A Logical Framework for the Algorithmic Debugging of Lazy Functional-Logic Programs *

Rafael Caballero, Francisco J. López-Fraguas, and Mario Rodríguez-Artalejo
{rafa,fraguas,mario}@sip.ucm.es

Abstract. Traditional debugging techniques are not well suited for lazy functional programming, because of the difficult-to-predict evaluation order. Therefore, declarative debugging techniques have been proposed, which allow to focus on the intended meaning of programs, abstracting away operational concerns. Similar techniques are known also for logic programming and for combined functional logic languages. The aim of this paper is to provide theoretical foundations for the declarative debugging of wrong answers in lazy functional logic programming. We propose 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 a novelty w.r.t. previous related works, we obtain a completely formal specification of the debugging method. In particular, we extend known soundness and completeness results for the debugging of wrong answers in logic programming to a substantially more difficult context. This work should serve as a clear guideline for a future prototype implementation of a debugging system.

1 Introduction

One of the major adventages of declarative programming is that the control component of a program can be left unspecified. Programmers only have to describe *what* are the characteristics of the problem to be solved, almost dismissing considerations, such as the evaluation strategy, related to *how* the problem is solved actually. As a consequence, the execution flow is hard to predict from a program's text, which makes it problematic to debug declarative programs using conventional debugging tools such as breakpoints, tracing and variable watching. These difficulties are not serious in the case of eager functional programming, where the evaluation order is easy to predict. In the field of lazy functional programming, the problem is well known since the mid eighties. Most work in this area, starting with early proposals as [16], suggests to generate some form of execution record as a basis for debugging. In the field of logic programming, E.Y. 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, 7].

* Work partially supported by the Spanish CICYT (project CICYT-TIC98-0445-C03-02/97 "TREND")

Declarative debugging has been adapted to other programming paradigms, including lazy functional programming [14, 15, 10, 13, 18] and combined functional logic programming [12, 11]. 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 [11], 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, [11] 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. In particular, due to the lack of an adequate formalization of the relationship between computation trees and the operational behaviour of lazy evaluation, no formal correctness proof is available for the existing debuggers of lazy functional (logic) languages.

The aim of this paper is to provide theoretical foundations for the declarative debugging of wrong answers in lazy functional logic programming. Going out from [4, 3], we propose a logical framework to 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 [17]. Then we define a declarative debugger, following the generic scheme from [11]. However, in contrast to [15, 13] and the other related works cited above, we give 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, 7] to a substantially more difficult context. Moreover, our work is intended as a guideline for a future prototype implementation of a working debugger for the functional logic programming system $\mathcal{TOY}$ [9].

The paper is organized as follows. Section 2 presents the general debugging scheme from [11] and recalls some of the known approaches to the declarative debugging of lazy functional and logic programs. Section 3 introduces the simple functional logic language used in the rest of the paper. In Section 4 the logical framework which gives a formal semantics to this language is presented. Section 5 defines the debugger, as well as the formal procedure to build proof trees from succesful computations. Section 6 concludes and points to future work.

2 Debugging with Computation Trees

In this section we revisit the known approaches to declarative debugging of lazy functional and logic languages. First, we recall the generic debugging scheme from [11]. Then we explain how to understand some existing debuggers as instances of the scheme.

2.1 A General Debugging Scheme

The debugging scheme proposed in [11] 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. 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. For instance, consider the computation tree of Figure 1. The erroneous nodes are enclosed in a (maybe double) thick circle. The result of the computation step corresponding to the erroneous node 1 may depend on the result of the erroneous node 2. 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 [11], 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. Given the tree of Figure 1, it is sound to return the buggy node 2 as an erroneous step.

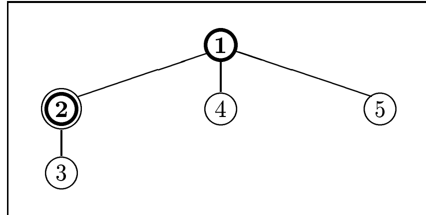


Figure 1: Tree with erroneous and buggy nodes

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. Moreover, reiterated application of the scheme can lead to the detection of more bugs. Coming back to the example, suppose that node 1 corresponds truly to a wrong computation step. In a first stage it is not returned by the debugger because it is erroneous but not buggy. However, after fixing the bug in node 2, node 1 becomes a buggy node which can be detected by using again the debugger.

2.2 Instances of the Debugging Scheme

The known declarative debuggers can be understood as concrete instances of the debugging scheme. Each particular instance is determined by three parameters:

a) a notion of *computation tree*, b) a notion of *erroneous node*, and c) a method to extract a *bug indication* from a buggy node. The choice of these parameters depends on the programming paradigm and the kind of errors to be debugged.

The instances of the debugging scheme needed for diagnosing *wrong* and *missing* answers in pure Prolog are described in [11, 1]. 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, 7].

The existing declarative debuggers for lazy functional [14, 15, 10, 13, 18] and functional logic programs [12, 11] have proposed different, but essentially similar notions of computation tree. Each node contains an oriented equation $f a_1 ... a_n \Rightarrow r$ corresponding to a function call which has been evaluated, together with the returned result. Both the arguments a_i and the result r are displayed in the most evaluated form eventually reached during the computation, but they can still include unevaluated function calls, corresponding to *suspensions* at the implementation level. The children of a such a node correspond to those function calls whose evaluation became eventually needed in order to obtain $f a_1 ... a_n \Rightarrow r$. This 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 which gave rise to the function call in that node. In contrast to the logic programming case, no formal proofs of correctness exist (to our best knowledge) for the known lazy functional (logic) declarative debuggers. To achieve such a proof, one needs a sufficiently formal characterization of the relationship between proof trees and a suitable formalization of program semantics. An attempt to formalize deduction trees for lazy functional programming has been made in [15], using denotational semantics. However, as the authors acknowledge, their definition only gives an informal characterization of the function calls whose evaluation becomes eventually demanded. Our formal procedure for building computation trees supplies this crucial information.

3 The Functional Logic Programming Paradigm

The instance of the general debugging scheme we are going to define will be applied to a functional logic language. The functional logic programming (FLP for short) paradigm tries to bridge the gap between the two main streams in declarative programming: functional programming (FP) and logic programming (LP) (see [5] for a survey). For the purposes of this paper, we have chosen to work with a simple variant of the FLP language studied in [4, 3], and implemented in the $\mathcal{TOY}$ system (see [9]). This choice give us the possibility, adventageous from the point of view of declarative debugging, of exploiting the proof-theoretic and model-theoretic semantics developed in [4, 3], in contrast to other existing FLP languages with a more operational basis, like *Curry* [6]. Our FLP programs support logic variables, higher-order programming, lazy functions and

non-determinism. Horn clause logic programs and Haskell-like functional programs are easily expressible in our language. Next we define some preliminary concepts. Then we present the syntax and the informal meaning of programs and goals 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 *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 the existence of a countable set $\mathcal{V}$ of variables.

The set of *partial expressions* built up with aid of Σ and $\mathcal{V}$ will be denoted as $Exp_\perp$ and defined as: $Exp_\perp ::= \perp \mid X \mid h \mid (e\, e')$ with $X \in \mathcal{V},\ h \in \Sigma,\ e, e' \in Exp_\perp$. 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 $((\ldots(e_0\, e_1)\ldots)\, e_n)$. The symbol $\perp$ (read *bottom*) represents an undefined value.

We distinguish an important kind of partial expressions called *partial patterns*, denoted as $Pat_\perp$ and defined as: $Pat_\perp ::= \perp \mid X \mid c\, t_1 \ldots t_m \mid f\, t_1 \ldots t_m$ where $t_i \in Pat_\perp$, $c \in DC^n$, $0 \leq m \leq n$ and $f \in FS^n$, $0 \leq m < n$. Partial patterns represent partially computed *values* for expressions, and play an important role in the debugging process, since our computation trees will contain statements of the form $f\ t_1 \ldots t_n \to t$, with $t_i, t \in Pat_\perp$, about whose correctness the user will be asked by the debugger. Within the t_i and t, the symbol $\perp$ will appear in place of of those function calls whose evaluation was not needed in order to compute the main result.

Expressions and patterns without any occurrence of $\perp$ are called *total*. We write Exp and Pat for the sets of total expressions and patterns, respectively. Notice that the *partial application* of a constructor or function symbol to a number of pattern arguments less than its arity, is also a pattern, which represents a *functional value*. Such *higher order* patterns are a convenient way of representing (possibly intermediate) HO results of computations (see Appendix A for an example). In other proposals like [14, 15, 13, 18] it is unclear how to deal with functional results. In the rest of the paper we will use the following classification of expressions: $X\ e_1 \ldots e_m$, with $X \in \mathcal{V}$, $m > 0$ is called a *flexible expression*, while $h\, e_1 \ldots e_m$ with $h \in DC^n \cup FS^n$ is called a *rigid expression*. Moreover, a rigid expression is called *active* iff $h \in FS^n$ and $m \geq n$, and *passive* otherwise. The intuition behind this classification is that outermost reduction (at the root position) makes sense only for active expressions.

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 $Subst_\perp$ of all the *partial substitutions* $\theta : \mathcal{V} \to Pat_\perp$ 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 function rules. 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 rule for a function $f \in FS^n$ is:

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

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

(ii) $r, e_1, \dots, e_k$ are expressions.

(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 $e_1, \dots, e_{i-1}$).

Conditions fulfilling property (*iii*) and such that the sequence $p_1, \dots, p_k$, is linear are called *admissible*. 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 $e_j \rightarrow p_j$ are satisfied. The linearity condition over $t_1, \dots, t_n$ is common in FP and FLP; in the latter case, it implies that unification of the arguments of a function call with the patterns of a rule (a part of the narrowing process) can be made without occur-check. Conditions $e_j \rightarrow p_j$ are called *approximation statements* and are satisfied whenever e_j can be evaluated to match the pattern p_j. Variables in the p_j's (called *produced* variables) serve then to get intermediate results of the computation. The linearity condition over $p_1, \dots, p_k$ means that variables can be produced only once, and the condition *(iii)* expresses that variables are produced before being used.

Readers familiar with [4, 3] will note that *joinability conditions* $e \bowtie e'$ (written as $e == e'$ in $\mathcal{TOY}$'s concrete syntax) are replaced by *approximation conditions* $e \rightarrow p$ in this paper. This is done in order to simplify the presentation, while keeping expressivity enough for most programming purposes.

```
(from.1) from N                 → (N : from N)
(take.2) take z     Xs          → [ ]
(take.3) take (s N) [ ]         → [ ]
(take.4) take (s N) (X : Xs)    → (X : take N Xs)
```

Figure 2: A program example

Fig. 2 presents a Haskell-like program that will be used as running example in the rest of the paper. Different rules for the same function f are labeled as $f.i$ where i is the relative position of the rule for the function f. Notice that the conditional part has been omitted, as it is empty for the three rules. The signature of this program is $DC = \{z/0, s/1, :/2\}$, $FS = \{from/1, take/2\}$. The constructors s and z represent the successor of a natural number and the natural number zero, respectively. The infix constructor : builds a list from an element X and another list X_s. Function *take* returns, when applied to a number N and a list X_s, the first N elements of X_s. Function *from* is intended to compute the infinite list of all the numbers starting from an initial value N given as a a parameter. However,

there is a bug, because its right-hand side should be $(N : \mathit{from}\ (\underline{s}\, N))$ instead of $(N : \mathit{from}\ N)$.

A *general goal* is an admissible condition $e_1 \to p_1, \ldots, e_k \to p_k$. An *atomic goal* is a general goal with $k = 1$. By proposing a goal $e \to p$ the user 'asks' whether it exists a substitution θ that makes $(e \to p)\theta$ deducible from the program (the precise meaning of 'deducible' will be established in next section). The possibility of having variables in goals makes a great difference between FLP and FP, even for 'functional' programs like that of Fig. 2. A possible goal for this program could be $\mathit{take}\ N\ (\mathit{from}\ X)) \to Y_s$. The goal solving mechanism of Section 4.3 will produce the two correct answers (w.r.t. the intended meaning of the program) $\theta_1 = \{N/z, Y_s/[\,]\}$, $\theta_2 = \{N/(s\,z), Y_s/X:[\,]\}$ and afterwards the incorrect answer $\theta_3 = \{N/s\,(s\,z),\ Y_s/X:X:[\,]\}$ will be obtained. In a debugger for FP, the user would have to guess values for N and X that lead to a wrong result. In a FLP setting, such values can be found by the system.

4 A Logical Framework for FLP

As explained in Section 1, our debugging scheme is based on a declarative semantics of programs. This is provided by a semantic calculus in the first subsection. In the rest of the section we introduce models of programs and a goal solving calculus which formalizes the operational semantics.

4.1 A Semantic Calculus

The meaning of our programs is specified by a Semantic Calculus (shortly SC) based on the *Higher Order Goal Oriented Rewriting Calculus* ($GORC$) presented in [3]. The purpose of SC is to derive *approximation statements* $e \to t$, intended to mean that "t *approximates* e's value". Notice that the equational meaning "t *equals* e's value" is not the intended one. In the calculus rules, $e, e_i \in Exp_\perp$ are partial expressions, $t_i, t, s \in Pat_\perp$ are partial patterns and $h \in \Sigma$.

Goal Oriented Rewriting Calculus (GORC):

BT Bottom: $$e \to \perp$$

RR Restricted Reflexivity: $$X \to X \qquad X \in Var$$

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

OR Outer Reduction: $$\frac{e_1 \to t_1\ \ldots\ e_n \to t_n \quad C \quad r\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \to t} \qquad f\ \overline{t}_n \to r \Leftarrow C \in [P]_\perp,\ t \neq \perp$$

The notation $[P]_\perp$ in rule OR stands for the set $\{(l \to r \Leftarrow C)\theta | (l \to r \Leftarrow C) \in P, \theta \in Subst_\perp\}$ of partial instances of the rules from P. Notice that OR is the only rule of the calculus that depends on the given program. This rule specifies how to compute a *partial pattern* t as value for the function application $f\ \overline{e}_n\ \overline{a}_k$. It is done by computing suitable *partial patterns* t_i as values for the argument expressions e_i and then applying an instance of a program rule for f. Working with partial patterns here allows to express non-strict semantics with the syntactic simplicity of strict semantics. If $k > 0$, f must be a higher-order function and its result must be applied to the remaining arguments $\overline{a}_k$.

The nodes of our computation trees will include special approximation statements of the form $f\ \overline{t}_n \to t$, with $f \in FS^n$, $t_i, t \in Pat_\perp$, called *basic facts*. In order to make the role of basic facts more explicit, we substitute rule OR of $GORC$ by a new rule FA (Function Application):

$$\textbf{FA}\quad \frac{e_1 \to t_1 \;\ldots\; e_n \to t_n \quad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t}_n \to s}} \quad s\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \to t} \quad \begin{array}{l} f\ \overline{t}_n \to r \Leftarrow C \in [P]_\perp, \\ t \neq \perp \end{array}$$

Our Semantic Calculus SC is hence defined to consist of rules BT, RR, DC and FA. The following result ensures the equivalence between the two calculus.

Proposition 2 *Let P be a program and $G = e \to t$ an approximation statement. Then $e \to t$ is derivable in SC iff it is derivable in $GORC$.*

In the sequel we write $P \vdash e \to t$ to indicate that $e \to t$ can be deduced from P using SC.

Given a program P and an atomic goal $G = e \to t$, we say that a *total* substitution $\theta \in Subst$ is an *answer* for G iff $P \vdash (e \to t)\theta$. For the sake of simplicity we will consider the following variant of FA as part of the SC calculus:

$$\textbf{(FA')}\quad \frac{e_1 \to t_1 \;\ldots\; e_n \to t_n \quad \dfrac{C \quad r \to t}{\boxed{f\ \overline{t}_n \to t}}}{f\ \overline{e}_n \to t} \quad f\ \overline{t}_n \to r \Leftarrow C \in [P]_\perp,\ t \neq \perp$$

It can be shown that SC with FA' is equivalent to SC without FA'.

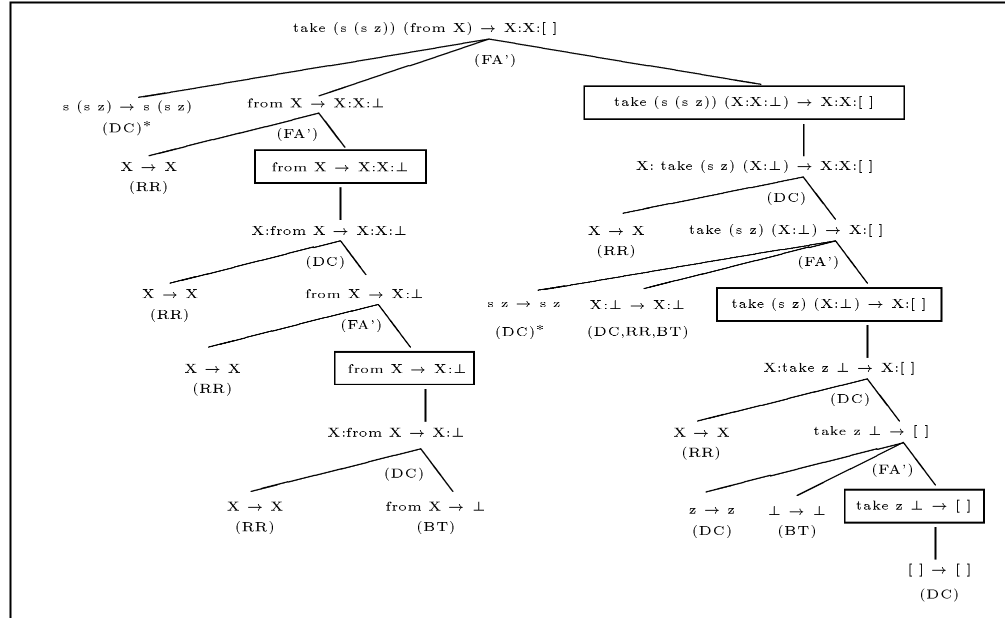


Figure 3: Proof Tree in the semantic calculus SC

A derivation of $e\theta \to t\theta$ in the semantic calculus SC can be represented as a tree, which we will call a *proof tree* (PT) for $G\theta$. The proof tree from Fig. 3 shows that

the approximation statement $take\,(s\,(s\,z))(from\ X) \to X : X : [\,]$ follows from the program P displayed in Fig. 2. The nodes appear decorated with the name of the SC rule (or sequence of rules) used at that point. Therefore the substitution $\theta = \{N/s\,(s\,z), Y_s/X : X : [\,]\}$ is an answer for the goal $take\,N(from\ X) \to Y_s$ since $P \vdash (take\,N(from\ X) \to Y_s)\theta$, showing a symptom of a bug in the program. In fact, the user expects the second element of the list $from\,X$ to be $(s\,X)$. Any basic fact in a node *FA* has an *associated rule instance* which is the instance of the program rule used at this node.

4.2 Models

Intended models of logic programs, as used in [2, 7], can be represented as sets of atomic formulas belonging to the programs Herbrand base. The *open Herbrand universe* (i.e. the set of terms with variables) gives raise to a more informative semantics. In our FLP setting, a natural analogous to the open Herbrand universe is the set $Pat_\perp$ of all the partial patterns, equipped with the approximation ordering: $t \sqsubseteq t' \Longleftrightarrow_{\text{def.}} t' \sqsupseteq t \Longleftrightarrow_{\text{def.}} \emptyset \vdash_{SC} t' \to t$. Similarly, a natural analogous to the open Herbrand base is the collection of all the basic facts $f\ \overline{t}_n \to t$. Therefore, we can define a *Herbrand interpretation* as a set $\mathcal{I}$ of basic facts fulfilling the following three requirements for all $f \in FS^n$ and arbitrary partial patterns $t, \overline{t}_n$:

- $f\,\overline{t}_n \to \perp\ \in \mathcal{I}$.
- 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 $f\ \overline{t}_n \to t\ \in \mathcal{I}$, $\theta \in Subst_\perp$ then $(f\ \overline{t}_n \to t)\theta \in \mathcal{I}$.

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 $from\ X \to \perp$, $from\ X \to (X :\perp)$, $from\ X \to X : s\,X :\perp$ or $take\,(s(s\,z))(X : s\,X :\perp) \to X : s\,X : [\,]$.

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:

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

For instance, the approximation condition $take\,(s\,(s\,z))\,(from\,X) \to X : s\,X : [\,]$ is valid in the intended model $\mathcal{I}$ of our running example. For basic facts $f\ \overline{t}_n \to t$, it turns out 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 $e \in Exp_\perp$ in $\mathcal{I}$ is defined as the set: $[\![e]\!]^\mathcal{I} = \{t \in Pat_\perp \mid e \to t \text{ valid in } \mathcal{I}\}$. 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 $\theta \in Subst_\perp$, $\mathcal{I}$ 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 instantiated 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 $l \to r \Leftarrow C$, $\theta \in Subst_\perp$ and $t \in Pat_\perp$ such that:
1. $e\theta \to p\theta$ valid in $\mathcal{I}$ for any $e \to p \in C$. 2. $r\theta \to t$ valid in $\mathcal{I}$. 3. $l\theta \to t \notin \mathcal{I}$.
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 rule for function *from* is incorrect w.r.t. the intented interpretation $\mathcal{I}$. An incorrect instance of this rule is $from\,X \to X : from\ X$ (i.e. the rule itself). Indeed $X : from\ X \to X : X :\perp$ is valid in $\mathcal{I}$ but $from\ X \to (X : X :\perp) \notin \mathcal{I}$. This corresponds to items 2. and 3. above, with $(X : X :\perp)$ acting as t. By a straightforward adaptation of results given in [4, 3] we can obtain the following relationships between programs and models:

Proposition 3 *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) $M_P = \{f\ \bar{t}_n \to t | P \vdash f\ \bar{t}_n \to t\}$ 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 agree with the intented 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 valid in the intended model.

4.3 A Goal Solving Calculus

Although the semantic calculus can be used for proving whether an approximation statement $e \to t$ holds in a program, it is not feasible for solving goals, i.e. for finding the answers σ that make $(e \to t)\sigma$ deducible in SC. This is due to two different reasons:
• SC does not *find* answers of the goal. Instead it proves the goal under a *given* substitution.
• The rule FA uses instances of the program rules. But there are infinitely many instances of each program rule and FA does not tell *how* to choose the right one.
Therefore a *goal solving calculus GSC* is presented below. It is based on the *Higher Order Lazy Narrowing Calculus* (*HOCLNC*) presented in [3].

The GSC calculus consists of rules which specify how to transform 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. Thus, all the rules of the calculus have the shape $G,\ e \to t,\ G' \Vdash_{\sigma_i} G, G'', G'$,

representing a valid goal solving step. A GSC computation will be successful if it ends in the empty goal (represented as $\Box$). The composition of all the substitutions σ_i yields an answer σ for the initial goal, in the sense of Subsect. 4.1. As auxiliary notions, we need to introduce *user-demanded variables* and *demanded variables*. Intuitively, we say that a variable X is user-demanded if X occurs in t for some condition $e \to t$ 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 set of user-demanded variables ($udvar$) are defined in the following way:

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

Let $e \to t$ 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 if there is any 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: $G,\, t \to X,\, G' \Vdash_{\{X/t\}} G,\, G'$, 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 $h\ \overline{e}_m$ rigid, passive and not 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.

We introduce a variant of rule FA for the case $k = 0$, as we did in the semantic calculus:

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.

We also adopt a suitable strategy in order to determine at each step the atomic subgoal that will be transformed, i.e. the *selected atom*. In this paper we assume the *quasi-leftmost selection strategy:* 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 suspensions in lazy evaluation. We also assume in our execution model that the program rules are tried by rule FA in the same order they appear in the program. Below we show some solving steps for the initial goal $take\,N\,(from\,X) \to Ys$ w.r.t. the program of Figure 2. Selected subgoals appear underlined and demanded variables X are marked as $X!$.

$\underline{\text{take N (from X)} \to \text{Ys!}} \Vdash_{(FA')}$

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

$\underline{\text{from X} \to \text{(X':Xs')}}$, X':take N' Xs' → Ys! $\Vdash_{(FA')}$

$\underline{\text{X} \to \text{M}}$, (M:from M) → (X':Xs'), X':take N' Xs' → Ys! $\Vdash_{(IB)\{M/X\}}$

$\underline{\text{(X:from X)} \to \text{(X':Xs')}}$, X':take N' Xs' → Ys! $\Vdash_{(DC)}$ ... □

The composition of all the substitutions yields a computed substitution σ such that $N\sigma = s\,(s\,z)$, $X\sigma = X$ and $Y_s\sigma = X : X : [\,]$. Adapting similar results from [4, 3], we can prove the *soundness* of the goal solving calculus w.r.t. the semantic calculus from subsection 4.1:

Proposition 4 *Assume that GSC computes an answer substitution σ for the atomic goal $e_0 \to t_0$ using the program P. Then $P \vdash_{SC} (e_0 \to t_0)\sigma$.*

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 yet proved this conjecture. Completeness results for closely related (but more complex) goal solving calculi are given in [4, 3].

5 Debugging Lazy Narrowing Computations

In this section we introduce an instance of the general scheme for debugging wrong answers in our FLP language. As explained in subsection 2.2, this is done by defining a suitable computation tree, which will be called the *abbreviated proof tree* (APT in short), by characterizing erroneous nodes and establishing the method that finally extracts the bug. We also establish the soundness and completeness of the resulting debugger.

5.1 Obtaining Proof Trees from Successful Computations

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

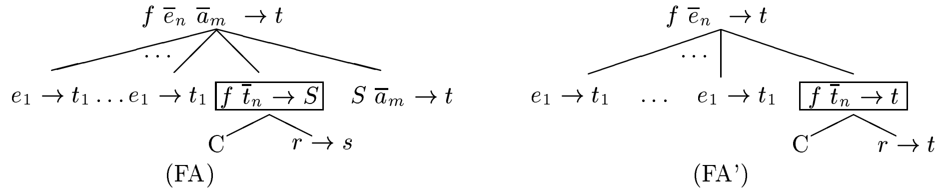
- The only node in T_0 is G_0.
- For any computation step corresponding to a rule of the GSC different from FA and FA':

$\underbrace{G,\ e \to t,\ G'}_{G_{i-1}} \Vdash_{\sigma_i} \underbrace{G, G'', G'}_{G_i}$ 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''.

- 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 simplication 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 rules BT, RR, DC to its leaves, until no further application of these rules is possible.

In the case of our running example, 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 can be deduced from the program. Note that Proposition 4 is a simple corollary of Proposition 5.

Proposition 5 *The tree T described above is a PT for goal $G_0\sigma$.*

5.2 Simplifying Proof Trees

In this second phase we obtain 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 $f\ \overline{t}_n \rightarrow t$ with $t \neq \perp$.

The idea behind this simplification is that all the removed nodes correspond either to unevaluated function calls or to computation steps, as they do not rely on the application of any program rule. To define completely a instance of the general schema we also need to define a criterium 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. This instance has the form $f\ \overline{t}_n \rightarrow r \Leftarrow C$ and its components appear explicitly in the PT (see rule FA in the SC) but not in the APT. Hence it is necessary include this information in each APT node.

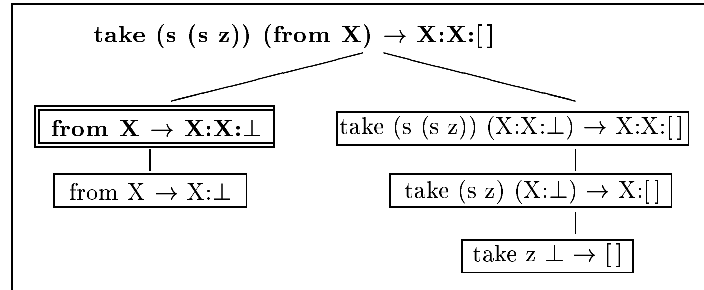


Figure 5: APT corresponding to the PT of Figure 3

Fig. 5 shows the APT corresponding to the PT of our running example. Erroneous nodes are displayed in bold letters, and the only buggy node appears surrounded by a double box. This tree is similar to the computation trees used in [14, 15, 10, 13, 18, 12] except that the statements at its nodes are not interpreted as equalities (see Subsect. 4.1). Assuming a debugger that traverses the tree in preorder looking for a topmost buggy node (see [11, 1] for a discussion about different search strategies when looking for buggy nodes in computation trees), a debugging session could be:

```
from X → X:X:⊥? n
from X → X:⊥? y
Rule from.1 has the incorrect instance from X → X:from X
```

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.

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 [14, 15, 10, 13, 18] and functional logic languages [12, 11], we rely on the generic debugging scheme from [11]. 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. 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 another improvement w.r.t. previous proposals, our framework has the ability to deal with functional values both as arguments and as results of higher order functions.

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 [9]. This will require to deal with some additional language features, and also to consider the operational semantics of $\mathcal{TOY}$, which is based on *demand driven narrowing* [8]. To implement the generation of computation trees, we plan to follow a transformational approach, as described in [15, 13, 18].

References

1. R. Caballero, F.J. López-Fraguas and M. Rodríguez-Artalejo, *A Functional Specification of Declarative Debugging for Logic Programming* in Proceedings of the 8th International Workshop on Functional and Logic Programming. Grenoble, 1999.
2. G. Ferrand. *Error diagnosis in Logic Programming, an adaptation of E.Y. Shapiro's method.* The Journal of Logic Programming, 1987. 4(3):177-198
3. J.C. González-Moreno, T. Hortalá-González, M. Rodríguez-Artalejo. *A Higher Order Rewriting Logic for Functional Logic Programming.* Procs. of ICLP'97, The MIT Press, 153–167, 1997.
4. J.C. González-Moreno, T. Hortalá-González, F.J. López-Fraguas, M. Rodríguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic.* J. of Logic Programming, 40(1), pp 47–87, 1999.
5. M. Hanus. *The Integration of Functions into Logic Programming: A Survey.* J. of Logic Programming 19-20. Special issue "*Ten Years of Logic Programming*", 583–628, 1994.
6. M. Hanus (ed.), *Curry: an Integrated Functional Logic Language*, Version 0.7, February 2, 2000.
7. J. W. Lloyd. *Declarative Error Diagnosis.* New Generation Computing 5(2):133–154, 1987.
8. R. Loogen, F.J. López-Fraguas, M. Rodríguez-Artalejo. *A Demand Driven Computation Strategy for Lazy Narrowing.* Procs. of PLILP'93, Springer LNCS 714, 184–200, 1993.
9. F.J. López Fraguas, J. Sánchez Hernández. *$\mathcal{TOY}$: A Multiparadigm Declarative System*, in Proc. RTA'99, Springer LNCS 1631, pp 244–247, 1999.
10. L. Naish. *Declarative debugging of lazy functional programs.* Australian Computer Science Communications, 15(1):287–294, 1993.
11. L. Naish. *A declarative debugging scheme.* J. of Functional and Logic Programming, 1997-3.
12. L. Naish, 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, Vol. 2, pp. 91–99, 1995. DSTO General Document 51.
13. L. Naish, and Timothy Barbour. *Towards a portable lazy functional delarative debugger.* Australian Computer Science Communications, 18(1):401–408, 1996.
14. H. Nilsson, P. Fritzson. *Algorithmic debugging of lazy funcional languages.* The Journal of Functional Programming, 4(3):337-370, 1994.
15. H. Nilsson and J. Sparud. *The Evaluation Dependence Tree as a Basis for Lazy Funtional Debugging.* Automated Software Engineering, 4(2):121-150, 1997.
16. J.T. O'Donnell and C.V. Hall. *Debugging in applicative languages.* Journal of LISP and Symbolic Computation, 29(1):113–145, 1988.
17. J. Peterson, and K. Hammond (eds.), *Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language*, 1 February 1999.
18. B. Pope. *Buddha. A Declarative Debugger for Haskell.* Honours Thesis, Department of Computer Science, University of Melbourne, Australia, June 1998.
19. E.Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, Cambridge, Mass., 1982.
20. L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.