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")

H. Kuchen and K. Ueda (Eds.): FLOPS 2001, LNCS 2024, pp. 170–184, 2001.

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 $\mathcal{TOY}$ [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.*

This result provides a 'weak' notion of *completeness* for the debugging scheme that is satisfactory in practice. Usually, actual debuggers look only for a topmost buggy node in a computation tree whose root is erroneous. Multiple bugs can be found by reiterated application of the debugger.

The known declarative debuggers can be understood as concrete instances of Naish's debugging scheme. The instances of the debugging scheme needed for diagnosing *wrong* and *missing* answers in pure Prolog are described in [12]. In these two cases, computation trees can be formally defined so that they relate answers computed by SLD resolution to the declarative semantics of programs in a precise way. This fact allows to prove logical correctness of the debugger [2, 8]. The existing declarative debuggers for lazy functional [15–17, 11, 14] and functional logic programs [13, 12] have proposed different, but essentially similar notions of computation tree. Each node contains an oriented equation $fa_1...a_n = r$ corresponding to a function call which has been evaluated, together with the returned result, and the children nodes (if any) correspond to those function calls whose evaluation became eventually needed in order to obtain $fa_1...a_n = r$. Moreover, the result r is displayed in the most evaluated form eventually reached during the computation, and the same happens for each argument a_i, except in the case of the root node [1]. Such a tree structure abstracts away the actual order in which function calls occur under the lazy evaluation strategy. A node is considered erroneous iff its oriented equation is false in the intended interpretation of the program, and the bug indication extracted from a buggy node is the instance of the oriented equation in the program applied at the outermost level to evaluate the function call in that node.

To illustrate these ideas, let us consider the small program shown in Fig. 1, written in Haskell-like, hopefully self-explanatory syntax. The data constructors s and z represent the successor of a natural number and the natural number zero, respectively, while : acts as an infix binary constructor for non-empty lists and [] is a nullary constructor for the empty list. Different defining equations for the same function f are labeled as $f.i$, with indices $i \geq 1$.

```
(from.1) from N              → N : from N
(take.1) take z     Xs       → [ ]
(take.2) take (s N) [ ]      → [ ]
(take.3) take (s N) (X : Xs) → X : take N Xs
```

Figure 1: A program example

A function call $(take\ N\ Xs)$ is intended to compute the first N elements of the list Xs, while $(from\ N)$ is intended to compute the infinite list of all numbers greater or equal than N. The definition of $from$ is mistaken, because its right-hand side should be $(N : from\ (\underline{s}\ N))$. Due to this bug, the program can compute $take\ (s(s\,z))\ (from\,z) = z : z : []$, which is false in the intended interpretation. A computation tree (built according to the method suggested in [16, 13] and related

[1] In order to avoid this exception, some actual debuggers assume a call to a nullary function `main` at the root node.

papers) would look as shown in Fig. 2, where erroneous nodes are displayed in boldface and the leftmost-topmost buggy node is surrounded by a double box.

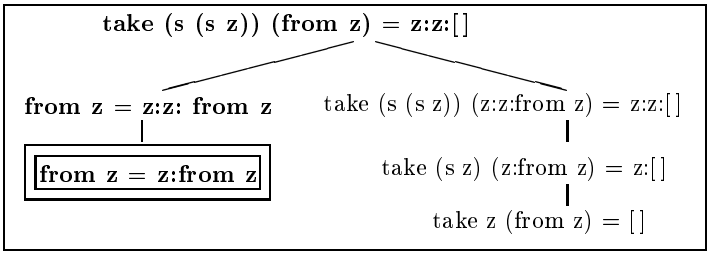


Figure 2: Computation tree with oriented equations

To the best of our knowledge, no formal proofs of correctness exist for the known lazy functional (logic) declarative debuggers, in contrast to the pleasant results shown in [2, 8] for the logic programming case. To achieve such a proof, one needs a sufficiently formal characterization of the relationship between computation trees and a suitable formalization of program semantics. The best attempt we know to formalize computation trees for lazy functional programming has been made in [16], using denotational semantics. However, as the authors acknowledge, their definition only gives an informal characterization of the function calls whose evaluation becomes eventually demanded[2]. A more practical problem with existing debuggers for lazy functional (logic) languages is related to the presentation of the questions asked to the oracle. In principle, such questions should ask whether the oriented equations $fa_1...a_n = r$ found at the tree's nodes are valid according to the intended program meaning. In these equations, both the argument expressions a_i and the result expression r can include arbitrarily complex, suspended function calls. Several solutions to this problem have been proposed, trying to ban the offending closures in various ways. In particular, Lee Naish [11] suggests the following simplification procedure: to replace unevaluated closures within the a_i by fresh variables $\overline{X}$; to replace unevaluated closures within r by fresh variables $\overline{Y}$; and to append a quantifier prefix $\forall\overline{X}\,\exists\overline{Y}$ in front of the new oriented equation. Applying this simplification method to Fig. 2, we obtain that the second child of the root node simplifies to $\forall Xs.\ take\ (s(s\,z))(z : z : Xs) = z : z : []$, while the buggy node simplifies to $\exists Ys.\ from\,z = z : Ys$. Note that the simplified question at the older buggy node has become valid in the intended program meaning. Therefore, the older buggy node is not buggy any more in the simplified tree. Its parent becomes buggy instead (and it points to the same program bug).

The example shows that Naish's simplification does not preserve the semantics of oracle questions. Moreover, a simplified oracle question like $\forall\,\overline{X}\exists\overline{Y}.fa'_1...a'_n = r'$ has the same meaning as $ft_1...t_n \rightarrow t$, where the quantifier prefix has been removed and t_i resp. t are obtained from a'_i resp. r' by substituting the *bottom* symbol $\perp$ (meaning an undefined value) in place of the new variables $\overline{X}$,

[2] Maybe this problem has been better solved in [17], a reference we obtained from the referees upon finishing the paper.

$\overline{Y}$ introduced by simplification. Due to the occurrences of $\bot$ in places where suspended function calls occurred before, the meaning of $\rightarrow$ cannot be understood as oriented equality any more. Instead, $f\,t_1 ... t_n \rightarrow t$ means that t *approximates* the value of $f\,t_1 ... t_n$, where each t_i approximates the value of the original argument expression a_i. Coming back to our example, the simplified question $\forall Xs.\ take\ (s(s\,z))\ (z : z : Xs) = z : z : []$ is equivalent to $take\ (s(s\,z))(z : z : \bot) \rightarrow z : z : []$, while $\exists Ys.from\,z = z : Ys$ is equivalent to $from\,z \rightarrow z : \bot$.

We aim at a debugging method based on computation trees whose nodes include statements of the form $f\,t_1 ... t_n \rightarrow t$, where t_i and t include no function calls, but can include occurrences of the undefined symbol $\bot$. Such statements will be called *basic facts* in the sequel. As we have seen, basic facts have a natural (not equational) meaning, and they help to obtain more simple oracle questions. Moreover, there is a well developed logical framework for functional logic programming [5, 4], based on the idea of viewing basic facts as the analogon of atomic formulas in logic programming. Relying on a variant of this framework, we will obtain a formal characterization of our debugging method.

3 A Simple Functional Logic Programming Language

The functional logic programming (FLP for short) paradigm [6] tries to bridge the gap between the two main streams in declarative programming: functional programming (FP) and logic programming (LP). For the purposes of this paper, we have chosen to work with a simple variant of a known logical framework for FLP [5, 4], which enjoys well-defined proof-theoretic and model-theoretic semantics and has been implemented in the $\mathcal{TOY}$ system [10]. In this section we present the syntax and informal semantics used for programs and goals in the rest of the paper. Pure Prolog [20] programs and Haskell-like programs [18] can be expressed in our language.

3.1 Preliminaries

A *signature with constructors* is a countable set $\Sigma = DC_\Sigma \cup FS_\Sigma$, where $DC_\Sigma = \bigcup_{n\in\mathbb{N}} DC^n_\Sigma$ and $FS_\Sigma = \bigcup_{n\in\mathbb{N}} FS^n_\Sigma$ are disjoint sets of *data constructors* and *defined function symbols* respectively, each one with an associated arity. In the sequel the explicit mention of Σ is omitted. We also assume a countable set $\mathcal{V}$ of variables, disjoint from Σ.

The set of *partial expressions* built up with aid of Σ and $\mathcal{V}$ will be denoted as $Exp_\bot$ and defined as: $Exp_\bot ::= \bot \mid X \mid h \mid (e\,e')$ with $X \in \mathcal{V}$, $h \in \Sigma$, $e, e' \in Exp_\bot$. Expressions of the form $(e\,e')$ stand for the application of e (acting as a function) to e' (acting as an argument). As usual, we assume that application associates to the left and thus $(e_0\,e_1 \ldots e_n)$ abbreviates $((\ldots(e_0\,e_1)\ldots)\,e_n)$. As explained in Sect. 2, the symbol $\bot$ (read *bottom*) represents an undefined value. We distinguish an important kind of partial expressions called *partial patterns*, denoted as $Pat_\bot$ and defined as: $Pat_\bot ::= \bot \mid X \mid c\,t_1 \ldots t_m \mid f\,t_1 \ldots t_m$ where $t_i \in Pat_\bot$, $c \in DC^n$, $0 \leq m \leq n$ and $f \in FS^n$, $0 \leq m < n$. Partial patterns represent *approximations* of the values of expressions. Moreover, partial patterns

of the form $f\ t_1 \ldots t_m$ with $f \in FS^n$ and $m < n$ serve as a convenient representation of functions as values; see [4]. Expressions and patterns without any occurrence of $\bot$ are called *total*. We write Exp and Pat for the sets of total expressions and patterns, respectively.

Total substitutions are mappings $\theta : \mathcal{V} \to Pat$ with a unique extension $\hat{\theta} : Exp \to Exp$, which will be noted also as θ. The set of all substitutions is denoted as $Subst$. The set $Subst_\bot$ of all the *partial substitutions* $\theta : \mathcal{V} \to Pat_\bot$ is defined analogously. We write $e\theta$ for the result of applying the substitution θ to the expression e. As usual, $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ stands for the substitution that satisfies $X_i\theta \equiv t_i$, with $1 \leq i \leq n$ and $Y\theta \equiv Y$ for all $Y \in \mathcal{V} \backslash \{X_1, \ldots, X_n\}$.

3.2 Programs and Goals

In our framework programs are considered as ordered sets of *defining rules* for function symbols. Rule order is not important for the logical meaning of a program. Each rule has a *left-hand side*, a *right-hand side* and an optional *condition*. The general shape of a defining rule for $f \in FS^n$ is:

(R) $\underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \to \underbrace{r}_{\text{right-hand side}} \Leftarrow \underbrace{e_1 \to p_1, \ldots, e_k \to p_k}_{\text{condition}}$ where:

(i) $t_1, \ldots, t_n, p_1, \ldots, p_k$, $(n, k \geq 0)$ is a linear sequence of patterns, where *linear* means that no variable occurs more than once in the sequence.

(ii) $r, e_1, \ldots, e_k$ are expressions. They can contain *extra variables* that don't appear in the left-hand side.

(iii) A variable in p_i can occur in e_j only if $j > i$ (in other words: p_i has no variables in common with $e_1, \ldots, e_i$).

Conditions *(i)*, *(ii)* and *(iii)* above are not too restrictive for programming and technically helpful to obtain well-behaved goal-solving calculi (as the one presented in Subsection 4.3 below). Conditions fulfilling property *(iii)* and such that the sequence $p_1, \ldots, p_k$, is linear will called *admissible* in the sequel. The intended meaning of a rule like (R) is that a call to function f can be reduced to r whenever the actual parameters match the patterns t_i and the conditions $e_j \to p_j$ are satisfied. In [5, 4], conditions of this kind are called *approximation statements*. They are satisfied whenever e_j can be evaluated to match the pattern p_j. The basic facts $f\ t_1 \ldots t_n \to t$ mentioned in Sect. 2 are particularly simple approximation statements. Readers familiar with [5, 4] will note that *joinability conditions* $e \bowtie e'$ (written as $e == e'$ in $\mathcal{TOY}$'s concrete syntax) are replaced by *approximation conditions* $e \to p$ in this paper. This is done in order to simplify the presentation, while keeping expressivity enough for our present purposes.

Fig.1 in Sect.2 shows a program for the signature $DC = \{z/0, s/1, []/0, :/2\}$, $FS = \{from/1, take/2\}$. It will be used as a running example in the rest of this paper. The reader is referred to [5, 4] for more programming examples.

A *goal* in our setting is any admissible condition $e_1 \to p_1, \ldots, e_k \to p_k$. Goals with $k = 1$ are called *atomic*. As we will see more formally in the next section, solutions to a goal $e \to p$ are substitutions θ such that $p\theta$ approximates the value of $e\theta$, according to the semantics of the current program. As in LP, a goal can include *logic variables* that are bound to patterns when the goal is solved.

For instance, considering the goal $take\ N\ (from\ X)) \to Y_s$ for our running example program, and the goal solving calculus presented in Subsection 4.3, the following solutions would be computed in this order: $\theta_1 = \{N/z, Y_s/[]\}$; $\theta_2 = \{N/(s\,z), Y_s/X : []\}$; and $\theta_3 = \{N/s\,(s\,z),\ Y_s/X : X : []\}$. Solution θ_3 is incorrect w.r.t. the intended meaning of the program, which calls for debugging. Note that the values for N and X leading to a wrong result can be found by the execution system. In a purely FP setting, the user would have been forced to guess them.

4 A Logical Framework for FLP

We are now ready to formalize a semantics and a goal solving calculus for the simple FLP language described in the previous section. We will follow the approach from [5, 4], with some modifications needed for our present purposes.

4.1 A Semantic Calculus

The *Semantic Calculus SC* displayed below specifies the meaning of our lazy FLP programs. SC is intended to derive an approximation statement $e \to t$ from a given program P just in the case that t approximates the value of e, as computed by the defining rules in P. In the SC inference rules, $e, e_i \in Exp_\bot$ are partial expressions, $t_i, t, s \in Pat_\bot$ are partial patterns and $h \in \Sigma$. Moreover, the notation $[P]_\bot$ in rule RA stands for the set $\{(l \to r \Leftarrow C)\theta \mid (l \to r \Leftarrow C) \in P, \theta \in Subst_\bot\}$ of partial instances of the defining rules in P. The SC rules are:

BT Bottom: $e \to \bot$ **RR** Restricted Reflexivity: $X \to X,\ X \in Var$

DC Decomposition: $$\frac{e_1 \to t_1\ \ldots\ e_m \to t_m}{h\ \overline{e}_m \to h\ \overline{t}_m} \qquad h\ \overline{t}_m \in Pat_\bot$$

AR Argument Reduction: $$\frac{e_1 \to t_1\ \ldots\ e_n \to t_n \quad \boxed{f\ \overline{t}_n \to s} \quad s\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \to t,} \qquad f \in FS^n,\ t \neq \bot$$

RA Rule Application: $$\frac{C \qquad r \to s}{\boxed{f\ \overline{t}_n \to s}}, \quad f\ \overline{t}_n \to r \Leftarrow C \in [P]_\bot$$

SC is similar to the rewriting calculus $GORC$ from [5, 4]. The main difference is that the $GORC$ rule OR for *Outer Reduction* has been replaced by AR and RA. Taken together, these two rules say that a call to a function f is evaluated by computing approximated values for the arguments, and then applying a defining rule for f. This is related to the *strictification* idea in [15, 17], which was intended as an emulation of the innermost evaluation order, but evaluating the arguments only as much as demanded by the rest of the computation.

The conclusion $f\overline{t}_n \to s$ of RA is a basic fact that must coincide with the corresponding premise of AR when the two rules are combined. The older calculus $GORC$ did not explicitly introduce such a basic fact, which is needed for debugging, as we have motivated in Sect. 2. The case $k > 0$ in rule AR corresponds to a higher order function f, returning as result another function (represented by the pattern s) which must be applied to some arguments $\overline{a}_k$. Just for convenience, we add to the SC calculus the following variant AR' of AR, to be used only in

the case $k = 0$. It can be shown that SC with AR' is equivalent to SC without AR'.

$$\textbf{AR'}\ \frac{e_1 \to t_1\ \ldots\ e_n \to t_n \quad \boxed{f\ \overline{t}_n \to t}}{f\ \overline{e}_n\ \to t}\ t \neq \bot, f \in FS^n$$

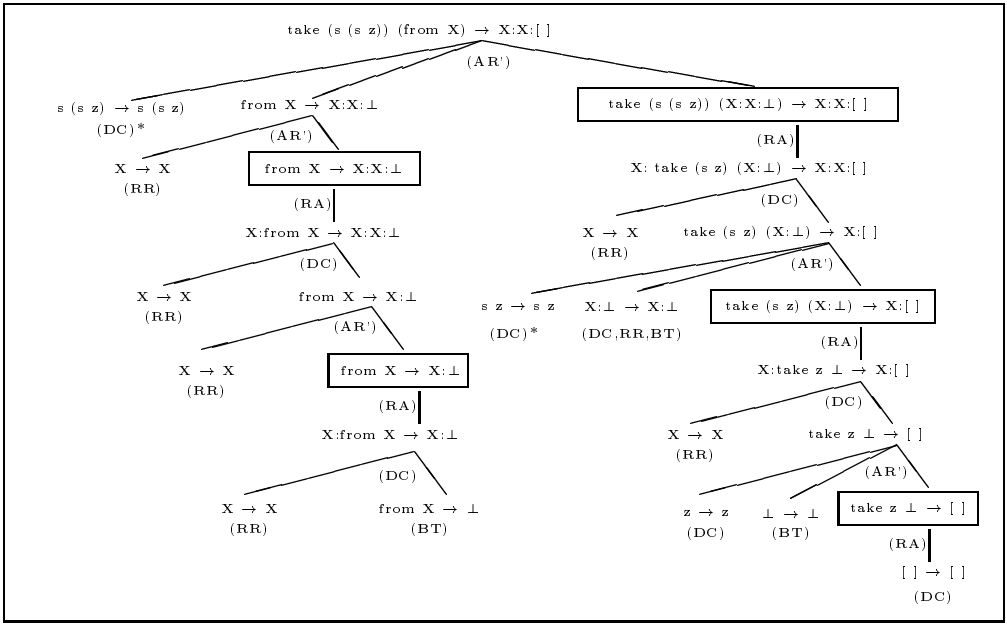


Figure 3: Proof Tree in the semantic calculus SC

We write $P \vdash e \to t$ to indicate that $e \to t$ can be deduced from P using SC. We also define a correct *solution* for a goal $G = e \to t$ w.r.t. program P as any *total substitution* $\theta \in Subst$ such that $P \vdash e\theta \to t\theta$. An SC derivation proving that this is the case can be represented as a tree, which we will call a *proof tree* (PT) for $G\theta$. Each node in a PT corresponds to an approximation statement that follows from its children by means of some SC inference. For instance, the PT from Fig. 3 shows that $\theta = \{N/s\,(s\,z),\ Ys/X : X : []\}$ is a solution for the goal $take\,N(from\ X) \to Ys$ w.r.t. our running example program. This is indeed a *bug symptom*. The right solution, according to the program's intended meaning, should be $\theta = \{N/s\,(s\,z),\ Ys/X : s\,X : []\}$.

4.2 Models

In LP the intended meaning of a program can be formalized as an *intended model*, represented as a set of atomic formulas belonging to the program's Herbrand base [2, 8]. The *open Herbrand universe* (i.e. the set of terms with variables) gives raise to a more informative semantics [3]. In our FLP setting, a natural analogon to the open Herbrand universe is the set $Pat_\bot$ of all the partial patterns, equipped with the approximation ordering: $t \sqsubseteq t' \iff_{\text{def.}} t' \sqsupseteq t \iff_{\text{def.}} \emptyset \vdash_{SC} t' \to t$. Similarly, a natural analogon to the open Herbrand base is the collection of all

the basic facts $f\ \overline{t}_n \to t$. Therefore, we define a *Herbrand interpretation* as a set $\mathcal{I}$ of basic facts fulfilling the following three natural requirements, for all $f \in FS^n$ and arbitrary partial patterns $t, \overline{t}_n$:

- $f\ \overline{t}_n \to \bot \in \mathcal{I}$.
- if $f\ \overline{t}_n \to t \in \mathcal{I}$, $t_i \sqsubseteq t'_i, t \sqsupseteq t'$ then $f\ \overline{t}'_n \to t' \in \mathcal{I}$.
- if $f\ \overline{t}_n \to t \in \mathcal{I}$, $\theta \in Subst$ then $(f\ \overline{t}_n \to t)\theta \in \mathcal{I}$.

This definition of Herbrand interpretation is simpler than the one in [5, 4], where a more general notion of interpretation (under the name *algebra*) is presented. The trade-off for this simpler presentation is to exclude non-Herbrand interpretations from our consideration. In our debugging scheme we will assume that the intended model of a program is a Herbrand interpretation $\mathcal{I}$. Herbrand interpretations can be ordered by set inclusion. In our running example, the intended interpretation contains basic facts such as $from\ X \to \bot$, $from\ X \to X : \bot$, $from\ X \to X : s\ X : \bot$ or $take\ (s\ (s\ z))\ (X\ |!\ :\ s\ X\ |!\ :\ \bot) \to X : s\ X : [\,]$.

By definition, we say that an approximation statement $e \to t$ is *valid* in $\mathcal{I}$ iff $e \to t$ can be proved in the calculus $SC_{\mathcal{I}}$ consisting of the SC rules BT, RR and DC together with the rule $FA_{\mathcal{I}}$ below, whose rôle is similar to the combination of the two SC rules AR and RA:

$$\mathbf{FA}_{\mathcal{I}} \quad \frac{e_1 \to t_1\ \ldots\ e_n \to t_n \quad s\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \to t} \quad \begin{array}{l} t \text{ pattern},\ t \neq \bot, s \text{ pattern} \\ f\ \overline{t}_n \to s \in \mathcal{I} \end{array}$$

For instance, the approximation statement $take\ (s\ (s\ z))\ (from\ X) \to X : s\ X : [\,]$ is valid in the intended model of our running example program. For any basic fact $f\ \overline{t}_n \to t$ and any Herbrand interpretation $\mathcal{I}$, it can be shown that $f\ \overline{t}_n \to t$ is valid in $\mathcal{I}$ iff $f\ \overline{t}_n \to t \in \mathcal{I}$. The *denotation* of $e \in Exp_\bot$ in $\mathcal{I}$ is defined as the set: $[\![e]\!]^{\mathcal{I}} = \{t \in Pat_\bot \mid e \to t \text{ valid in } \mathcal{I}\}$. Given a program P without bugs, the intended model $\mathcal{I}$ should be a model of P. This relies on the following definition of model, which generalizes the corresponding notion from logic programming:

- $\mathcal{I}$ is a model for P ($\mathcal{I} \models P$) iff $\mathcal{I}$ is a model for every program rule in P.
- $\mathcal{I}$ is a model for a program rule $l \to r \Leftarrow C$ ($\mathcal{I} \models l \to r \Leftarrow C$) iff for any substitution $\theta \in Subst_\bot$, $\mathcal{I}$ satisfies $l\theta \to r\theta \Leftarrow C\theta$.
- $\mathcal{I}$ satisfies a rule instance $l' \to r' \Leftarrow C'$ iff either $\mathcal{I}$ does not satisfy C' or $[\![l']\!]^{\mathcal{I}} \supseteq [\![r']\!]^{\mathcal{I}}$.
- $\mathcal{I}$ satisfies an admissible condition C' iff for any $e' \to p' \in C'$, $[\![e']\!]^{\mathcal{I}} \supseteq [\![p']\!]^{\mathcal{I}}$. It can be shown that $[\![e']\!]^{\mathcal{I}} \supseteq [\![p']\!]^{\mathcal{I}}$ iff $p' \in [\![e']\!]^{\mathcal{I}}$.

A straightforward consequence of the previous definitions is that $\mathcal{I} \not\models P$ iff there exists a program rule $l \to r \Leftarrow C$, $\theta \in Subst_\bot$ and $t \in Pat_\bot$ such that $e\theta \to p\theta$ is valid in $\mathcal{I}$ for any $e \to p \in C$, $r\theta \to t$ is valid in $\mathcal{I}$, but $l\theta \to t \notin \mathcal{I}$. Under these conditions we say that the program rule $l \to r \Leftarrow C$ is *incorrect* w.r.t. the intended model $\mathcal{I}$ and that $(l \to r \Leftarrow C)\theta$ is an *incorrect instance* of the program rule. In our running example, the program rule $from\ X \to X : from\ X$ is incorrect w.r.t. the intended model $\mathcal{I}$, because $X : from\ X \to X : X : \bot$ is valid in $\mathcal{I}$ but $from\ X \to X : X : \bot \notin \mathcal{I}$. By a straightforward adaptation of results given in [5, 4], we can obtain the following relationships between programs and models:

Proposition 2 *Let P be a program and $e \to t$ an approximation statement. Then:*
(a) If $P \vdash e \to t$ then $e \to t$ is valid in any Herbrand model of P.
(b) $M_P = \{f\ \overline{t}_n \to t | P \vdash f\ \overline{t}_n \to t\}$ is the least Herbrand model *of P w.r.t. the inclusion ordering.*
(c) If $e \to t$ is valid in $\mathcal{M}_P$ then $P \vdash e \to t$.

According to these results, the least Herbrand model of a correct program should be a subset of the intended model. This is not the case for our running example, where the approximation statement $take\ (s(s\,z))\ (from\ X) \to X : X : [\,]$ is valid in $\mathcal{M}_P$ but not in the intended model.

4.3 A Goal Solving Calculus

We next present a *Goal Solving Calculus* GSC which formalizes the computation of solutions for a given goal. GSC is inspired by the lazy narrowing calculi from [5, 4], adapted to the modified language in this paper. Since we have no joinability statements here, the rules to deal with them have been omitted. The rules given in [4] to deal with *higher order logic variables* have been also omitted for simplicity; they could be added without any difficulty.

The GSC calculus consists of rules intended to transform a goal step by step. Each step transforms a goal G_{i-1} into a new goal G_i, yielding a substitution σ_i. This is done by selecting an atomic subgoal of G_{i-1} and replacing it by new subgoals according to some GSC rule. Therefore, GSC-rules have the shape $G,\ e \to t,\ G' \Vdash_{\sigma_i} (G, G'', G')\sigma_i$. A GSC computation succeeds when the *empty goal* (represented as $\Box$) is reached. The composition σ of all the substitutions σ_i along a successful computation is called a GSC *computed answer* for the initial goal.

As auxiliary notions, we need to introduce *user-demanded variables* and *demanded variables*. Informally, we say that a variable X is user-demanded if X occurs in t for some atomic subgoal $e \to t$ of the initial goal, or X is introduced by some substitution which binds another user-demanded variable. Formally, let $G_0 = e_1 \to t_1, \ldots\ e_k \to t_k$ be the initial goal, G_{i-1} any intermediate goal, and $G_{i-1} \Vdash_{\sigma_i} G_i$ any calculus step. Then the sets of user-demanded variables ($udvar$) are defined in the following way:

$$udvar(G_0) = \bigcup_{i=1}^{k} var(t_i) \qquad udvar(G_i) = \bigcup_{x \in\, udvar(G_{i-1})} var(x\sigma_i),\ i > 0$$

Let $e \to t$ be an atomic subgoal of a goal G. By definition, a variable X in t is demanded if it is either a user-demanded variable or else there is some atomic subgoal in G of the shape: $X\,\overline{e}_k \to t,\ k > 0$, where t must be also demanded if it is a variable. Now we can present the goal solving rules. Note that the symbol $\Vdash$ is used in those rules which compute no substitution.

DC Decomposition: $G,\ h\ \overline{e}_m \to h\,\overline{t}_m,\ G' \Vdash G,\ e_1 \to t_1, \ldots e_m \to t_m,\ G'$.
OB Output Binding: $G,\ X \to t,\ G' \Vdash_{\{X/t\}} (G,\ G')\{X/t\}$, with t not a variable.
IB Input binding: $G,\ t \to X,\ G' \Vdash_{\{X/t\}} (G,\ G')\{X/t\}$, with t a pattern and either X is a demanded variable or X occurs in $(G,\ G')$.

IIM Input Imitation:
$G,\ h\ \overline{e}_m \to X,\ G' \Vdash_{\{X/h\ \overline{X}_m\}} (G,\ e_1 \to X_1, \ldots, e_m \to X_m,\ G')\{X/h\ \overline{X}_m\}$ with $h\ \overline{e}_m$ not a pattern, $\overline{X}_m$ fresh variables, $h\ \overline{X}_m$ a pattern, and either X is a demanded variable or X occurs in (G, G').

EL Elimination: $G, e \to X,\ G' \Vdash_{\{X/\bot\}} G,\ G'$
if X is not demanded and it does not appear in $(G,\ G')$.

FA Function Application: $G,\ f\ \overline{e}_n\ \overline{a}_k \to t,\ G' \Vdash G, e_1 \to t_1, \ldots, e_n \to t_n,\ C,\ r \to S,$
$S\ \overline{a}_k \to t,\ G'$ where S must be a new variable, $f\ \overline{t}_n \to r \Leftarrow C$ is a variant of a program rule, and t must be demanded if it is a variable.

The GSC rules are intended to be related to the SC rules in a way which will become clear in Subsection 5.1. In particular, the GSC rule FA has been modified w.r.t. the analogous rule in the goal solving calculi from [5, 4], so that it can be related to the combined application of the two SC rules AR and RA. As we did in SC, we introduce an optimized variant FA' of rule FA, for the case $k = 0$:

FA' $G,\ f\ \overline{e}_n\ \to t,\ G' \Vdash G,\ e_1 \to t_1, \ldots, e_n \to t_n,\ C,\ r \to t,\ G'$
where $f\ \overline{t}_n \to r \Leftarrow C$ is a variant of a program rule and t demanded if it is a variable.

As another difference w.r.t. [5, 4], where no particular *selection strategy* was considered, here we view goals as *ordered sequences* of atomic subgoals, and we adopt a *quasi-leftmost selection strategy*. This means to select the leftmost atomic subgoal $e \to t$ for which some GSC rule can be applied. Note that this is not necessarily the leftmost subgoal. Subgoals $e \to X$, where X is a non-demanded variable, may be not eligible at some steps. Instead, they are delayed until X becomes demanded or disappears from the rest of the goal. This formalizes the behaviour of shared suspensions in lazy evaluation. In particular, rule EL takes care of detecting undemanded suspensions, whose formal characterization was missing in previous approaches such as [16]. Below we show the first steps of a GSC computation for the goal $take\ N\ (from\ X) \to Ys$ w.r.t. our running example program. Selected subgoals appear underlined and demanded variables X are marked as $X!$. The composition of all the substitutions yields the computed answer: $\sigma = \{N/s\ (s\ z),\ Y_s/X\!:\!X\!:\![]\}$ (wrong in the intended model, as we have seen already).

$\underline{\text{take N (from X)} \to \text{Ys!}} \Vdash_{(FA')} \underline{\text{N} \to \text{s N'}}$, from X $\to$ X':Xs', X':take N' Xs' $\to$ Ys!
$\Vdash_{(OB),\ \{N/sN'\}} \underline{\text{from X} \to \text{X':Xs'}}$, X':take N' Xs' $\to$ Ys! $\Vdash_{(FA')}$
$\underline{\text{X} \to \text{M}}$, M:from M $\to$ X':Xs', X':take N' Xs' $\to$ Ys! $\Vdash_{(IB)\{M/X\}}$ ... $\Box$

Answers computed by GSC are correct solutions in the sense defined in Subsection 4.1. This *soundness* result is a straightforward corollary of Proposition 3, shown in the next section. Regarding *completeness*, we conjecture that GSC can compute all the answers expected by SC, under the assumption that no application of a free logic variable as a function occurs in the program or in the initial goal. We have not checked this conjecture, which is not important for our present purposes. Completeness results for closely related (but more complex) goal solving calculi are given in [5, 4].

5 Debugging Lazy Narrowing Computations

In this section we introduce an instance of Naish's general scheme [12] for debugging wrong answers in our FLP language. Our computation trees will be called the *abbreviated proof tree* (*APTs* in short). An *APT* is obtained in two phases: first a *SC* proof tree *PT* is built from a successful *GSC* computation. Then the *PT* is simplified obtaining the *APT* tree. We next explain these two phases and establish the correctness of the resulting debugger.

5.1 Obtaining Proof Trees from Successful Computations

Given any *GSC* successful computation $G_0 \Vdash_{\sigma_1} G_1 \Vdash_{\sigma_2} \ldots G_{n-1} \Vdash_{\sigma_n} \square$ with computed answer $\sigma = \sigma_1 \ldots \sigma_n$, we build a sequence of trees $T_0, T_1, \ldots, T_n, T$ as follows:

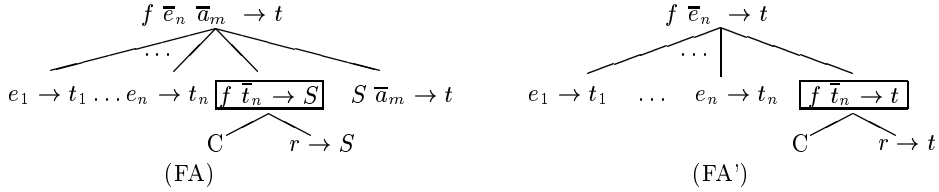
- The only node in T_0 is G_0.
- For any computation step corresponding to a rule of the *GSC* different from *FA* and *FA'*:

$\underbrace{G,\ e \to t,\ G'}_{G_{i-1}} \Vdash_{\sigma_i} \underbrace{(G, G'', G')\sigma_i}_{G_i}$ the tree T_i is built from $T_{i-1}\sigma_i$ by including as children of the leaf $(e \to t)\sigma_i$ in $T_{i-1}\sigma_i$ all the atomic goals in $G''\sigma_i$.

- For any computation step corresponding to rule *FA* of the *GSC*:

$G,\ f\ \overline{e}_n\ \overline{a}_k \ \to t,\ G' \Vdash G,\ e_1 \to t_1, \ldots, e_n \to t_n,\ C,\ r \to S,\ S\ \overline{a}_k \to t,\ G'$

the tree T_i is built by 'expanding' the leaf $f\ \overline{e}_n \overline{a}_k \to t$ of T_{i-1} as shown in the diagram below. Analogously for the case of the simplification of *FA*, i.e. rule *FA'*, a similar diagram can be depicted.



- Finally, the last tree T is obtained from T_n by repeatedly applying the *SC* rule *DC* to its leaves, until no further application of this rule is possible.

For instance, the *PT* of Fig. 3 can be obtained from the *GSC* computation whose first steps have been shown in Subsection 4.3. The next result guarantees that the tree T constructed mechanically in this way is indeed a *PT* showing that the computed answer is a correct solution.

Proposition 3 *The tree T described above is a PT for goal $G_0\sigma$.*

Proof Sketch. By induction on the number of goal solving steps, showing that the algorithm associates a valid *SC* step to each *GSC* rule. This correspondence is:

GSC rule	DC	OB	IB	IIM	EL	FA	FA'
SC rule	DC	-	-	DC	BT	AR+RA	AR'+RA

Rules *IB* and *OB* only apply a substitution and therefore do not correspond to an *SC* inference step. Therefore, the internal nodes of each tree T_i obtained by the algorithm correspond to valid *SC* inferences. Moreover, it can be shown that

the leaves of each T_i either can be proved by repeatedly applying the SC rules DC, BT and RR or occur in G_i. This means that once the empty goal is reached the tree T_n can be completed as indicated to build the final PT. Note that each boxed node in the final PT corresponds to an application of the SC rule RA with an *associated program rule instance*, which comes from some original program rule variant, affected by the computed answer σ.

5.2 Simplifying Proof Trees

The second phase obtains the APT from the PT by removing all the nodes which do not include non-trivial boxed facts, excepting the root. More precisely, let T be the PT for a given goal G. The APT T' of G can be defined recursively as follows:

- The root of T' is the root of T.
- Given any node N in T' the children of N in T' are the closest descendants of N in T that are boxed basic facts $f\ \overline{t}_n \to t$ with $t \neq \bot$.
- Attached to any boxed fact in the APT, an implicit reference to an associated program rule instance is kept. This information is available in the PT.

The idea behind this simplification is that all the removed nodes correspond either to unevaluated function calls (i.e. undemanded suspensions) or to correct computation steps, as they do not rely on the application of any program rule. To complete an instance of Naish's debugging scheme [12], we still have to define a criterion to determine erroneous nodes, and a method to extract a bug indication from an erroneous node. These definitions are the following:

- Given an APT, we consider as erroneous those nodes which contain an approximation statement not valid in the intended model. Note that, with the possible exception of the root node, all the nodes in an APT include basic facts. This simplifies the questions asked to the oracle (usually the user).
- For any buggy node N in the APT, the debugger will show its associated instance of program rule as incorrect.

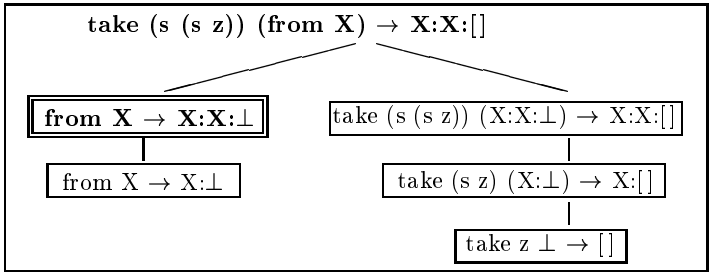


Figure 5: APT corresponding to the PT of Fig. 3

Fig. 5 shows the APT corresponding to the PT of Fig. 3. Erroneous nodes are displayed in bold letters, and the only buggy node appears surrounded by a double box. The relation between this tree and the computation trees used in previous approaches [15–17, 11, 14, 13] has been already discussed in Sect. 2. Assuming a debugger that traverses the tree in preorder looking for a topmost buggy node (see [12] for a discussion about different search strategies when looking for buggy nodes in computation trees), a debugging session could be:

```
from X → X:X:⊥? no
from X → X:⊥? yes
Rule 'from.1' has the incorrect instance 'from X → X:from X'
```

5.3 Soundness and Completeness of the debugger

Now we are in a position to prove the logical correctness of our debugger:

Theorem

(a) Soundness. For every buggy node detected by the debugger, the associated program rule is incorrect w.r.t. the intended model.
(b) Completeness. For every computed answer which is wrong in the intended model, the debugger finds some buggy node.

Proof Sketch

(a) Due to the construction of the APT, every buggy node corresponds to the application of some instance of a program rule R. The node (corresponding to R's left hand side) is erroneous, while its children (corresponding to R's right hand side and conditions) are not. Using these facts and the relation between APTs and PTs, it can be shown that R is incorrect w.r.t. the intended model.
(b) Assuming a wrong computed answer, the root of the APT is not valid in the intended model, and a buggy node must exist because of Proposition 1.

6 Conclusions and Future Work

We have proposed theoretical foundations for the declarative debugging of wrong answers in a simple but sufficiently expressive lazy functional logic language. As in other known debuggers for lazy functional [15–17, 11, 14] and functional logic languages [13, 12], we rely on the generic debugging scheme from [12]. As a novelty, we have obtained a formal characterization of computation trees as *abbreviated proof trees* that relate computed answers to the declarative semantics of programs. Our characterization relies on a formal specification of both the declarative and the operational semantics, using *approximation statements* rather than equations. Thanks to this framework, we have obtained a proof of logical correctness for the debugger, extending older results from the logic programming field to a more complex context. To our best knowledge, no previous work in the lazy functional (logic) field has provided a formalization of computation trees precise enough to prove correctness of the debugger. As additional advantages, our framework helps to simplify oracle questions and supports a convenient representation of functions as values.

As future work we plan an extension of our current proposal, supporting the declarative debugging of both *wrong* and *missing* answers. This will require two different kinds of computation trees, as well as suitable extensions of our logical framework to deal with negative information. We also plan to implement the resulting debugging tools within the $\mathcal{TOY}$ system [10], which uses a *demand driven* narrowing strategy [9, 1] for goal solving. To formalize the generation of computation trees for $\mathcal{TOY}$, we plan to modify the goal solving calculus, so that the demand driven strategy and other language features are taken into account. To implement the generation of computation trees, we plan to follow a transformational approach, adapting techniques described in [16, 14].

Acknowledgement

We are grateful to the anonymous referees for their constructive remarks.

References

1. S. Antoy, R. Echahed, M. Hanus. *A Needed Narrowing Strategy.* 21st ACM Symp. on Principles of Programming Languages, Portland, ACM Press, pp. 268–279, 1994.
2. G. Ferrand. *Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method.* The Journal of Logic Programming 4(3), pp. 177–198, 1987.
3. M. Falaschi, G. Levi, M. Martelli, C. Palamidessi. *A Model-theoretic Reconstruction of the Operational Semantics of Logic Programs.* Information and Computation 102(1). pp. 86-113, 1993.
4. J.C. González-Moreno, M.T. Hortalá-González, M. Rodríguez-Artalejo. *A Higher Order Rewriting Logic for Functional Logic Programming.* Procs. of ICLP'97, The MIT Press, pp. 153–167, 1997.
5. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, M. Rodríguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic*, Journal of Logic Programming 40(1), pp. 47–87, 1999.
6. M. Hanus. *The Integration of Functions into Logic Programming: A Survey.* J. of Logic Programming 19-20, special issue "*Ten Years of Logic Programming*", pp. 583–628, 1994.
7. M. Hanus (ed.), *Curry: an Integrated Functional Logic Language*, Version 0.7, February 2, 2000. Available at `http://www.informatik.uni-kiel.de/~curry/`.
8. J. W. Lloyd. *Declarative Error Diagnosis.* New Generation Computing 5(2), pp. 133–154, 1987.
9. R. Loogen, F.J. López-Fraguas, M. Rodríguez-Artalejo. *A Demand Driven Computation Strategy for Lazy Narrowing.* Procs. of PLILP'93, Springer LNCS 714, pp. 184–200, 1993.
10. F.J. López-Fraguas, J. Sánchez-Hernández. *TOY: A Multiparadigm Declarative System*, in Proc. RTA'99, Springer LNCS 1631, pp 244–247, 1999. Available at `http://titan.sip.ucm.es/toy`.
11. L.Naish. *Declarative dDebugging of Lazy Functional Programs.* Australian Computer Science Communications, 15(1), pp. 287–294, 1993.
12. L. Naish. *A Declarative Debugging Scheme.* J. of Functional and Logic Programming, 1997-3.
13. L. Naish, T. 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, Vol. 2, pp. 91–99, 1995. DSTO General Document 51.
14. L. Naish, T. Barbour. *Towards a Portable Lazy Functional Declarative Debugger.* Australian Computer Science Communications, 18(1), pp. 401–408, 1996.
15. H. Nilsson, P. Fritzson. *Algorithmic Debugging of Lazy Funcional Languages.* The Journal of Functional Programming, 4(3), pp. 337-370, 1994.
16. H. Nilsson, J. Sparud. *The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging.* Automated Software Engineering, 4(2), pp. 121-150, 1997.
17. H. Nilsson. *Declarative Debugging for Lazy Functional Languages.* Ph.D. Thesis. Dissertation No. 530. Univ. Linköping, Sweden. 1998.
18. J. Peterson, K. Hammond (eds.), *Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language*, 1 February 1999.
19. E.Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, Cambridge, Mass., 1982.
20. L. Sterling, E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.