Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs

Rafael Caballero, Francisco J. López-Fraguas, and Mario Rodríguez-Artalejo *
E-mail: {rafa,paco,mario}@sip.ucm.es

Departamento de Sistemas Informáticos y Programación,
Universidad Complutense de Madrid

Abstract. The aim of this paper is to provide theoretical foundations for the declarative debugging of wrong answers in lazy functional logic programming. We rely on a logical framework which formalizes both the intended meaning and the execution model of programs in a simple language which combines the expressivity of pure Prolog and a significant subset of Haskell. As novelties w.r.t. to previous related approaches, we deal with functional values both as arguments and as results of higher order functions, we obtain a completely formal specification of the debugging method, and we extend known soundness and completeness results for the debugging of wrong answers in logic programming to a substantially more difficult context. A prototype implementation of a working debugger is planned as future work.

1 Introduction

Traditional debugging techniques are not well suited for declarative programming languages, because of the difficult-to-predict evaluation order. In the field of logic programming, Shapiro [19] proposed declarative debugging (also called algorithmic debugging), a semi-automatic technique which allows to detect bugs on the basis of the intended meaning of the source program, disregarding operational concerns. Declarative debugging of logic programs can diagnose both wrong and missing computed answers, and it has been proved logically sound and complete [2, 8]. Later on, declarative debugging has been adapted to other programming paradigms, including lazy functional programming [15–17, 11, 14] and combined functional logic programming [13, 12]. A common feature of all these approaches is the use of a computation tree whose structure reflects the functional dependencies of a particular computation, abstracting away the evaluation order. In [12], Lee Naish has formulated a generic debugging scheme, based on computation trees, which covers all the declarative debugging methods cited above as particular instances. In the case of logic programming, [12] shows that the computation trees have a clear interpretation w.r.t. the declarative semantics of programs. On the other hand, the computation trees proposed up to now for the declarative debugging of lazy functional programs (or combined functional logic programs) do not yet have a clear logical foundation.

* Work partially supported by the Spanish CICYT (project CICYT-TIC98-0445-C03-02/97 "TREND")
The aim of this paper is to provide firm theoretical foundations for the declarative debugging of wrong answers in lazy functional logic programming. Adapting a logical framework borrowed from [5, 4], we formalize both the declarative and the operational semantics of programs in a simple language which combines the expressivity of pure Prolog [20] and a significant subset of Haskell [18]. Our approach supports a simple syntactical representation of functions as values. Following the generic scheme from [12], we define a declarative debugging method, giving a formal characterization of computation trees as proof trees that relate computed answers to the declarative semantics of programs. More precisely, we formalize a procedure for building proof trees from successful computations. This allows us to prove the logical correctness of the debugger, extending older results from the field of logic programming [2, 8] to a substantially more difficult context. Our work is intended as a foundation for the implementation of declarative debuggers for languages such as TOY [10] and Curry [7], whose execution mechanism is based on lazy narrowing.

The paper is organized as follows. Sect. 2 presents the general debugging scheme from [12], recalls some of the known approaches to the declarative debugging of lazy functional and logic programs, and gives an informal motivation of our own proposal. Sect. 3 introduces the simple functional logic language used in the rest of the paper. In Sect. 4 the logical framework which gives a formal semantics to this language is presented. Sect. 5 specifies the debugging method, as well as the formal procedure to build proof trees from successful computations. Sect. 6 concludes and points to future work.

2 Debugging with Computation Trees

The debugging scheme proposed in [12] assumes that any terminated computation can be represented as a finite tree, called computation tree. The root of this tree corresponds to the result of the main computation, and each node corresponds to the result of some intermediate subcomputation. Moreover, it is assumed that the result at each node is determined by the results of the children nodes. Therefore, every node can be seen as the outcome of a single computation step. The debugger works by traversing a given computation tree, looking for erroneous nodes. Different kinds of programming paradigms and/or errors need different types of trees, as well as different notions of erroneous. A debugger is called sound if all the bugs it reports do really correspond to wrong computation steps. Notice, however, that an erroneous node which has some erroneous child does not necessarily correspond to a wrong computation step. Following the terminology of [12], an erroneous node with no erroneous children is called a buggy node. In order to avoid unsoundness, the debugging scheme looks only for buggy nodes, asking questions to an oracle (generally the user) in order to determine which nodes are erroneous. The following relation between buggy and erroneous nodes can be easily proved:

**Proposition 1** A finite computation tree has an erroneous node iff it has a buggy node. In particular, a finite computation tree whose root node is erroneous has some buggy node.