ denote a set of variables that occur in a syntactic object E. We use the notation $\langle \lambda_c, H, \lambda_s \rangle^{\mathsf{AND}}$ to denote an AND-node, where H is the head of a clause, say $H \leftarrow B_1, \ldots, B_n$. λ_c is an abstract substitution that describes the bindings of variables $vars(H)$ before entering the clause, and λ_s keeps the bindings after exiting the clause. The AND-node with the atom H has children, each being an OR-node, written as $\langle \lambda_{i-1}, B_i, \lambda_i \rangle^{\mathsf{OR}}$, for $1 \leq i \leq n$. A substitution λ_i describes bindings of variables just before calling an atom B_{i+1} (or after succeeding in the clause body if $i = n$), and it ranges over all $vars((H \leftarrow B_1, \ldots, B_n))$. We use a function *children*(N) to denote an ordered set of children of an AND- or OR-node N. We assume that all substitutions in the graph are

[4] Clearly, children of nodes have to be ordered. Moreover, there might be more than one AND-node per clause.

different from $\bot$, i.e., branches corresponding to sub-computations known to fail are not present in the graph. Variables in the program and those in the abstract AND-OR graph are renamed apart. During the analysis phase, when the abstract AND-OR graph is constructed the following abstract operations (functions) are used (following [15]):

- $\mathsf{Aproj}(\lambda, V)$ performs the projection of a substitution λ on a set of variables V.
- $\mathsf{Aextend}(\lambda, V)$ extends the substitution λ to the set of variables V.
- $\mathsf{Aunif}(E_1, E_2, \lambda)$ performs abstract unification of two expressions $E_1 = E_2$ and conjoins the results with λ.
- $\mathsf{Aconj}(\lambda_1, \lambda_2)$ performs the abstract conjunction of two substitutions.
- $\mathsf{Alub}(\lambda_1, \lambda_2)$ performs the abstract disjunction of two substitutions[5].

We recall the notion of *topmost abstract substitution* over the set of variables V (introduced in [5]), which captures the abstract representation of the most general concrete substitution, and which, e.g., for our example type domain can be defined as $\lambda_V^\top \stackrel{def}{=} \{X/\top \mid X \in V\}$. V will be dropped if it is clear from the context.

4 Assertions, Symptoms, and Errors

Our objective is to find a reason for a symptom, where the symptom is understood as a deviation of the program behavior from the user expectations. The expectations have to be expressed as a formal specification. Such specification is commonly written in terms of assertions (e.g., see [11, 19, 21]). A specific assertion language definition is not required for our results (however, the implementation and experiments are carried out in the Ciao Preprocessor [14] using its assertion language [21]). For simplicity we assume that a (partial) specification is provided in terms of abstract substitutions assigned to program points. We will refer to such substitutions as *program-point assertions* or *expected properties*, interchangeably. We explicitly allow the specification to be partial, i.e., the program can contain just a few or even no assertions, the only assertions then checked being those in libraries. A program-point assertion λ_{Prop} between atoms B_i and B_{i+1} expresses an expected success pattern of B_i or expected call pattern of B_{i+1}. Consequently, λ_{Prop} ranges over $vars(B_i)$ or $vars(B_{i+1})$. Note that we have chosen program point assertions without loss of generality since predicate-level assertions (cf. [21]) can be translated to program point assertions with a simple program transformation. We also assume that the expected properties are renamed along with the clauses, so that they range over the same variables as the copies of the clauses present in the abstract graph. These assumptions simplify the subsequent presentation. Finally, we assume that abstract values are *over*-approximations (a dual approach applies for *under*-approximations).

[5] While we will not need this operation in the paper it is included in order to make the description complete and avoid confusion.

Now we are in a position to define a notion of *symptom*. Assume that at a program point Ⓢ there is an associated assertion λ_{Prop}. A *symptom* of violating the assertion occurs whenever there is a derivation $D = G_0 \overset{+}{\rightsquigarrow} G_k \rightsquigarrow \cdots$ with a state $G_k = \leftarrow (A^{\textcircled{S}}, \ldots)\theta_0 \cdots \theta_k$ s.t. $(\theta_0 \cdots \theta_k)_{|A} \notin \gamma(\lambda_{Prop})$. In other words the assertion is expected to be satisfied for all variable bindings at Ⓢ in any possible execution. We say that D is an *assertion violating derivation*. A symptom of violating the assertion can be signaled by compile-time checking whenever the static analysis produces an abstract substitution λ at Ⓢ (for all concrete θ at Ⓢ $\theta \in \gamma(\lambda)$), s.t. $\lambda \not\sqsubseteq \lambda_{Prop}$.

We are interested in finding the reason for the symptom. We say that a derivation state $G_l = \leftarrow (B^{\textcircled{E}}, \ldots)\theta_0 \cdots \theta_l$ $(l < k)$, and the corresponding program point Ⓔ *contribute* to the symptom at Ⓢ iff for any substitution θ there is a sub-derivation D' of an assertion violating derivation D which is of the form $\leftarrow (B^{\textcircled{E}}, \ldots)\theta \overset{+}{\rightsquigarrow} \leftarrow (A^{\textcircled{S}}, \ldots)\theta\theta'_{l+1} \cdots \theta'_k$ s.t. $(\theta\theta'_{l+1} \cdots \theta'_k)_{|A} \notin \gamma(\lambda_{Prop})$. D' differs from the appropriate part of D only in computed substitutions. We assume the same clauses are selected in corresponding steps. By replacing the input substitution $\theta_0 \cdots \theta_l$ by a universally quantified substitution θ we try to determine whether the source of violating the assertion lies between derivation states G_l and G_k or it is instead propagated from states preceding G_l along with the substitution $\theta_0 \cdots \theta_l$. Note that the initial state G_0 always contributes to the symptom. This however is not useful for locating bugs. We define a *binding error* as a derivation state (and the corresponding program point) for which the sub-derivation D' has the shortest possible length. A binding error indicates a program point where a variable binding takes place which eventually leads to the symptom. Note that the actual symptom might be due to some other (non-binding) error which we are not able to detect. Nonetheless, we provide a strong indication that the actual error should be searched between the binding error and the symptom.

Our objective in this paper is to locate binding errors statically. The starting point of the error diagnosis process is compile-time assertion checking based on the output of the abstract interpretation framework. Interestingly, although abstract interpretation in general provides only safe approximations of the properties, in practice it is often possible to definitely prove or disprove an assertion. The latter case occurs when we have $\lambda_a \sqcap \lambda_{Prop} = \bot$. In those cases we say that λ_a is *incompatible* with the expected property λ_{Prop} (we will use this notion later in our diagnosis algorithm) and we have a definite error. However in some cases the system will not be able to prove or disprove a given assertion ($\lambda_a \not\sqsubseteq \lambda_{Prop}$ and $\lambda_a \sqcap \lambda_{Prop} \neq \bot$). In this case the system allows the user to choose (via flags) whether this should be considered an error, a warning, or be ignored. Our practical experience (and we understand it is also that of for example the ASTREE developers [8]) is that these cases are often actual symptoms, even if sometimes they are not and the "false alarm" is simply due to loss of precision in the analysis. Herein we will consider such cases indeed as symptoms, and start diagnosis for them, and will accept that in some cases no real error will be responsible for them, in which case we will guarantee that the diagnosis procedure will never

locate such a non-existing error and will simply report that no error could be found.

5 Traversing abstract AND–OR graphs

The core of our binding error searching procedure consists of traversing (parts of) an abstract AND-OR graph and performing abstract operations, resulting in abstract substitutions, in a similar way that they are performed during analysis. Therefore, we will need a notion of traversing an abstract AND-OR graph, and, at the same time, replacing existing abstract substitutions with the newly generated ones.

In the following, an abstract AND-OR graph R with all substitutions replaced by $\lambda^\top$ will be denoted by $R^\top$.

Definition 1 (Forward traversal of an abstract AND-OR graph R. Defintion of transition $\Rightarrow_R$).

Let $\Rightarrow_R$ be a transition over nodes of an abstract AND-OR graph R:

(Entry) $\langle \lambda'_{j-1}, B'_j, \lambda'_j \rangle^{\mathsf{OR}} \Rightarrow_R \langle \lambda_c, H, \lambda_s \rangle^{\mathsf{AND}}$ *if* $H \in children(B'_j)$
$\lambda_{add} := \mathsf{Aunif}(B'_j, H, \lambda'_{j-1})$
$\lambda_c := \mathsf{Aproj}(\lambda_{add}, vars(H))$

(Enter Body) $\langle \lambda_c, H, \lambda_s \rangle^{\mathsf{AND}} \Rightarrow_R \langle \lambda_0, B_1, \lambda_1 \rangle^{\mathsf{OR}}$ *if* $B_1 \in children(H)$
$\lambda_0 := \mathsf{Aextend}(\lambda_c, vars(H \leftarrow B_1, \ldots, B_n))$

(Move Right) $\langle \lambda_{i-1}, B_i, \lambda_i \rangle^{\mathsf{OR}} \Rightarrow_R \langle \lambda_i, B_{i+1}, \lambda_{i+1} \rangle^{\mathsf{OR}}$ *if* $\exists H', \{B_i, B_{i+1}\} \subseteq children(H')$

(Exit Body) $\langle \lambda_{n-1}, B_n, \lambda_n \rangle^{\mathsf{OR}} \Rightarrow_R \langle \lambda_c, H, \lambda_s \rangle^{\mathsf{AND}}$, *if* $B_n \in children(B)$ *and* $|children(B)| = n$ *(i.e., B_n is the rightmost child of H)*
$\lambda_s := \mathsf{Aproj}(\lambda_n, vars(H))$

(Exit) [6] $\langle \lambda_c, H, \lambda_s \rangle^{\mathsf{AND}} \Rightarrow_R \langle \lambda'_{j-1}, B'_j, \lambda'_j \rangle^{\mathsf{OR}}$, *if* $H \in children(B'_j)$
$\lambda_{add} := \mathsf{Aunif}(H, B'_j, \lambda_s)$
$\lambda_{ext} := \mathsf{Aextend}(\lambda_{add}, vars(H' \leftarrow B'_1, \ldots, B'_{n'}))$
$\lambda'_j := \mathsf{Aconj}(\lambda_c, \lambda_{ext})$ □

We will omit the subscript R in $\Rightarrow_R$ if it is clear from the context. We extend the $\Rightarrow_R$ relation to a traversal over a finite sequence of nodes $s = [s_1, \ldots, s_n]$, which will be written as $\overset{s}{\Rightarrow}_R$, and defined as follows: $s_1 \overset{s}{\Rightarrow}_R s_n$ iff $\forall_{1 \leq i < n}. s_i \Rightarrow_R s_{i+1}$, The ":" operator will denote concatenation of node sequences. For a given s, $s^\top$ will denote a sequence of nodes identical to s but with all substitutions replaced by $\lambda^\top$.

The $\overset{s}{\Rightarrow}$ relation is a basis for our binding error searching algorithm. Note that $\overset{s}{\Rightarrow}$ mimics the basic operations performed by abstract interpretation, and therefore safely approximates the concrete semantics. In the following an abstract AND–OR graph is called *fresh* if it has been adorned directly by the abstract interpretation process, i.e., no node has been modified by $\overset{s}{\Rightarrow}$ afterward.

[6] This operation differs from the corresponding one used in constructing the whole graph (cf. [3, 20, 15]), as we propagate the success substitution from one clause only.

Lemma 1. *Let R be a fresh abstract* AND-OR *graph, resulting from analyzing a program P. Assume an* OR*-node $N = \langle \lambda_i, A, \lambda_{i+1} \rangle^{\mathsf{OR}}$ in R. Assume also a sub-derivation $D = \leftarrow (A, \ldots)\theta \overset{+}{\rightsquigarrow} \leftarrow (B, \ldots)\theta\theta'$ which occurs when executing P.*

(i) If $\theta_{|A} \in \gamma(\lambda_i)$ then there is a sequence of nodes s s.t. $N \overset{s}{\Rightarrow} \langle \lambda'_j, B, \lambda'_{j+1} \rangle^{\mathsf{OR}}$ and $\theta\theta'_{|B} \in \gamma(\lambda'_j)$.

(ii) Moreover, there is a corresponding OR*-node $N^* = \langle \lambda^\top, A, \_ \rangle^{\mathsf{OR}}$ and a sequence of nodes $s^\top$ in $R^\top$ s.t. $N^* \overset{s^\top}{\Rightarrow} \langle \lambda^*_j, B, \_ \rangle^{\mathsf{OR}}$ and $\theta\theta'_{|B} \in \gamma(\lambda^*_j)$. (Obviously, we have $\lambda'_j \sqsubseteq \lambda^*_j$.)*

We say that s and $s^\top$ *approximate* a sub-derivation D.

Corollary 1. *Take the assumptions of Lemma 1. Part (ii) holds for all θ.*

Since $\overset{s}{\Rightarrow}$ performs the same sequence of abstract operations as the entire abstract interpretation process but limited to one specific path in the abstract AND-OR graph, it is evident that every step of $\overset{s}{\Rightarrow}$ generates abstract substitutions that are not more general than those produced by the static analyzer. In other words, $\overset{s}{\Rightarrow}$ never loses precision with respect to the full analysis process.

Now we justify applying $\overset{s^\top}{\Rightarrow}$ to locate binding errors.

Proposition 1. *Let $\ldots, B_i \, @ \, B_{i+1}, \ldots$ be a fragment of a clause body in the program P where λ_{Prop} is an expected property at point @. Let R denote a (fresh) abstract* AND-OR *graph obtained by the static analysis of P. Assume also that there exists in $R^\top$ a sequence of nodes $s^\top$ and an* OR*-node with atom B', s.t. $\langle \lambda^\top, B', \lambda^\top \rangle^{\mathsf{OR}} \overset{s^\top}{\Rightarrow} \langle \lambda^*_i, B_{i+1}, \_ \rangle^{\mathsf{OR}}$*

If $\lambda^*_i \sqcap \lambda_{Prop} = \bot$, and if there exists a sub-derivation D reaching @ and approximated by $s^\top$ then D contains a symptom at @. Moreover a derivation state with B' as the leftmost atom and with the corresponding substitution is a binding error related to the symptom.

PROOF: Follows from Corollary 1. □

Notice that even though the $\overset{s}{\Rightarrow}$ relation approximates the concrete semantics it is finer grained in the sense that $\overset{s}{\Rightarrow}$ distinguishes steps taken as one in an SLD derivation. The steps are selecting atoms in a resolvent, entering and exiting clauses. In fact, a starting node of a $\overset{s}{\Rightarrow}$ traversal does not have to be an OR-node, it can be an AND-node as well. This gives us an opportunity to locate some errors more precisely than would be captured by the SLD resolution semantics.

6 An example

In this section we explain informally, in terms of an example, how the $\overset{s}{\Rightarrow}$ transition is used to locate binding errors. The general idea is to traverse the abstract AND–OR graph starting from the symptom, and moving against the direction of execution, in DFS fashion. This makes it feasible to examine only those nodes which are involved in the (abstract) execution prior the symptom, and therefore only those which may potentially contain an error.

To illustrate the algorithm let us consider the following *slowsort* example, depicted in Figure 2. *slowsort* is a program that is meant to sort a list of numbers by first generating a permutation of the input list and then checking whether the generated permutation is a sorted list. The predicate `perm/2` generates permutations by removing non-deterministically an element from the list (predicate `del/3`) and then calling itself recursively for the rest of the list. The test that checks if the list is sorted is performed by `sorted/2`. This code can be augmented with an `entry` declaration: `:- entry slowsort(A,B) : list(A,num).` which declares an intended initial call pattern to the top-level/exported predicate (cf. [21]). The `entry` declaration is used by the static analyzer as the starting point of the (top-down) analysis graph.

```
slowsort(L,S) :- perm(L,S), sorted(S).

perm([],[]).
% There is a bug here:
perm(L,[H,L1]) :- del(L,H,L2), perm(L2,L1).

del([H|L],H,L).
del([H|L],E,[H|L1]) :- del(L,E,L1).

sorted([]).
sorted([_]).
sorted([X,Y|L]):- X =< Y, sorted([Y|L]).
```

Fig. 2. An erroneous *slowsort* program.

Observe that the head of the second clause of `perm/2` contains a binding error: the head should look like: `perm(L,[H|L1])`. The error results in a run-time exception ("illegal arithmetic expression") raised when the computation reaches the library predicate `=</2` in the third clause of `sorted/1`, since the second element `Y` of the input list is a list itself, rather than a number, as one would expect. In the Ciao system libraries (which subsume the classical notion of "built-ins") predicates are equipped with assertions specifying their expected call and success patterns. Therefore, the expected value of `Y` is known to the diagnoser (thanks to the modular nature of analysis) without any prior effort from the user. In fact, static assertion checking [22] is able to detect that the value of `Y` is of type $rt21$, defined by the following regular term grammar rules (see, e.g., [9]):

$$rt21 \rightarrow [\,]$$
$$rt21 \rightarrow [num, rt21].$$

with the meaning that a term of type $rt21$ is either an empty list or a two-element list, with the first element being a number and the second one being a term of type $rt21$. Therefore the value of `Y` is not compatible with the expected type *arithexpr* (arithmetic expression) which appears in the assertion for `=</2`. Let this point be the starting symptom for our diagnosis session.

A part of the abstract AND-OR graph R generated by the analyzer is depicted in Figure 3. Abstract substitutions have been left out for the sake of readability. The root OR-node ① corresponds to the `entry` declaration. The starting point of the diagnosis is the illegal call to `=</2` in the clause:

```
sorted([X,Y|L]):- X =< Y, sorted([Y|L]).
```

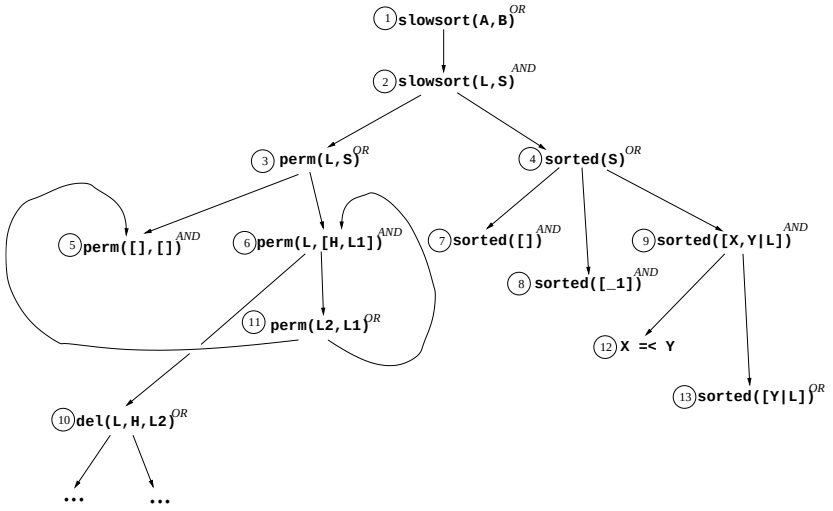

Fig. 3. A part of the abstract AND-OR graph R, an output of the analysis of the *slowsort* example from Figure 2.

which corresponds to node ⑫. Observe that node ⑬ has no descendants, as the call to `X =< Y` never succeeds, and computations never reach the goal `sorted([Y|L])`. During the diagnosis process we build a sequence of visited nodes, moving backwards, as discussed in Section 5. We will traverse the graph $R^{\top}$, starting with the AND-node ⑨, as ⑫ is its child. Let the constructed sequence of nodes be s. Initially s contains one node:

$$N_9 = \langle \{\mathtt{X}/\top, \mathtt{Y}/\top, \mathtt{L}/\top\}, \mathtt{sorted}([\mathtt{X}, \mathtt{Y}|\mathtt{L}]), \{\mathtt{X}/\top, \mathtt{Y}/\top, \mathtt{L}/\top\}\rangle^{\mathsf{AND}}$$

We keep the indexing of N's consistent with Figure 3. At this point we have reached the head of the clause, i.e., we have to find a calling atom, which is a parent OR-node ④. We add the node

$$N_4 = \langle \{\mathtt{S}/\top\}, \mathtt{sorted}(\mathtt{S}), \{\mathtt{S}/\top\}\rangle^{\mathsf{OR}}$$

to the sequence s. We need to determine if entering the clause, i.e., performing (abstract) unification when matching a calling atom and the head of clause, could cause the error symptom. To achieve this we run abstract operations corresponding to entering the clause from node ④ to ⑨ (i.e., we do the traversal $N_4 \overset{s}{\Rightarrow} N_9$ with application of rule **Entry** of Definition 1). As a result, variable `Y` is given value $\top$ in ⑫. This is not incompatible with *arithexpr* and we cannot conclude that the node ④ supports our symptom. Thus we keep on traversing the graph towards AND-node ②, and the left sibling of ④, i.e., to OR-node ③:

$$N_3 = \langle \{\mathtt{L}/\top, \mathtt{S}/\top\}, \mathtt{perm}(\mathtt{L}, \mathtt{S}), \{\mathtt{L}/\top, \mathtt{S}/\top\}\rangle^{\mathsf{OR}}$$

We now have $s = [N_3, N_4, N_9]$, which corresponds to succeeding in `perm(L,S)`. Nothing is assumed at this point about the values of `L` and `S`. The diagnoser checks whether erroneous bindings could be propagated through these variables from one of the clauses defining `perm/2`. In order to achieve this a child

(AND-node) of ③ is selected. Assume that ⑥ has been taken first. Now we have to check if the (abstract) unification performed when exiting the clause

```
perm(L,[H,L1]) :- del(L,H,L2), perm(L2,L1).
```

can cause the symptom. Traversal $N_6 \overset{s}{\Rightarrow} N_9$ is performed, using the appropriate rules **Exit, Move Right, Entry** (see Definition 1). As a result, variable `S` obtains a type of two-element list composed of $\top$. Thus, `Y` is given value $\top$ again and nothing can be concluded yet about an error causing the symptom. After performing two more steps we reach ⑥ again and we have $s = [N'_6, N_{11}, N_6, N_3, N_2, N_4, N_9]$, where each node contains related atoms with appropriate top-most abstract substitutions (we rename apart two copies of ⑥). After performing traversal $N'_6 \overset{s}{\Rightarrow} N_9$ the variable `Y` is abstractly bound to the type t defined as: $t \rightarrow [\top, \top]$. This is now incompatible with the expected property *arithexpr* and the diagnoser signals that ⑥ supports the symptom. More precisely, the point of exiting the second clause of `perm/2` after recursive call from the same clause causes the symptom. In other words, we are sure that whatever execution follows the nodes of s it will cause a run-time error at `X =< Y`.

Note that the diagnoser performs non-deterministic choices in OR-nodes. In fact, our system returns a second answer as a potential source of the symptom: the point exiting the first clause of `perm/2` after the recursive call. The corresponding sequence of nodes is $s = [N_5, N_{11}, N_6, N_3, N_2, N_4, N_9]$. This program point appears counterintuitive, and obviously is not an actual error. Nevertheless, observe that writing for example `perm([],2)` instead of `perm([],[])` would make the goal `?- slowsort([1],L)` succeed without the run-time error.

7 Binding error searching algorithm

In this section we present the actual algorithm for finding binding errors. The algorithm makes use of the $\overset{s}{\Rightarrow}$ transition introduced in Section 5, as illustrated in the example of the previous section.

The diagnosis procedure is shown in Algorithm 1. The algorithm takes as input an abstract AND–OR graph R, a clause containing a symptom at the given program point, and an expected property at that point. The set $\mathcal{E}$ of program points supporting the symptom is returned as output. The algorithm consists of the main module (lines 1-3) and two mutually recursive procedures search_AND(A, i, s) and search_OR(A, s), which implement the traversal of the graph against the control flow.

The search_OR(O, s) procedure takes the atom O in the OR–node, and the sequence s of nodes visited so far, i.e., the nodes following O in the control-flow order. Then, for every child (an AND–node) A of O, A is concatenated with s and the algorithm verifies whether the transition $\Rightarrow$ over the new sequence of nodes leads to a violation of the expected property at the program point of the initial symptom (line 25). Note that we collect nodes from $R^\top$ rather than from R. This allows us to differentiate abstract substitutions generated in the analysis phase from those generated during the error location step. I.e.,

Algorithm 1 – The diagnosis algorithm.

Input:
- analysis output in the form of an abstract AND–OR graph R,
- a clause $C^s = H^s \leftarrow B^s_1, \ldots, B^s_{n_s}$,
- an index (program point) $1 \leq i_s \leq n_s$,
- an expected abstract substitution λ_{Prop} at the program point between $B^s_{i_s}$ and $B^s_{i_s+1}$, (or after $B^s_{n_s}$ if $i_s = n_s$).

Output: a set of binding errors $\mathcal{E}$.

1: $\mathcal{E} := \emptyset$, Visited $:= \emptyset$
2: let $A := \langle \lambda^\top, H^s\sigma, \lambda^\top \rangle^{\mathsf{AND}}$ be an AND-node in $R^\top$ corresponding to clause C^s, where σ is a renaming substitution
3: search_AND$(A, i_s, [A])$
4: **procedure** search_AND(A, i, s) {A: AND–node, i: index, s: sequence of nodes}
5: **if** $i = 0$ **then**
6: let O be an OR–node s.t. $A \in \mathit{children}(O)$ in $R^\top$
7: $s' := [O] : s$
8: **if** $O \stackrel{s'}{\Rightarrow} \langle \lambda, B^s_{i_s+1}, \lambda^\top \rangle^{\mathsf{AND}}$ and $\lambda \sqcap \lambda_{Prop} = \bot$ **then**
9: $\mathcal{E} := \mathcal{E} \cup \{\mathit{entry}(O, A)\}$
10: **else**
11: let A' be an AND–node in $R^\top$ s.t. O is the j-th child of A'
12: **if** $(O, A') \notin$ Visited **then**
13: Visited $:=$ Visited $\cup \{(O, A')\}$
14: search_AND$(A', j-1, [A'] : s')$
15: **end if**
16: **end if**
17: **else if** $(A, O') \notin$ Visited **then**
18: Visited $:=$ Visited $\cup \{(A, O')\}$
19: let O' be the i-th child of A
20: search_OR$(O', [O'] : s)$
21: **end if**
22: **procedure** search_OR(O, s) {O: OR–node, s: sequence of nodes}
23: **for all** $A \in \mathit{children}(O)$ in $R^\top$ **do**
24: $s' := [A] : s$
25: **if** $A \stackrel{s'}{\Rightarrow} \langle \lambda, B^s_{i_s+1}, \lambda^\top \rangle^{\mathsf{AND}}$ and $\lambda \sqcap \lambda_{Prop} = \bot$ **then**
26: $\mathcal{E} := \mathcal{E} \cup \{\mathit{exit}(A, O)\}$
27: **else**
28: $n := |\mathit{children}(A)|$
29: search_AND(A, n, s')
30: **end if**
31: **end for**

we are able to identify if the problem causing the assertion violation is within the current sequence of visited nodes (ideally in the first one in the sequence). If we take a node from R the variable bindings that cause the problem might have been placed in the current node or they may have been propagated from the preceding (in control-flow sense) nodes through abstract substitutions. By

using $R^\top$ we "isolate" abstract values of the current nodes from the ones in the preceding nodes. If A supports the symptom the term $exit(A, O)$ is added to $\mathcal{E}$ to indicate that the critical binding occurs when the (abstract) execution leaves a clause with head A after completing call O. Otherwise, the search_AND(A, n, s) procedure is called which performs similar actions in an AND–node.

The search_AND(A, i, s) procedure performs one step of backwards traversal of an instance of a clause with head A. The body atom in question is determined by the index i. If the clause body has already been traversed ($i = 0$, line 5), the OR–node O corresponding the call to A is found and the algorithm checks whether entering the clause from O causes the symptom (line 8). If true, then, similarly to the OR–node case, the term $entry(O, A)$ is recorded in $\mathcal{E}$ (line 9). Otherwise, the search continues in the upper AND–node A' (A' is a head and O a body atom of the same clause) with the index value $j-1$ pointing to the atom just before O in the clause body (line 14). If $i > 0$ when calling search_AND(A, i, s), then the OR–node corresponding to the i-th atom in the body is inspected. Note that when search_AND is called from inside search_OR (line 29) the second argument is set to n, i.e., to the length of the corresponding clause body. The reason for this is that we want to examine the last atom in the body first, in order to find an atom supporting the symptom located as close as possible to the symptom.

As the AND–OR graph may contain cycles, the algorithm keeps track of visited nodes using for that purpose the global variable Visited. Observe, however, that AND–nodes can be visited multiple times during traversal of the graph, and therefore we need to store not only a node but also a node visited in the previous step of the current traversal. That is why the elements of Visited are pairs of nodes rather than individual nodes.

8 Conclusions and future work

We have implemented a prototype of the diagnoser in Ciao [4] and integrated it into the Ciao Preprocessor, CiaoPP [14], whose abstract interpretation engine, PLAI [20, 15]. The diagnoser makes use of abstract operations of PLAI and its data structures. The diagnoser inherits the parametric nature of the PLAI system, which allows the addition of arbitrary abstract domains as plugins. As a consequence of this the symptoms for which errors can be localized range over the same properties that can be inferred with the different domains available in the system: types/shapes, instantiation modes, pointer aliasing and structure sharing, determinacy, non-failure, etc.

The efficiency of the diagnosis procedure seems to be satisfactory, at least for the relatively small-sized programs that we have tested to date. For the *slowsort* example from Section 6, for example, the diagnosis time, including searching for all the errors, was 9.33 ms., compared to 34.66 ms. taken by the analysis. For standard *quicksort* the diagnosis took 205.97 ms. and analysis 92.98 ms. For another version of *quicksort*, i.e., with a different error, we measured 3457.47 ms. for diagnosis and 333.94 ms for analysis. Diagnosing the same bug starting from

two other, different symptoms took 2474.62 ms. and 1185.82 ms. respectively.[7] Further benchmarking of the system is planned as future work.

Inevitably, as shown in Section 6, our system may identify several points as sources of an error symptom. Not all of them are actual errors in the sense of the user's expectations, but they are all reasons for the symptom. Also, due to the approximate (but safe) nature of reasoning in abstract interpretation and consequently in our algorithm, we are not guaranteed to find sources for every symptom. In particular, this happens when the abstract value inferred by the analyzer at the point of the symptom is $\top$. This problem can often be overcome by adding more assertions to the program (something which may encourage programmers to write more assertions). In this case the assertions holding expected properties can guide the diagnosis process.

In principle, a similar effect to that achieved by our error location method could be achieved by means of backward analysis [17], and this was indeed the first solution that we considered. However, backwards analysis requires the definition of new and relatively complex operations on the abstract domain. In addition, the abstract domains used must be condensing, which is a property satisfied only by a reduced number of the domains used in practice. Our approach allows using an arbitrary abstract domain, and simplifies the implementation since it reuses the standard operations which are already defined in the system for each domain. We argue that this is a practical advantage, worth taking perhaps some performance penalty.

Acknowledgments: The authors thank the anonymous reviewers for useful comments and suggestions. This work was funded in part by EU IST FET project FP6 IST-15905 *MOBIUS*, MEC project TIN2005-09207-C03 *MERIT-COMVERS*, and CAM project S-0505/TIC/0407 PROMESAS-CM. M. Hermenegildo is also supported in part by the Prince of Asturias Chair in Information Science and Technology at UNM. P. Pietrzak is supported by a MEC "Juan de la Cierva" grant.

References

1. K.R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Model and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.
2. M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. In *ACM Letters on Programming Languages*, volume 2, pages 17–30, December 1993.
3. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
4. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.10). Technical report, School of Computer Science (UPM), 2004. Available at `http://www.ciaohome.org`.
5. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *Prooc. ESOP*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[7] The tests where run on CPU Pentium 4, 1.8GHz, and 1 GB RAM.

6. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.
7. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. POPL*, pages 238–252, 1977.
8. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *Proc. ESOP*, pages 21–30. LNCS 3444, 2005.
9. P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
10. W. Drabent, J. Małuszyński, and P. Pietrzak. Using parametric set constraints for locating errors in CLP programs. *TPLP*, 2(4–5):549–611, 2002.
11. W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.
12. T. Gyimóthy and J. Paakki. Static Slicing of Logic Programs. In *Proc. AADEBUG'95*, pages 87–103. IRISA-CNRS, 1995.
13. M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, 1999.
14. M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
15. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM TOPLAS*, 22(2):187–223, March 2000.
16. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.
17. A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *TPLP*, 2(4–5):32, July 2002.
18. J.W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd Ext. Ed., 1987.
19. D. Le Métayer. Proving properties of programs defined over recursive data structures. In *ACM PEPM*, pages 88–99, 1995.
20. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *JLP*, 13(2/3):315–347, 1992.
21. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.
22. G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Proc. of LOPSTR'99*, LNCS 1817, pages 273–292. Springer-Verlag, March 2000.
23. S. Schoenig and M. Ducasse. A Backward Slicing Algorithm for Prolog. In *Static Analysis Symposium (SAS'96)*, pages 317–331. Springer LNCS 1145, 1996.
24. E. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.
25. C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *Proc. of SAS'02*, pages 102–116. Springer LNCS 2477, 2002.
26. M. Wand. Finding the source of type errors. In *Proc. POPL*, pages 38–43, January 1986.