A Functional Specification of Declarative Debugging for Logic Programming

Rafael Caballero   Francisco J. López-Fraguas   Mario Rodríguez-Artalejo *

Departamento de Sistemas Informáticos y Programación,
Universidad Complutense de Madrid

1 Introduction

One of the major adventages of declarative programming is that it allows users to describe *what* are the characteristics of the problem they would like to solve, almost dismissing the considerations about *how* the problem is solved actually. In the same line, the aim of *algorithmic debugging* (also called *declarative debugging*) is to base the debugging process in the declarative meaning of the the program.

In this work we first look at declarative debugging as a general framework, introducing a scheme based on the concept of *computation tree*. We will discuss three possible strategies and present their specifications in a functional language, Haskell [HAS97]. Indeed our debuggers do not intend to be efficient, but only to present clearly some ideas relative to declarative debugging.

In the second part we will move into the paradigm of pure logic programming, presenting a debugging framework for logic programs written in Haskell. This completely separates the language and the meta-language, avoiding some disadvantages of other approaches.

2 The general debugging scheme

In order to develop a general scheme for declarative debugging, we assume that any computation can be represented as a convenient finite tree. Each node in the tree corresponds to the result of a partial computation. In particular the root of the tree corresponds to the result of the main computation. Moreover, we assume that the result at each node is *determined* by the results of the children nodes, and hence every node can be actually seen as the outcome of a single *computation step*. The aim of the debugger is to find out wrong computation steps in a given computation tree, and the only way of doing so is examining whether the associated results are *erroneous*. Different types of errors will need different types of trees, and also different definitions of *erroneous*.

It seems natural to say that a debugger is *sound* if all the nodes it returns have an associated wrong computation step, and *complete* if every erroneous node with an associated wrong computation step is returned. The reason is that a correct computation step can produce and erroneous node, due to deeper erroneous steps in a subcomputation within some of the children. For instance, consider the tree of Figure 2. The erroneous nodes are enclosed in a (maybe double) thick circle. In this case nodes 1,2, 4, and 7 are considered erroneous. However, the result of the computation step corresponding

*Work partially supported by the Spanish CICYT (project CICYT-TIC98-0445-C03-02/97 "TREND") and the ESPRIT Working Group 22457 (CCL-II).

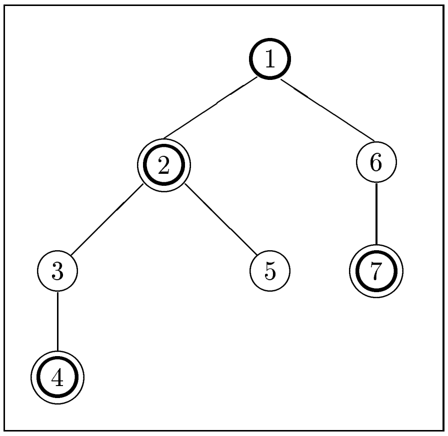

Figure 1: Tree with erroneous and buggy nodes

to node 1 may depend on the result of node 2, which is erroneous. Therefore it could be unsound to point out node 1 as a source of bugs. The situation is different in the case of node 2: it is erroneous, but it has no erroneous children, so the mistake must be in the computation of the node itself.

Following the terminology of [Nai97], an erroneous node with no erroneous children will be called a *buggy node*. In figure 2 nodes 2, 4 and 7 are buggy. The following relation between buggy and erroneous nodes holds:

A finite computation tree has an erroneous node iff it has a buggy node.

Thus, in order to avoid unsoundness, the general debugger will only look for buggy nodes. But doing so it becomes incomplete. In the case of the example of figure 2, node 1 may be erroneous due to node 2, but it also may be erroneous due to its own computation step. Hence we cannot achieve soundness and completeness simultaneously. Nevertheless, this kind of incompleteness is not extremely severe. In the same example, suppose that node 1 corresponds to a wrong computation step, but has computed a sound result just by chance, because the result of node 2 was incorrect. Then after fixing the bug in node 2, node 1 will become a buggy node, and it can be detected by using again the debugger. Therefore the debugger will take a tree representing the computation as input parameter and will return the bugs associated with the computation. We consider three possible generic debuggers. The first one finds all the buggy nodes in the given computation tree. The second and third versions find the *topmost* buggy nodes only, following two different notions of topmost: considering only the buggy nodes without buggy ancestors, or considering only the buggy nodes with no erroneous ancestors.

3 Debugging Logic Programs

The problem of declarative debugging of pure logic programming has been considered in different papers (see eg. [Fer87, Llo84]). The starting point is usually to develop the debugger as a special meta-interpreter that mimics the computation and at the same time examines it, looking for a bug. This meta-interpreter is almost always built in the same language as the programs that it debugges (e.g. Prolog). Although convinient for practical purposes, from a theoretical point of view, this approach also presents some disadventages:

• Mixing the language and the meta-language can produce subtle errors. For example, suppose that we have a metapredicate `missed` that succeeds if its argument cannot be solved with empty substitution by a given program, and that the terms of the program are represented by themselves in the debugger level. Then, if we consider the program { `p(a)` $\longleftarrow$ }, the goal `missed( p(X) )` must succeed. But this means that any instance of the goal must succeed, and in particular `missed( p(a) )`. Therefore `missed` is not correct.

• Combining the meta-interpreter and the debugger codes can benefit the efficiency, but hides the structure of the debugger itself. In particular this means that some details about the debugging strategy (i.e. which parts of the computation are actually examined) may be implicit.

In order to solve this problems we specify the debugger for logic programming in the functional language Haskell. Moreover, we will define two debuggers as instances of the general debugger.

We consider *Horn logic*, with programs as finite sets of definite clauses of the form $A \leftarrow B_1 \dots B_n$, and definite goals $\leftarrow B_1, \dots B_n$, where A, B_i are atoms and $n \geq 1$. The goals are supposed to be solved by using SLD resolution with leftmost selection rule and depth first search, and the clauses tried in the same order as they appear in the program.

We also assume the existence of an *intended interpretation* $\mathcal{I}$ of the given program P, and of signature Σ that includes all the predicate, functions and constant symbols ocurring in both P and $\mathcal{I}$.

The intended model is expected to be an open Herbrand model, presented as a set of (not necessarily ground) Σ-atoms, closed under the application of arbitrary Σ-substitutions. In particular, an atom A is valid in $\mathcal{I}$ iff $A \in \mathcal{I}$. If this is the case, then $A\theta \in \mathcal{I}$ will also hold, for every Σ-substitution θ.

Our first debugger will look for *incorrect clauses* of the program. We say that a clause $A \leftarrow W$ is incorrect w.r.t. the intended interpretation if there exists a substitution θ such that $W\theta$ is valid in $\mathcal{I}$ whereas $A\theta$ is not. On the other hand, a clause has an incorrect instance in our sense iff it has an incorrect instance in the sense of [Llo84], Chapter 4, which requires $W\theta$ to be valid in $\mathcal{I}$ and $A\theta$ to be unsatisfiable in $\mathcal{I}$. The visible symptoms of this kind of bug usually are goals W that succeed with some substitution θ such that $W\theta$ is not valid in $\mathcal{I}$. These answers $W\theta$ are called *wrong answers*.

Our second debugger will look for *atoms with incomplete sets of answers*. We define the set of instances and the set of valid instances of an atom A as: $[\![A]\!] = \{A\theta \mid \theta\, subst\}$ and $[\![A]\!]_{\mathcal{I}} = \{A\theta \mid \theta\, subst, A\theta \in \mathcal{I}\}$ respectively. If $\{\theta_1 \dots \theta_n\}$ is the set of computed answers for A, then we say that A has an incomplete set of answers iff:

$$[\![A]\!]_{\mathcal{I}} \setminus \bigcup_{i=1}^{n} [\![A\theta_i]\!] \neq \emptyset$$

This definition corresponds to the third style of missing answer diagnosis presented in [NB96], and to the model of [Nai97].

References

[Fer87] Gérard Ferrand. *Error diagnosis in Logic Programming, an adaptation of E.Y. Shapiro's method.* The Journal of Logic Programming, 1987:4:177-198

[HAS97] *Report on the Programming Language Haskell: a Non-strict, Purely Functional Language.* Version 1.4, Peterson J. and Hammond K. (eds.), January 1997.

[Llo84] John W. Lloyd. *Foundations of Logic Programming.* Springer Series in Symbolic Computation. Springer-Verlag, Berlin, Germasny, 1984.

[Nai97] Lee Naish. *A declarative debugging scheme.* The Journal of Functional and Logic Programming, 1997-3.

[NB96] Lee Naish and Timothy Barbour. *A Declarative debugger for a logical-functional language* In Graham Forsyth and Moonis Ali, eds. Eight International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems-Invited and additional papers, Vo. 2 91-99, 1995. DSTO Aeronautical and Maritime Research Laboratory.