Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs [1]

Rafael Caballero Roldán Francisco J. López Fraguas
Mario Rodríguez Artalejo

TECHNICAL REPORT SIP - 104-00

Dep. Lenguajes, Sistemas Informáticos y Programación
Univ. Complutense de Madrid

July 2000

[1]Work partially supported by the Spanish CICYT (project CICYT-TIC98-0445-C03-02/97 "TREND".

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 report 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.

Chapter 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 [**?**], suggests to generate some form of execution record as a basis for debugging. In the field of logic programming, E.Y. Shapiro [**?**] 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 [**?**, **?**].

Declarative debugging has been adapted to other programming paradigms, including lazy functional programming [**?**, **?**, **?**, **?**, **?**] and combined functional logic programming [**?**, **?**]. 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 [**?**], 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, [**?**] 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 report is to provide theoretical foundations for the declarative debugging of

wrong answers in lazy functional logic programming. Going out from [?, ?], 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 [?] and a significant subset of Haskell [?]. Then we define a declarative debugger, following the generic scheme from [?]. However, in contrast to [?, ?] 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 [?, ?] 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 $\mathcal{TOY}$ [?].

The report is organized as follows. Chapter 2 presents the general debugging scheme from [?] and recalls some of the known approaches to the declarative debugging of lazy functional and logic programs. Chapter 3 introduces the simple functional logic language used in the rest of the report. In chapter 4 the logical framework which gives a formal semantics to this language is presented. Chapter 5 defines the debugger, as well as the formal procedure to build proof trees from succesful computations. Chapter 6 concludes and points to future work. A small collection of buggy programs that could be treated by the debugger has been included as Appendix A. Most proofs have been omitted due to the lack of space. They are collected in Appendix B.

Chapter 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 [**?**]. 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 [**?**] 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. 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 [**?**], 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. 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

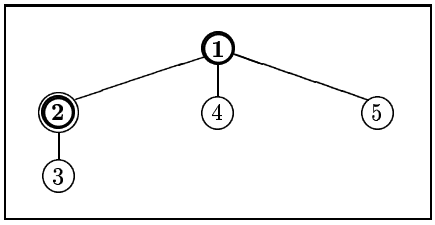


Figure 1: Tree with erroneous and buggy nodes

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 [**?**, **?**]. 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 [**?**, **?**].

The existing declarative debuggers for lazy functional [**?**, **?**, **?**, **?**, **?**] and functional logic programs [**?**, **?**] have proposed different, but essentially similar notions of computation tree. Each node contains an oriented equation $fa_1...a_n \Rightarrow r$ 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 $fa_1...a_n \Rightarrow r$. 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 [**?**], 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.

Chapter 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 [?] for a survey). For the purposes of this report, we have chosen to work with a simple variant of the FLP language studied in [?, ?], and implemented in the $\mathcal{TOY}$ system (see [?]). 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 [?, ?], in contrast to other existing FLP languages with a more operational basis, like *Curry* [?]. 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 $Exp_\bot$ and defined as: $Exp_\bot ::= \ \bot \ | \ X \mid h \mid (e\, e')$ 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 $((\ldots(e_0\, e_1)\ldots)\, e_n)$. 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: $Pat_\bot ::= \ \bot \ | \ X \mid c\, t_1 \ldots t_m \mid f\, t_1 \ldots t_m$ where $t_i \in Pat_\bot$, $c \in DC^n$, $0 \le m \le n$ and $f \in FS^n$, $0 \le m < n$. 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 $f\ t_1 \ldots t_n \rightarrow t$, 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 [**?**, **?**, **?**, **?**] it is unclear how to deal with functional results.

In the rest of the report 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 $\theta : \mathcal{V} \rightarrow Pat$ with a unique extension $\hat{\theta} : Exp \rightarrow Exp$, 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* $\theta : \mathcal{V} \rightarrow Pat_\perp$ 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) $\underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \rightarrow \underbrace{r}_{\text{right-hand side}} \Leftarrow \underbrace{e_1 \rightarrow p_1, \ldots, e_k \rightarrow p_k}_{\text{condition}}$ 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.

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

Conditions fulfilling property (*iii*) and such that the sequence $p_1, \ldots, p_k$, 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 $t_1, \ldots, t_n$ 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

```
(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

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 $p_1, \ldots, p_k$ means that variables can be produced only once, and the condition *(iii)* expresses that variables are produced before being used.

Readers familiar with [**?**, **?**] 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 report. This is done in order to simplify the presentation, while keeping expressivity enough for most programming purposes. Appendix A contains some simple examples. More significant ones can be found in [**?**, **?**]. Fig. 2 presents a Haskell-like program that will be used as running example in the rest of the report. 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 $(N : from\ (\underline{s}\ N))$ instead of $(N : from\ N)$.

3.3 Goals

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 chapter). 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 $take\ N\ (from\ X)) \to Y_s$. The goal solving mechanism of Chapter **??** 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.

Chapter 4

A Logical Framework for FLP

As explained in Chapter **??**, our debugging scheme is based on a declarative semantics of programs. This is provided by a semantic calculus in the first section. In the rest of the chapter 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 [**?**]. 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:	$X \to X$	$X \in Var$
DC Decomposition:	$\dfrac{e_1 \to t_1 \ \ldots \ e_m \to t_m}{h\ \overline{e}_m \to h\ \overline{t}_m}$	$h\ \overline{t}_m \in Pat_\perp$
OR Outer Reduction:	$\dfrac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad C \quad r\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \to t}$	$f\ \overline{t}_n \to r \Leftarrow C \in [P]_\perp,$ $t \neq \perp$

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. In the case $k > 0$, f must be a higher-order function and its result must be applied to the remaining arguments $\overline{a}_k$.

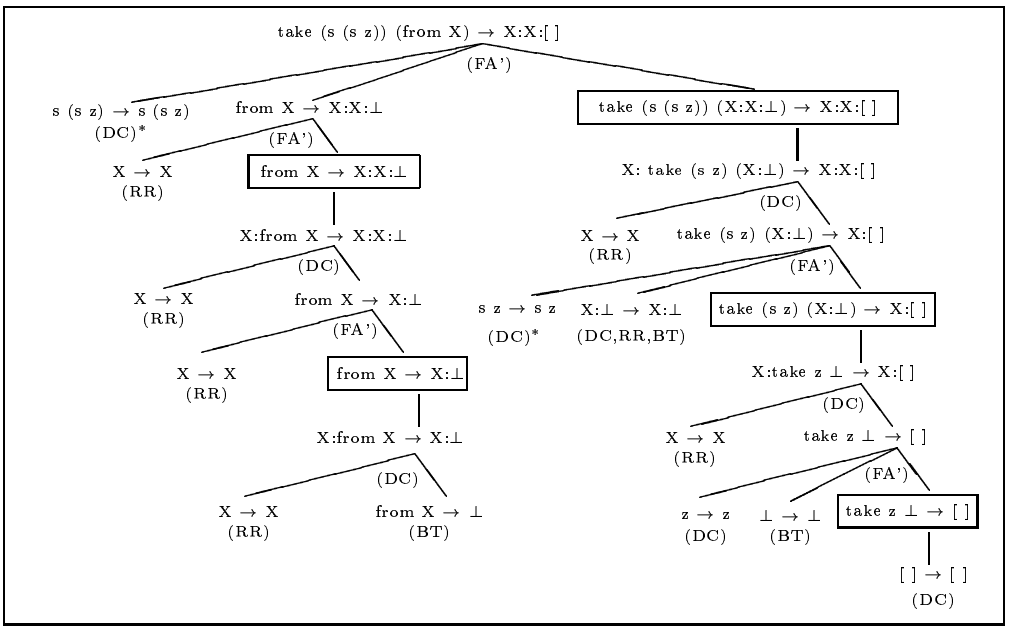


Figure 3: Proof Tree in the semantic calculus SC

The nodes of our computation trees will include special approximation statements of the form $f\ \overline{t}_n \rightarrow t$, 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:

FA Function Application: $$\frac{e_1 \rightarrow t_1\ \ldots\ e_n \rightarrow t_n \quad \dfrac{C \quad r \rightarrow s}{\boxed{f\ \overline{t}_n \rightarrow s}} \quad s\ \overline{a}_k \rightarrow t}{f\ \overline{e}_n\ \overline{a}_k \rightarrow t} \quad f\ \overline{t}_n \rightarrow r \Leftarrow C \in [P]_\perp,\ t \neq \perp$$

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 $G = e \rightarrow t$ an approximation statement. Then $e \rightarrow t$ is derivable in SC iff it is derivable in GORC.*

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

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

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

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

A derivation of $e\theta \rightarrow t\theta$ 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 [**?**, **?**], 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 $Pat_\bot$ 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 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 $t, \overline{t}_n$:

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

This definition of Herbrand interpretation is simpler than the one in [**?**], 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:

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

For instance, the approximation 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 $e \in Exp_\bot$ 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 $\theta \in Subst_\perp$, $\mathcal{I}$ 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 [**?**, **?**] 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) $M_P = \{f\ \overline{t}_n \to t | P \vdash f\ \overline{t}_n \to t\}$ 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 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 [**?**]. The main differences are the following:
• Due to our simpler language we do not need to include rules for dealing with joinability statements. Also, for the sake of a shorter presentation, we have not included rules for higher order logic variables, although adding them does not lead to any problem.
• The rule for function application has been made compatible with its companion rule in SC.
• The calculus includes a particular selection function.

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 section 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. Other strategies for selecting program rules such as *demand driven* strategy [**?**], also known as *needed narrowing* [**?**], are known to improve both the efficiency and termination properties of computations. However, for the sake of simplicity, these strategies are not considered here.

Below we show the solving steps for the initial goal $take\, N\, (from\, X) \rightarrow Ys$ w.r.t. the program of Figure 2. Selected subgoals appear underlined and demanded variables X are marked as $X!$.

take N (from X) → Ys! $\Vdash_{(FA')}$
N → (s N'), from X → X':Xs', X':take N' Xs' → Ys! $\Vdash_{(OB),\ \{N/(s\,N')\}}$
from X → (X':Xs'), X':take N' Xs' → Ys! $\Vdash_{(FA')}$
X → M, (M:from M) → (X':Xs'), X':take N' Xs' → Ys! $\Vdash_{(IB)\{M/X\}}$
(X:from X) → (X':Xs'), X':take N' Xs' → Ys! $\Vdash_{(DC)}$
X → X', from X → Xs', X':take N' Xs' → Ys! $\Vdash_{(IB)\,\{X'/X\}}$
from X → Xs', X:take N' Xs' → Ys! $\Vdash_{(IIM)\,\{Ys/Z:Zs\}}$
from X → Xs', X → Z!, take N' Xs' → Zs! $\Vdash_{(IB)\,\{Z/X\}}$
from X! → Xs', take N' Xs' → Zs! $\Vdash_{(FA')}$
from X! → Xs', N' → (s N"), Xs' → (X":Xs"), (X":take N" Xs") → Zs! $\Vdash_{(OB)\,\{N'/(s\,N'')\}}$
from X! → Xs', Xs' → (X":Xs"), (X":take N" Xs") → Zs! $\Vdash_{(OB)\,\{Xs'/(X'':Xs'')\}}$
from X! → X":Xs", (X":take N" Xs") → Zs! $\Vdash_{(NR')}$
X! → R! $\Vdash_{(IB),\ \{R!/z\}}$
□

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 [**?**, **?**], we can prove the *soundness* of the goal solving calculus w.r.t. the semantic calculus from section 4.1:

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

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 [**?**, **?**].

Chapter 5

Declarative Debugging of Lazy Narrowing Computations

In this chapter we introduce an instance of the general scheme for debugging wrong answers in our FLP language. As explained in chapter **??**, this is done by defining a suitable computation tree, characterizing erroneous nodes and establishing the method that finally extracts the bug. Our notion of computation tree will be called the *abbreviated proof tree* (APT in short). The 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 review these two phases and establish the soundness and completeness of the resulting debugger.

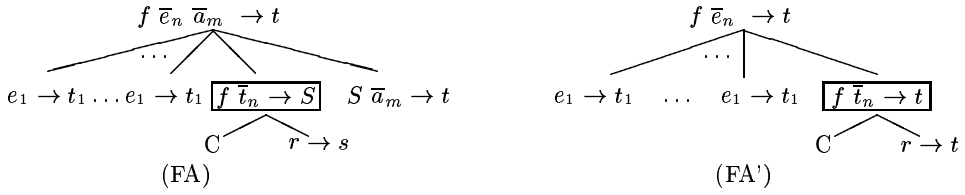
5.1 Obtaining Proof Trees from Successful Computations

Given any GSC successful computation $G_0 \Vdash_{\sigma_1} G_1 \Vdash_{\sigma_2} \ldots G_{n-1} \Vdash_{\sigma_n} \square$ 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 Section **??**. 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 $G_0\sigma$.*
Proof Sketch. The result can be proved by induction on the number of resolution steps, showing that the algorithm associates a valid SC step to each GSC rule. This correspondence is as follows:

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

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 BT, RR and DC 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.

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 $f\ \bar{t}_n \rightarrow t$ 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 $f\ \bar{t}_n \rightarrow r \Leftarrow C$ 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.

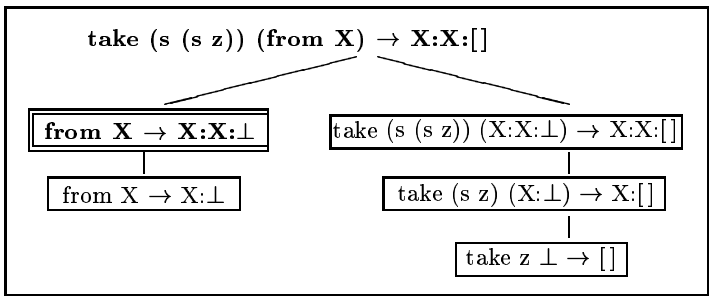


Figure 5: APT corresponding to the PT of Figure 3

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 [**?**, **?**, **?**, **?**, **?**, **?**] 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 [**?**, **?**] 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
```

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 as well as the relation between the APT and the PT, it can be shown that R is incorrect w.r.t. the intented model.
(b) Assuming an wrong computed answer, the root of the *APT* is not valid in the intended model, and a buggy node must exists because of Proposition 1.

Chapter 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 [?, ?, ?, ?, ?] and functional logic languages [?, ?], we rely on the generic debugging scheme from [?]. 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 [?]. 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* [?]. To implement the generation of computation trees, we plan to follow a transformational approach, as described in [?, ?, ?].

Bibliography

[1] S. Antoy, R. Echahed, M. Hanus. *A Needed Narrowing Strategy.* 21st ACM Symp. on Principles of Programming Languages, 268–279, Portland 1994.

[2] 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.

[3] 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

[4] M. Falaschi, G. Levi, M. Martelli, C. Palamidesi. *A model-theoretic reconstruction of the operational semantics of logic programs.* Information and Computation 102(1), 86-113, 1993.

[5] 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.

[6] 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.

[7] 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.

[8] M. Hanus (ed.), *Curry: an Integrated Functional Logic Language*, Version 0.7, February 2, 2000. Available at `http://www-i2.informatik.rwth-aachen.de/ hanus/curry`.

[9] J. W. Lloyd. *Declarative Error Diagnosis.* New Generation Computing 5(2):133–154, 1987.

[10] 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.

[11] F.J. López Fraguas, J. Sánchez Hernández. *TOY: A Multiparadigm Declarative System*, in Proc. RTA'99, Springer LNCS 1631, pp 244–247, 1999.

[12] L. Naish. *Declarative debugging of lazy functional programs.* Australian Computer Science Communications, 15(1):287–294, 1993.

[13] L. Naish. *A declarative debugging scheme.* J. of Functional and Logic Programming, 1997-3.

[14] 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.

[15] L. Naish, and Timothy Barbour. *Towards a portable lazy functional delarative debugger.* Australian Computer Science Communications, 18(1):401–408, 1996.

[16] H. Nilsson, P. Fritzson. *Algorithmic debugging of lazy funcional languages.* The Journal of Functional Programming, 4(3):337-370, 1994.

[17] H. Nilsson and J. Sparud. *The Evaluation Dependence Tree as a Basis for Lazy Funtional Debugging.* Automated Software Engineering, 4(2):121-150, 1997.

[18] J.T. O'Donnell and C.V. Hall. *Debugging in applicative languages.* Journal of LISP and Symbolic Computation, 29(1):113–145, 1988.

[19] J. Peterson, and K. Hammond (eds.), *Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language*, 1 February 1999.

[20] B. Pope. *Buddha. A Declarative Debugger for Haskell.* Honours Thesis, Department of Computer Science, University of Melbourne, Australia, June 1998.

[21] E.Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, Cambridge, Mass., 1982.

[22] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

Appendix A: Some Examples

Example 1
This example is based on a logic program presented in [**?**]:

```
rev  []          →  []
rev  (X:Xs)      →  app (rev Xs) (X:[])
app  []      Y   →  Y
app  (X:Xs)  Y   →  app Xs Y
```

The rule app.2 is erroneous and the goal

```
rev (U:V:[]) → R
```

yields the erroneus answer $\{R/(U:[])\}$.

Example 2
This example shows how the *insertion sort* algorithm can be programmed in our setting, taking adventage of the possibility of defining *non-deterministic* functions.

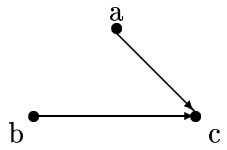
```
insertSort []      → []
insertSort (X:Xs)  → insert X (insertSort Xs)

% non-deterministic function
insert X []        → (X:[])
insert X (Y:Ys)    → (X:Y:Ys) ⇐ X ≤ Y → true
insert X (Y:Ys)    → insert X Ys ⇐ X ≤ Y → false
```

The right hand side of the rule insert.3 should be (Y:insert X Ys). Function $\leq$ can be considered predefined and hence correct. The goal:

```
insertSort (2:1:[]) → R
```

renders the incorrect answer $\{R/(2:[])\}$.

Example 3
This example presents a typical Prolog-like program, defining a graph G and all the possible paths within G. In this example the graph is:



and the program in our language can be written as:

```
arc a c → true
arc b c → true

path X Y → true ⇐ arc X Y→ true
path X Y → true ⇐ arc X Z→ true, path Y Z → true
```

The bug is in rule `path.2`; its right-hand side should be `true` $\Leftarrow$ `arc X Z`$\rightarrow$ `true, path Y Z` $\rightarrow$ `true`. Therefore the following goal erroneously succeeds:

```
path b a → true
```

Example 4
Next example is a Haskell-like program computing the frontier of a given tree T. Function `frontier` is expected to traverse the leaves of T from left to right, collecting them in a list. Trees are represented by constructors `leaf/1` and `node/2`.

```
frontier Tree                      → appendFrontier Tree []
appendFrontier (leaf X)            → (X:)
appendFrontier (node Left Right)   → appendFrontier Right .  appendFrontier Left
(F . G) X                          → F (G X)
```

The auxiliary function `appendFrontier` is intended to append the frontier of a given tree to a given list. However rule `appendFrontier.2` is wrong, its right hand side should be `appendFrontier Left . appendFrontier Right` (swapping `Left` and `Right`). This buggy function is higher-order, since its returns functions as result. Therefore the debugger will ask the oracle about basic facts whose right hand sides can be *higher order patterns*. These questions make sense in our framework, and are crucial to detect the bug in this case. For example, the goal:

```
frontier (node (leaf 0) (leaf 1)) → Xs
```

computes the wrong answer $\{Xs/[1,0]\}$ (the expected answer was $\{Xs/[0,1]\}$). During the debugging process a question such as:

```
appendFrontier (node (leaf 0) (leaf 1) → (1:).(0:)?
```

will be asked to the oracle. Note that `(1:).(0:)` is a higher order pattern.

Example 5
Next example is a program that produces an infinite list containing all the prime numbers:

```
primes           →  sieve (from 2)

from N           →  (N:from (N+1))

sieve (X:Xs)     →  (X:filter (notDiv X) (sieve Xs))

filter P []      →  []
filter P (X:Xs)  →  if P X then (X:filter P Xs)
                    else filter P Xs

notDiv X Y       →  X mod Y > 0

take N []        →  []
take N (X:Xs)    →  if N > 0 then (X:take (N-1) Xs)
                    else []
```

However, due to the mistake in rule `notDiv.1` (it should be `(Y mod X) > 0` the goal

```
take 3 primes → R
```

yields the incorrect answer $\{R/(2:3:4:[])\}$.

Appendix B: Proofs of the main results

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.*
Proof.
Let T be a tree corresponding to a finite computation.
First, by definition a buggy node is erroneous, so a finite computation tree with a buggy node has an erroneous node.

To prove that if T has an erroneous node N then it has a buggy node, we consider the subtree T_0 whose root is node N, and we reason by using induction on the depth of T_0.
Basis: Depth(T_0) = 0. Then N has no children and it is buggy.
Inductive Step: Depth(T_0) = k+1. If N has no erroneous children, then it is buggy. Otherwise it has an erroneous children node N_1. The subtree whose root is N_1 has depth k and by the induction hypothesis has a buggy node.

□

We next introduce two lemmas that will be used to prove the next proposition.

Lemma 1 *Let $u, \overline{v}_k \in Exp_\perp$, $t \in Pat_\perp$ and P a program such that $P \vdash_{SC} u\ \overline{v}_k \to t$. Then there exists $s \in Pat_\perp$ verifying $P \vdash_{SC} u \to s$ and $P \vdash_{SC} s\ \overline{v}_k \to t$.*
Proof.
If $k = 0$ then it suffices to take $s \equiv t$ ($t \to t$ can be proven by repeteadly applying BT, DC and RR). Otherwise, we reason by induction on the depth of a given proof tree T used to prove $u\ \overline{v}_k \to t$:
Basis: (depth(T) = 0). Since $k > 0$, the only possible proof tree of depth 0 must consist of one single appplication of BT. Therefore $t \equiv \perp$, and the result holds trivially with $s \equiv t$, using BT to prove both $P \vdash_{SC} u \to \perp$ and $P \vdash_{SC} s\ \overline{v}_k \to \perp$.

Induction Step: (depth(T) = $l + 1$, $l \geq 0$). We can distinguish two cases according to the SC inference applied at the root of T:

DC: Then $u\ \overline{v}_k \equiv \underbrace{h\ e_1 \ldots e_j}_{u}\ \underbrace{e_{j+1} \ldots e_m}_{\overline{v}_k}$ and $t \equiv h\ \overline{t}_m$. The first SC step must be of the form

$$\frac{e_1 \to t_1\ \ldots\ e_m \to t_m}{h\ \overline{e}_m \to h\ \overline{t}_m}$$

By taking $s \equiv h\, t_1 \ldots t_j$ and considering the SC rule DC, it can be seen readily that

$$\frac{e_1 \to t_1\ \ldots\ e_j \to t_j}{\underbrace{h\ e_1 \ldots e_j}_{u} \to \underbrace{h\ t_1 \ldots t_j}_{s}}$$

and that

$$\frac{t_1 \to t_1\ \ldots\ t_j \to t_j \quad e_{j+1} \to t_{j+1}\ \ldots\ e_m \to t_m}{\underbrace{h\ t_1 \ldots t_j}_{s}\ \underbrace{e_{j+1} \ldots e_m}_{\overline{v}_k} \to \underbrace{h\ t_1 \ldots t_m}_{t}}$$

FA: Then $u\ \overline{v}_k \equiv f\ \overline{e}_n\ \overline{a}_p$, with $f \in FS^n$, and the first SC step will be an application of the FA rule of the form:

$$\frac{e_1 \to t_1\ \ldots\ e_n \to t_n \qquad C \quad r\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_p \to t}$$

with $t \neq \perp$, $f\ \overline{t}_n \to r \Leftarrow C$ a variant of a program rule. We can distinguish two possibilities:

• $u \equiv f\ \overline{e}_i$, with $0 \leq i < n$. In this case we define $s \equiv f\ \overline{t}_i$ ($f\ \overline{t}_i$ is a pattern since i is smaller than the arity of f). Then by using DC we can prove:

$$\frac{e_1 \to t_1\ \ldots\ e_i \to t_i}{\underbrace{f\ \overline{e}_i}_{u} \to \underbrace{f\ \overline{t}_i}_{s}}$$

and by using FA (and repeadtely BT, DC, RR to prove the $t_j \to t_j$), we have that:

$$\frac{t_1 \to t_1\ \ldots\ t_i \to t_i \quad e_{i+1} \to t_{i+1}\ \ldots\ e_n \to t_n \quad C \quad r\ \overline{a}_p \to t}{\underbrace{f\ t_1\ \ldots\ t_j}_{s}\ \underbrace{e_{j+1}\ \ldots\ e_n\ \overline{a}_p}_{\overline{v}_k} \to t}$$

• $u \equiv f\ \overline{e}_n\ \overline{a}_j$, with $0 \leq j \leq k$. In this case we consider the expression $r\ \overline{a}_p \to t$, which can be proved in SC in l steps. The same expression can be written as $(r\ \overline{a_j})(a_{j+1}\ \ldots\ a_p) \to t$. By the inductive hipothesis we have that exists a pattern s such that $r\ \overline{a}_j \to s$ and that $s\ \underbrace{a_{j+1}\ \ldots\ a_p}_{\overline{v}_k} \to t$. Moreover, by using FA we can prove $u \to s$ as follows:

$$\frac{e_1 \to t_1\ \ldots\ e_n \to t_n \quad C \quad r\ \overline{a}_j \to s}{\underbrace{f\ \overline{e}_n\ \ \overline{a}_j}_{u} \to s}$$

□

Lemma 2 *Let $e, \overline{a}_k \in Exp_\perp$, $k \geq 0$, $t, t' \in Pat_\perp$ and P a program such that $P \vdash_{SC} e \to t$ and $P \vdash_{SC} t\ \overline{a}_k \to t'$. Then $P \vdash_{SC} e\ \overline{a}_k \to t'$.*

Proof.

If $t' \equiv \perp$ then the result holds trivially by using rule BT. Assuming t' different from $\perp$ we prove the result by induction on the depth of a proof tree T of $e \to t$.

Basis: (depth(T) 0). There are several possibilities, depending on the SC rule used at the root of T:

BT: Since we are assuming t' not $\perp$ this is not possible.

RR: It must be $e \equiv t \equiv X$, with X a variable. Since by assumption $P \vdash_{SC} X\ \overline{a}_k \to t$ it must be $k = 0$ and $t' \equiv X$ (otherwise no SC rule can be applied to $X\ \overline{a}_k$) and the result holds trivially.

DC: It must be $e \equiv t \equiv h$, with $h \in FS \cup DC$, and again the result is straightforward.

Induction Step: (depth(T)$= k+1,\ k \leq 0$). We can distinguish two cases depending on the SC rule used at the root of T:

DC: Then $e \equiv h\,\overline{e}_m,\ t \equiv h\,\overline{t}_m$, and the first derivation step for $e \rightarrow t$ must be of the form:

$$\frac{e_1 \rightarrow t_1\ \ldots\ e_m \rightarrow t_m}{h\ \overline{e}_m\ \rightarrow h\ \overline{t}_m}$$

We distinguish again two cases depending on the SC rule used to start the derivation of $t\ \overline{a_k} \rightarrow t'$.

DC: It must be $t' \equiv h\ \overline{t'}_m\ \overline{a'}_k$ and

$$\frac{t_1 \rightarrow t'_1\ \ldots\ t_m \rightarrow t'_m\ a_1 \rightarrow a'_1\ \ldots\ a_k \rightarrow a'_k}{h\ \overline{t}_m\ \overline{a_k} \rightarrow h\ \overline{t'}_m\ \overline{a'}_k}$$

Since for each $i = 1, \ldots, m$ we have $P \vdash_{SC}\ e_i \rightarrow t_i$ and $P \vdash_{SC}\ t_i \rightarrow t'_i$, we can deduce by the induction hypothesis that $P \vdash_{SC}\ e_i \rightarrow t'_i$, and then $e\ \overline{a_k} \rightarrow t'$ can be proved by using DC rule:

$$\frac{e_1 \rightarrow t'_1\ \ldots\ e_m \rightarrow t'_m\ a_1 \rightarrow a'_1\ \ldots\ a_k \rightarrow a'_k}{h\ \overline{e}_m\ \overline{a}_k \rightarrow h\ \overline{t'}_m\ \overline{a'}_k}$$

FA: It must be $t\,\overline{a}_k \equiv h\,\overline{t}_m\,\overline{a}_k \equiv f\,\overline{t}_m\,\overline{a}_l\ a_{l+1}\ \ldots\ a_k$, with $h \equiv f \in FS^{m+l}$, and the first derivation step for $t\,\overline{a}_k \rightarrow t'$ must take the form:

$$\frac{e_1 \rightarrow s_1\ \ldots\ t_m \rightarrow s_m\ a_1 \rightarrow s_{m+1}\ \ldots\ a_l \rightarrow s_{m+l}\quad \dfrac{C \quad r \rightarrow s}{\boxed{f\ \overline{s}_{m+l} \rightarrow s}}\quad s\ a_{l+1}\ a_k \rightarrow t'}{f\ \overline{t}_m\ \overline{a}_k \rightarrow t'}$$

In a similar way to the previous case, by the induction hypothesis we deduce that $P \vdash_{SC}\ e_i \rightarrow s_i$ for each $i = 1, \ldots, m$, and then $e\ \overline{a}_k\ \rightarrow t'\ \equiv\ f\ \overline{e}_m\ \overline{a}_k\ \rightarrow t'$ can be proved by using FA rule :

$$\frac{e_1 \rightarrow s_1\ \ldots\ e_m \rightarrow s_m\ a_1 \rightarrow s_{m+1}\ \ldots\ a_l \rightarrow s_{m+l}\quad \dfrac{C \quad r \rightarrow s}{\boxed{f\ \overline{s}_{m+l} \rightarrow s}}\quad s\ a_{l+1}\ a_k \rightarrow t'}{f\ \overline{e}_m\ \overline{a}_k \rightarrow t'}$$

FA: Then $e \equiv f\ \overline{e}_n\ \overline{b}_m$. If t is of the form $\perp$ or $X \in Var$, then the result holds directly because it must be $t' \equiv t$. Thus we only need to prove the case $t \equiv h\ \overline{t}_m,\ m \geq 0$. The derivation step corresponding to the root of the PT for $e \rightarrow t$ must be of the form:

$$\frac{e_1 \rightarrow p_1\ \ldots\ e_n \rightarrow p_n\quad \dfrac{C \quad r \rightarrow s}{\boxed{f\ \overline{p}_n \rightarrow s}}\quad s\ \overline{b}_m\ \rightarrow h\,\overline{t}_m}{f\ \overline{e}_n\ \overline{b}_m\ \rightarrow h\,\overline{t}_m}$$

The proof now proceed in a similar way to the case of the rule DC considered before, by distinguishing two cases (DC, FA) in the SC rule used to start the derivation of $t\ \overline{a_k} \rightarrow t'$.

□

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

- If $e \rightarrow t$ is derivable in SC then it is derivable in $GORC$. Let T_{SC} the proof tree of $e \rightarrow t$ in SC. We show, by induction on the depth of T_{SC} that a proof tree T_{GORC} for $e \rightarrow t$ in $GORC$ can be built.
  Basis: depth(T_{SC})=0. The only SC rule used must either BT or RR or DC with $m = 0$. All these rules are $GORC$ rules as well, and therefore it can be defined $T_{GORC} \equiv T_{SC}$.

  Inductive Step. depth(T_{SC})=0, $n \geq 0$
  We look at the SC rule applied at the root of T_{SC}. There are two possibilities:

  DC As in the basis, the same rule DC exists in $GORC$ and hence can be applied at the root of T_{GORC}. The rest of the tree can be built by the induction hypothesis.

  FA This rule has the shape:

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

  Then we apply rule OR at the root of T_{SC}:

$$\frac{e_1 \rightarrow t_1\ \ldots\ e_n \rightarrow t_n \quad C \quad r\ \overline{a}_k \rightarrow t}{f\ \overline{e}_n\ \overline{a}_k \rightarrow t} \qquad \begin{array}{l} f\ \overline{t}_n \rightarrow r \Leftarrow C \in [P]_\perp, \\ t \neq \perp \end{array}$$

  The $e_i \rightarrow t_i$ can be proven in SC because they appear in the FA rule and their proof trees have to be depth less than or equal to n (induction hypothesis). The same is valid for the conditions C. To show that the $r\,a_k \rightarrow t$ we consider $r \rightarrow s$ and $s\,\overline{a}_k \rightarrow t$. Both approximation statements appear in FA and by induction hypothesis can be proven in SC. Therefore, by the lemma **??** we have that $r\,a_k \rightarrow t$ can be proven in SC.

- If $e \rightarrow t$ is derivable in $GORC$ then it is derivable in SC. Let T_{GORC} the proof tree of $e \rightarrow t$ in $GORC$ we show, by induction on the depth of T_{GORC} that a proof tree T_{SC} for $e \rightarrow t$ in SC can be defined.
  Basis: depth(T_{GORC})=0. A The only $GORC$ rule used must either BT or RR or DC with $m = 0$. All these rules are SC rules as well and therefore ir can be defined $T_{SC} = T_{GORC}$.

  Inductive Step. depth(T_{GORC})=0, $n \geq 0$
  We look at the $GORC$ rule applied at the root of T_{GORC}. There are two possibilities:

DC As in the basis, the same rule DC exists in SC and hence can be applied at the root of T_{SC}. The rest of the tree can be defined by using the induction hypothesis.

OR The OR inference used at the root of T_{GORC} has the shape

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

By induction hypothesis, all the $e_i \to t_i$ as well as C and $r\,\overline{a}_k \to t$ are provable in SC. From the last fact and by Lemma **??** we can find some pattern s such that $r \to s$ and $s\,\overline{a}_k \to t$ are provable in SC. Then we will use rule FA to prove $f\ \overline{e}_n\ \overline{a}_k \to t$ in SC as follows:

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

□

Before proving the next proposition we will introduce some auxiliar lemmas:

Lemma 3 *Let $e \in Exp_\perp$ $t, t' \in Pat_\perp$ and $\mathcal{I}$ a Herbrand interpretation such that $e \to t$ and $t \to t'$ are valid in $\mathcal{I}$. Then $e \to t' \in \mathcal{I}$.*

Proof.

Analogous to the proof of Lemma **??**, simplified to the case of no additional arguments $\overline{a}_k$. The change of rule FA for rule $FA_{\mathcal{I}}$ in $SC_{\mathcal{I}}$ does not affect the essence of the proof.

□

Lemma 4 *Let e an expression and t a pattern and $\mathcal{I}$ a Herbrand interpretation. Then $[\![e]\!]^{\mathcal{I}} \supseteq [\![t]\!]^{\mathcal{I}}$ iff $t \in [\![e]\!]^{\mathcal{I}}$.*

Proof.

It is easy to check that for any Herbrand interpretation $t \to t$ is valid in $\mathcal{I}$ (by applying repeatedly rules DC, RR and BT). Hence, $t \in [\![t]\!]^{\mathcal{I}}$ and if $[\![e]\!]^{\mathcal{I}} \supseteq [\![t]\!]^{\mathcal{I}}$ then $t \in [\![e]\!]^{\mathcal{I}}$.
Conversely, suposse that $t \in [\![e]\!]^{\mathcal{I}}$. Let $t' \in [\![t]\!]^{\mathcal{I}}$. This means that $t \to t'$ and $e \to t$ valid in $\mathcal{I}$. From Lemma **??** for $SC_{\mathcal{I}}$ we have that $e \to t'$ and hence $[\![e]\!]^{\mathcal{I}} \supseteq [\![t]\!]^{\mathcal{I}}$.

□

Lemma 5 *Let $t, t' \in Pat_\perp$, P a program, and $\mathcal{I}$ a Herbrand interpretation. Then $t' \sqsubseteq t$ iff $P \vdash_{CS} t \to t'$ iff $t \to t'$ valid in $\mathcal{I}$.*

Proof.

First, note that, as long as we do not use rules FA or $FA_{\mathcal{I}}$, inferences valid in SC are also valid in $SC_{\mathcal{I}}$ and, conversely, inferences valid in $SC_{\mathcal{I}}$ are valid in SC as well. We do not use either of these rules in this proof and therefore it is only necessary to prove that $t' \sqsubseteq t$ iff $P \vdash_{SC} t \to t'$. In addition, the result holds trivially when $t' \equiv \perp$, so we assume in the rest of the proof that $t' \neq \perp$.

By induction on the structure of t.

Basis:

$t \equiv \bot$: Then $t' \sqsubseteq \bot$ iff $t' \equiv \bot$, and $P \vdash_{SC} \bot \to t'$ iff $t' \equiv \bot$, since the only SC rule that can be applied is BT and this requires $t' \equiv \bot$.

$t \equiv X$: with X any variable.
If $t' \sqsubseteq X$ then t' must be $\bot$ (see above) or X. Then $X \to X$ can be proved in SC by means of rule RR.
If $P \vdash_{SC} X \to t'$ then the only possibility is to apply rule RR and this requires $t' \equiv X$ or rule BT and in this case $t' \equiv \bot$. In both cases $t' \sqsubseteq X$.

$t \equiv h$: Analogous to the previous case, in the case $t' \sqsubseteq h$ this happens iff $t' \equiv \bot$ or $t' \equiv h$, and the same holds if $P \vdash_{SC} h \to t'$.

Inductive Step:
In this case $t \equiv h\ \overline{t}_m$ with $m > 0$.
Then $t' \sqsubseteq h\ \overline{t}_m$ iff $t' \equiv \bot$ or $t' \equiv h\ \overline{t'}_m$ with $t'_i \sqsubseteq t_i$ for all $i = 1 \ldots m$. By the induction hipothesis this means that $P \vdash_{SC} t_i \to t'_i$ for all $i = 1 \ldots m$ and then $h\ \overline{t}_m \to h\ \overline{t'}_m$ can be proved in SC by first applying the DC rule and then using the proofs for the $t_i \to t'_i$.

If, conversely, $P \vdash_{SC} h\,\overline{t}_m \to t'$, then the first SC derivation step must be an application of DC rule which requires t' to have the shape $t' \equiv h\ \overline{t'}_m$ and with every t'_i verifying $t_i \to t'_i$, $i = 1 \ldots m$.

□

Lemma 6 *Let $\mathcal{I}$ a Herbrand interpretation and $f\ \overline{t}_n \to t$ a basic fact. Then $f\ \overline{t}_n \to t$ is valid in $\mathcal{I}$ iff $f\ \overline{t}_n \to t \in \mathcal{I}$.*

Proof.
If $t \equiv \bot$ the result holds because $f\,\overline{t}_n \to \bot$ must be in every Herbrand interpretation and $f\,\overline{t}_n \to \bot$ provable in $\mathcal{I}$ by using the $SC_{\mathcal{I}}$ rule BT. In the rest of the proof we assume that t is not $\bot$.

If $f\ \overline{t}_n \to t \in \mathcal{I}$ then $\mathcal{I} \vdash f\ \overline{t}_n \to t$ as witnessed by the following $SC_{\mathcal{I}}$ derivation, ending with a $FA_{\mathcal{I}}$ step:

$$\frac{t_1 \to t_1\ \ldots\ t_n \to t_n \quad t \to t}{f\ \overline{e}_n \to t} \quad t \text{ pattern},\ t \neq \bot,\ f\ \overline{t}_n \to s \in \mathcal{I}$$

The derivation can be completed by applying repeteadly rules BT, RR and BT to prove all the $t_i \to t_i$ and $t \to t$ as well.

If, conversely, $\mathcal{I} \vdash f\ \overline{t}_n \to t$, this means that there is a proof tree in $SC_{\mathcal{I}}$ of $f\ \overline{t}_n \to t$. The $SC_{\mathcal{I}}$ derivation step corresponding to the root $f\ \overline{t}_n \to t$ must be done by means of $FA_{\mathcal{I}}$. It will be of the form:

$$\frac{t_1 \to t'_1\ \ldots\ t_n \to t'_n \quad s \to t}{f\ \overline{e}_n \to t} \quad t \text{ pattern},\ t \neq \bot,\ f\ \overline{t'}_n \to s \in \mathcal{I}$$

but this means that $t'_1 \sqsubseteq t_1,\ \ldots,\ t'_n \sqsubseteq t_n$ and that $s \sqsubseteq t$ by Lemma **??**. Since $f\ \overline{t'}_n \to s \in \mathcal{I}$ and $\mathcal{I}$ is a Herbrand interpretation this means that $f\ \overline{t}_n \to t \in \mathcal{I}$.

□

Lemma 7 *Let P any program and $M_P = \{f\ \overline{t}_n \rightarrow t | P \vdash_{SC} f\ \overline{t}_n \rightarrow t\}$. Then M_P is a Herbrand interpretation.*

Proof.

M_P must satisfy the three conditions of Herbrand interpretations:

• $f\,\overline{t}_n \rightarrow \bot \in M_P$.

This property holds since $f\,\overline{t}_n \rightarrow \bot$ can be proved in SC by means of the BT rule.

• if $f\ \overline{t}_n \rightarrow t \in M_P$, $t_i \sqsubseteq t'_i, t \sqsupseteq t'$ then $f\,\overline{t'}_n \rightarrow t' \in M_P$.

If $t' \equiv \bot$ then the results holds trivially. Otherwise $f\,\overline{t'}_n \rightarrow t'$ can be proved in SC as follows: First, notice that $t \neq \bot$ since we are assuming $t' \neq \bot$ and by hypothesis $t \sqsupseteq t'$. Also, by hypothesis $f\ \overline{t}_n \rightarrow t$ can be proved in SC. The last inference applied in the proof has to be FA':

$$\frac{t_1 \rightarrow t''_1\ \ldots\ t_n \rightarrow t''_n \qquad \dfrac{C \quad r \rightarrow t}{\boxed{f\ \overline{t''}_n \rightarrow t}}}{f\ \overline{t}_n \rightarrow t} \quad f\ \overline{t''}_n \rightarrow r \Leftarrow C \in [P]_\bot,\ t \neq \bot$$

By using Lemma **??** and the hypothesis $t_i \sqsubseteq t'_i$ for all $i = 1 \ldots n$ we have obtain $P \vdash t'_i \rightarrow t_i$. Moreover, we also have $P \vdash t_i \rightarrow t''_i$. Then by Lemma **??**, $P \vdash_{SC} t'_i \rightarrow t''_i$. Therefore we can prove $f\,\overline{t'}_n \rightarrow t'$ by applying the rule FA' as follows

$$\frac{t'_1 \rightarrow t''_1\ \ldots\ t'_n \rightarrow t''_n \qquad \dfrac{C \quad r \rightarrow t}{\boxed{f\ \overline{t''}_n \rightarrow t'}}}{f\ \overline{t'}_n \rightarrow t'} \quad f\ \overline{t''}_n \rightarrow r \Leftarrow C \in [P]_\bot,\ t' \neq \bot$$

• if $f\ \overline{t}_n \rightarrow t\ \in M_P$, $\theta \in Subst_\bot$ then $(f\ \overline{t}_n \rightarrow t)\theta \in M_P$.

Actually we can prove a more general result: If $P \vdash_{SC} e \rightarrow t$, $\theta \in Subst_\bot$ then $P \vdash_{SC} (e \rightarrow t)\theta$. In order to prove this assume a proof tree of $e \rightarrow t$ and show how to build a proof tree T' for $(e \rightarrow t)\theta$. If $t\theta \equiv \bot$ then the result holds trivially, because we can build T' in one step by using rule BT. Otherwise ($t\theta \neq \bot$) we prove the result by using induction on the depth of T:

Basis. ($depth(T) = 0$). The only possible rules applied at the root of T can be RR or DC (not BT because then $t, t\theta \equiv \bot$, and we are assuming $t\theta \neq \bot$).

RR Then $e \rightarrow t \equiv X \rightarrow X$, with X a variable, and $(e \rightarrow t)\theta \equiv t' \rightarrow t'$, with $t' \in Pat_\bot$. And we can build a SC proof tree T' for $t' \rightarrow t'$ by repeteadly using rules BT, RR and DC.

DC Since $depth(T) = 0$ then $e \rightarrow t \equiv h \rightarrow h$ with $h \in DC \cup FS$. Then $(e \rightarrow t)\theta \equiv h \rightarrow h$ and we can take $T' \equiv T$.

Inductive Step. ($depth(T)=k+1$, $k \geq 0$. The rules applied at the root can be either DC or FA.

DC: In this case $e \to t \equiv h\ \overline{e}_m \to h\ \overline{t}_m$, with $h\ \overline{t}_m$ any pattern, and the SC inference at the root is of the form:

$$\frac{e_1 \to t_1\ \ldots\ e_m \to t_m}{h\ \overline{e}_m \to h\ \overline{t}_m}$$

Therefore $(e \to t)\theta \equiv h\ (\overline{e}_m\theta) \to h\ (\overline{t}_m\theta)$. Also note that each $t_i\theta$ is still a pattern. Hence we can build a SC proof tree T' of the form:

$$\frac{e_1\theta \to t_1\theta\ \ldots\ e_m\theta \to t_m\theta}{(h\ \overline{e}_m \to h\ \overline{t}_m)\theta}$$

where each $e_i\theta \to t_i\theta$ has a SC proof by induction hypothesis.

FA: In this case the SC inference applied at the root to prove $e \to t \equiv f\ \overline{e}_n\ \overline{a}_m \to t$ must be of the form

$$\frac{e_1 \to t_1\ \ldots\ e_n \to t_n \qquad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t}_n \to s}} \qquad s\ \overline{a}_m \to t}{f\ \overline{e}_n\ \overline{a}_m \to t} \qquad \begin{array}{l} s \in Pat_\perp \\ f\ \overline{t}_n \to r \Leftarrow C \in [P]_\perp, \\ t \text{ any pattern},\ t \neq \perp \end{array}$$

The proof tree T' for $(e \to t)\theta \equiv f\ (\overline{e}_n)\theta\ (\overline{a}_m)\theta \to t\theta$ will be rooted by an application of rule FA as follows:

$$\frac{e_1\theta \to t_1\theta\ \ldots\ e_n\theta \to t_n\theta \qquad \dfrac{C\theta \quad r\theta \to s\theta}{\boxed{f\ (\overline{t}_n)\theta \to s\theta}} \qquad s\theta\ (\overline{a}_m)\theta \to t\theta}{f\ (\overline{e}_n)\theta\ (\overline{a}_m)\theta \to t\theta} \qquad \begin{array}{l} s\theta \in Pat_\perp \\ f\ (\overline{t}_n)\theta \to r\theta \Leftarrow C\theta \in [P]_\perp, \\ t\theta \text{ any pattern},\ t\theta \neq \perp \end{array}$$

Such an inference step is correct because:

- We are assuming $t\theta \neq \perp$.
- $s\theta$ is a pattern since s is a pattern.
- $(f\ \overline{t}_n \to r \Leftarrow C)\theta \equiv f\ (\overline{t}_n)\theta \to r\theta \Leftarrow C\theta$ is in $[P]_\perp$ since $f\ \overline{t}_n \to r \Leftarrow C$ is in $[P]_\perp$.

All the approximation statements $e'\theta \to t'\theta$ in the premises of this application of FA correspond to a premise $e \to t$ in the premises of the first derivation step corresponding to the root of T. Hence they have a SC proof tree T_i with $deph(T_i) \leq k$. This allows us to complete T'.

□

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) $M_P = \{f\ \overline{t}_n \to t | P \vdash f\ \overline{t}_n \to t\}$ 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 $P \vdash e \to t$.

Proof.
Let us prove (a), (c) and (b), in this order.

(a) Let $e \to t$ be an approximation statement such that $P \vdash e \to t$ and assume that $\mathcal{J}$ is a Herbrand model of P. Let T be the proof tree of $e \to$ in SC. We will build a proof tree T' of $e \to t$ in $SC_{\mathcal{J}}$, showing that $e \to t$ is valid in $\mathcal{J}$. This is done by using induction on the depth of T.

Basis: (depth(T)= 0). Then $e \to t$ is the only node of T and corresponds either to a BT or to a RR inference. Since these rules are also present in $SC_{\mathcal{J}}$ we can take $T' \equiv T$.

Inductive step: (depth(T)= $k+1$, $k > 0$.). We distinguish different cases depending on the SC rule applied at the root of T.

DC : Then $e \to t \equiv h\ \overline{e}_m \to h\ \overline{t}_m$ the derivation step corresponding to the root of T is:

$$\frac{e_1 \to t_1\ \ldots\ e_m \to t_m}{h\ \overline{e}_m \to h\ \overline{t}_m}$$

Since rule DC also exists in $SC_{\mathcal{J}}$ we can build T' with the same root as T and with the same children at the root. Morover by the induction hipothesis the $e_i \to t_i$ are valid in $\mathcal{J}$ and therefore there exists the proof trees in $SC_{\mathcal{J}}$ that will complete the construction of T'.

FA : In this case $e \to t \equiv f\ \overline{e}_n\ \overline{a}_m \to t$ and the SC inference at the root root will be:

$$\frac{e_1 \to t_1\ \ldots\ e_n \to t_n \qquad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t}_n \to s}} \qquad s\ \overline{a}_m \to t}{f\ \overline{e}_n\ \overline{a}_m \to t}$$

Then we build T' by using rule $FA_{\mathcal{J}}$ at the root:

$$\frac{e_1 \to t_1\ \ldots\ e_n \to t_n \quad s\ \overline{a}_m \to t}{f\ \overline{e}_n\, \overline{a}_m \to t} \quad t \text{ pattern},\ t \neq \bot,\ f\ \overline{t}_n \to s \in \mathcal{J}$$

and completing the tree by means of the $SC_{\mathcal{J}}$ proof trees of the $e_i \to t_i$ and of $s\ \overline{a}_m \to t$, which exist by induction hypothesis, since all these approximation statements also appear at depth k in T. However, we still have to check that the conditions required by $FA_{\mathcal{J}}$ are satisfied. First t is actually a pattern different from $\bot$ because these conditions are also required by rule FA. To see that $f\ \overline{t}_n \to s$ is in $\mathcal{J}$ we observe that $f\ t_1 \ldots t_n \to t \Leftarrow C$ is an instance of some rewrite rule belonging to P. Moreover, $\mathcal{J}$ satisfies C by induction hypothesis. Since $\mathcal{J}$ is a model of P, we can conclude that $[\![f\ t_1 \ldots t_n]\!]^{\mathcal{J}} \supseteq [\![r]\!]^{\mathcal{J}}$. By induction hypothesis we also know that $r \to s$ is valid in $\mathcal{J}$. It follows that $s \in [\![r]\!]^{\mathcal{I}} \subseteq [\![f t_1 \ldots t_n]\!]^{\mathcal{J}}$ and hence $s \in [\![f t_1 \ldots t_n]\!]^{\mathcal{J}}$, as we needed.

(c) $\mathcal{M}_P$ is an Herbrand interpretation as shown in Lemma **??**. Let $e \to t$ valid in $\mathcal{M}_P$ with proof tree T in $SC_{\mathcal{M}_P}$. Then we show by induction on $depth(T)$ that we can build a proof T' in SC for $e \to t$.

Basis (depth(T) = 0). The only possible inferences applied at the root of T are BT, RR or DC. Since these 3 rules are common to SC we can take $T' \equiv T$.

Inductive Step (depth$(T) = k+1$, $k \geq 0$). Then either DC or $FA_{\mathcal{M}_P}$ have been applied at the root of T. In the DC case, the same inference can be applied at the root of T' and by induction hypothesis SC proof trees T_i exist all the children. This completes the desired proof tree T'.

In the $FA_{\mathcal{M}_P}$ case the root inference of T has the form

$$\frac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad s\ \overline{a}_m \to t}{f\ \overline{e}_n\, \overline{a}_m \to t} \quad t \text{ pattern},\ t \neq \bot,\ f\ \overline{t}_n \to s \in \mathcal{M}_P$$

Since $f\ \overline{t}_n \to s \in \mathcal{M}_P$, then there exists a SC proof tree for $f\ \overline{t}_n \to s$. The derivation corresponding to the root of such tree must be of the form:

$$\frac{t_1 \to t'_1 \ \ldots \ t_n \to t'_n \quad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t'}_n \to s}}}{f\ \overline{t}_n \to s} \quad f\ \overline{t'}_n \to r \Leftarrow C \in [P]_\bot,\ t \text{ any pattern},\ s \neq \bot$$

corresponding to a FA' rule. Hence, each $t_i \to t'_i$, C and $r \to s$ have proof trees in SC . We call these trees $T_{e_i \to t_i}$, T_C and $T_{r \to t}$, respectively. Then the tree T' is built by using a FA inference at the root:

$$\frac{e_1 \to t'_1 \ \ldots \ e_n \to t'_n \quad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t'}_n \to s}} \quad s\ \overline{a}_m \to t}{f\ \overline{e}_n\ \overline{a}_m \to t} \quad \begin{array}{l} s \text{ any pattern} \\ f\ \overline{t'}_n \to r \Leftarrow C \in [P]_\bot, \\ t \text{ any pattern},\ t \neq \bot \end{array}$$

To complete T' we can use trees T_C and $T_{r \to t}$ for C and $r \to t$. Moreover, each of the $e_i \to t'_i$ $(i = 1 \ldots n)$ can be proved in SC since:

- $e_i \to t_i$ can be proved in SC by induction hypothesis.
- $t_i \to t'_i$ is proved by the SC proof tree $T_{e_i \to r_i}$.

Therefore, by Lemma **??** $e_i \to t'_i$ can be proved in SC.

(b) Note that $M_P \subseteq \mathcal{I}$ holds for any Herbrand model $\mathcal{I}$ of P, due to the definition of M_P, item (a) of this proposition, and Lemma 6. In order to show that M_P is a model of P, we must prove that M_P satisfies every $f\ \overline{t}_n \to r \Leftarrow C \in [P]_\bot$ (we know already by Lemma **??** that M_P is a Herbrand interpretation). First, suposse that M_P does not satisfy C. Then the result holds. On the contrary, if M_P satisfies C, then we have to prove that $[\![r]\!]^{M_P} \subseteq [\![f\ \overline{t}_n]\!]^{M_P}$. This means that given any $t \in Pat_\bot$ such that $M_P \vdash r \to t$ it should be that $M_P \vdash f\ \overline{t}_n \to t$, i.e. that $P \vdash f\ \overline{t}_n \to t$ (by Lemma **??** and the definition of M_P). If $t \equiv \bot$ then the result holds trivially. If $t \neq \bot$ then $P \vdash f\ \overline{t}_n \to t$ can be proven by using the SC rule FA', as follows:

$$\frac{t_1 \to t_1 \ \ldots \ t_n \to t_n \quad \dfrac{C \quad r \to t}{\boxed{f\ \overline{t}_n \to t}}}{f\ \overline{t}_n \to t} \quad f\ \overline{t}_n \to r \Leftarrow C \in [P]_\bot,\ t \neq \bot$$

To complete the SC proof tree observe that:

- Since M_P satisfies C, we have that each $e_i \to p_i \ \in C$ is valid in M_P, and by the part (c) of this proposition this means that $P \vdash \ e_i \to p_i$.
- $P \vdash r \to t$ holds by assumption.
- Finally the $t_i \to t_i$ can be proved by repeteadly applying rules BT, RR and DC.

□

The proof of the next proposition uses the concept of *incomplete SC proof tree* which is defined first and an auxiliar lemma.

Definition 1 *An incomplete SC proof tree is a tree such that the relationship between any node and its children corresponds to a valid SC inference.*

Lemma 8 *Let T an incomplete SC proof tree and $\sigma \in Subst_\perp$ such that T includes no FA node labelled with an statement $f\ e_1 \ldots e_n\ a_1 \ldots a_k \to Y$ verifying that $Y\sigma = \perp$. Then $T\sigma$ is an incomplete SC proof tree.*

Proof.
We have to check that all the internal nodes of $T\sigma$ correspond to an SC inference. Actually we can prove that they correspond to the *same SC* inference that they do in T. This is done by checking the rule applied at each internal node of T. Such rule only can be DC with $m > 0$ or FA.

DC Then the node considered is of the form $h\ \overline{e}_m \to h\ \overline{t}_m$ and the associated SC inference:

$$\frac{e_1 \to t_1\ \ldots\ e_m \to t_m}{h\ \overline{e}_m \to h\ \overline{t}_m} \quad h\ \overline{t}_m \in Pat_\perp$$

If e_i is an expression then $e_i\sigma$ is also an expression, and if t_i is a pattern then $t_i\sigma$ is also a pattern. Therefore $(h\ \overline{e}_m \to h\ \overline{t}_m)\sigma$ can be proved also by means of a DC inference:

$$\frac{(e_1 \to t_1)\sigma\ \ldots\ (e_m \to t_m)\sigma}{(h\ \overline{e}_m \to h\ \overline{t}_m)\sigma} \quad (h\ \overline{t}_m)\sigma \in Pat_\perp$$

FA The associated SC step must be of the form

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

and analogously to the case of DC we can apply the substitution to all the expressions and patterns appearing in the rule, and apply rule FA also at the node $(f\ \overline{e}_n\ \overline{a}_k \to t)\sigma$ in $T\sigma$. Notice also that if $f\ \overline{t}_n \to r \Leftarrow C \in [P]_\perp$ it happens that $(f\ \overline{t}_n \to r \Leftarrow C)\sigma \in [P]_\perp$ and that $t\sigma \neq \perp$ by assumption.

□

Proposition 4 *Assume a succesful GSC computation for the atomic goal $e_0 \to t_0$ using program P and with answer substitution σ. Then the tree T built by the algorithm described in Subsection 5.1 is a SC proof tree showing that $P \vdash_{SC} (e_0 \to t_0)\sigma$.*

Proof.

We prove this more general result:

Each T_i constructed by the algorithm verifies:
a) T_i is an incomplete SC with root $(\ldots((e_0 \to t_0)\sigma_1)\ldots)\sigma_i$.
b) Each leaf in T_i either appears in G_i or can proved in SC by means of repeteadly applying rules BT, RR, and DC.

As a consequence of this statement, all the leaves in T_n can be proved by using BT, DC, and RR rules, since G_n is the empty goal. Therefore the tree T produced by the algorithm is an SC proof tree for $(e_0 \to t_0)\sigma$ and the proposition holds.

In order to prove (a),(b) we reason by induction on i $(0 \leq i \leq n)$.

Basis. The result holds for T_0. By applying the algorithm $T_0 \equiv G_0 \equiv e_0 \to t_0$ is indeed an incomplete proof tree with root $e_0 \to t_0$, and its only node appears in G_0.

Inductive case. We assume the result for T_{i-1} and prove that it holds for T_i (with $1 \leq i \leq n$). First, by the inductive hypothesis the root of T_{i-1} was $(\ldots((e_0 \to t_0)\sigma_1)\ldots)\sigma_{i-i}$ and hence the root $T_{i-1}\sigma_i$ must be $(\ldots((e_0 \to t_0)\sigma_1)\ldots)\sigma_i$. And, by the algorithm, the root of $T_{i-1}\sigma_i$ is the root of T_i. Notice also that if $e \to t$ is a leaf of T_{i-1} that occurs in G_{i-1} then $(e \to t)\sigma_i$ is a leaf of $T_{i-1}\sigma_i$ occurring in G_i (all the GSC rules have the shape $G, e \to t, G' \Vdash_{\sigma_i} (G, G'', G')\sigma_i$). Finally, since $\sigma_i \in Subst_\perp$, if a leaf of $e \to t$ can be proved by means of rules BT, RR, and DC the same happens with $(e \to t)\sigma_i$.

To complete the proof we distinguish cases according to the goal transformation applied at the step i of the GSC derivation $G, e \to t, G' \Vdash_{\sigma_i} (G, G'', G')\sigma_i$.

DC In this case $e \to t$ must be $h\ \overline{e}_m \to h\ \overline{t}_m$, σ_i is the identity and $T_{i-1}\sigma_i \equiv T_{i-1}$ is an incomplete SC proof tree. Moreover, $h\ e_1 \ldots e_m \to h\ t_1 \ldots t_m$ becomes a node with children $e_j \to t_j$ in T_i. But this corresponds to a DC step in SC and hence T_i is still an incomplete SC proof tree and *a)* holds. Furthermore all the $e_i \to t_i$ are in G_i and also *b*) is satisfied.

OB In this case the GSC derivation has the form: $G, X \to t\ G' \Vdash_{X/t} (G, G')\{X/t\}$, and $T_i \equiv T_{i-1}\sigma_i$. Since $\sigma_i \in Subst$, *a)* holds by Lemma **??**. Also *b)* holds because the leaf $(X \to t)\sigma_i \equiv t \to t$ can be proved by means of the SC rules RR, BT, and DC.

IB Analogous to the case of rule OB.

IIM Then $e \to t \equiv h\ \overline{e}_m \to X$, $\sigma_i = \{X/h\overline{X}_m\}$ and $(e \to t)\sigma_i \equiv h\ \overline{e}_m \to h\overline{X}_m$. By Lemma **??** $T_{i-1}\sigma_i$ is an incomplete SC proof tree. T_i is built from $T_{i-1}\sigma_i$ by expanding node $h\ \overline{e}_m \to h\overline{X}_m$ with children $(e_1 \to X_1)\sigma_i, \ldots, (e_m \to X_m)\sigma_i$. Since this corresponds to an application of the SC rule DC, T_i is still an incomplete SC proof tree. Moreover, all the $(e_j \to X_j)\sigma_i$ occur in G_i and hence part *b)* also holds.

FA In this case $e \to t \equiv f\ \overline{e}_n\ \overline{a}_k \to t$ and σ_i is the indentity. Hence, the only difference between T_{i-1} and T_i is that the leaf $f\ \overline{e}_n\ \overline{a}_k \to t$ of T_{i-1} is an internal node in T_i with children $e_1 \to t_1, \ldots, e_n \to t_n,\ C,\ r \to S,\ S\ \overline{a}_k \to t$. But this corresponds exactly to the application of the SC rule FA, inncluding the conditions $(f\ \overline{t}_n \to r \Leftarrow C)$ and $t \neq \perp$ and the part *a)* of the result holds. Also, all the new leaves occur in G_i and therefore part *b)* of the result holds.

EL Then $e \to t \equiv e \to X$, $\sigma_i = \{X/\perp\}$, and $(e \to X)\sigma_i \equiv e\sigma_i \to \perp$. In this case although $e \to t$ is a leaf of T_{i-1} occurring in G_{i-1}, $(e \to t)\sigma_i$ is a leaf of T_i not included in G_i. However, $(e \to X)\sigma_i \equiv e\sigma_i \to \perp$ and the leaf can be proved in SC by means of rule BT. Therefore part *b)* of the result holds.

To prove part *a)* we apply Lemma **??**. Therefore, we must prove that T_{i-1} has no FA node labelled with an statement $f\ \overline{e}_n\ \overline{a}_k \to X$. We reason by contradiction. Assume that T_{i-1} contains such a node N. Then N must have been introduced by a FA step applied to G_j for some j with $0 \leq j < i-1$. In T_j, node N must include an statement $f\ \overline{e'}_n\ \overline{a'}_k \to Y$ selected in G_j for a FA step, and such that $(f\ \overline{e'}_n\ \overline{a'}_k \to Y)\sigma_{j+1} \ldots \sigma_{i-1} \equiv f\ \overline{e}_n\ \overline{a}_k \to X$. In particular, Y must be a demanded variable in G_j, because this is required by the FA goal transformation. Since Y is demanded in G_j, one of the two cases i) or ii) below must hold. In both cases, we can conclude that X is demanded in G_{i-1}, contradicting the fact that EL has been applied to G_{i-1} with selected subgoal $e \to X$.

i) Y is user demanded. Then X is also user demanded in G_{i-1}.
ii) G_j contains some subgoal of the form $Y\ e_1 \ldots e_p \to s$ with $p > 0$. Then G_{i-1} contains the subgoal $(Y\ e_1 \ldots e_p \to s)\sigma_{j+1} \ldots \sigma_{i-1}$. Since $Y \ldots \sigma_{i-1} \equiv X$, X is demanded in G_{i-1}.

□

The following Lemmas clarify the relationship between PT's and APT's.

Lemma 9 *Let T a PT and N a buggy node in T with children $N_1, \ldots, N_k$. Then N is a boxed basic fact.*

Proof.
First, remember that buggy means not valid in the indented model $\mathcal{I}$ and with all its children valid in $\mathcal{I}$. We distinguish cases according to the SC inference that has introduced node N.

BT Then N has the form $e \to \perp$ and cannot be buggy since it can be proved in $SC_{\mathcal{I}}$ by means of the BT rule.

RR In this case $N \equiv X \to X$, X variable. N cannot be buggy since it can be proved in $SC_{\mathcal{I}}$ by using the RR rule.

DC N must be of the form $N \equiv h\ \overline{e}_m \to h\ \overline{t}_m$ with children $N_1 \equiv e_1 \to t_1,\ \ldots\ N_m \equiv e_m \to t_m$. Since the children are valid in $\mathcal{I}$, by applying rule DC, which is also a $SC_{\mathcal{I}}$ rule, we have that N is valid in $\mathcal{I}$ and hence cannot be buggy.

FA Then the SC step correspondig to this node is:

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

At this point we have to distinguish two possibilities: either $N \equiv f\ \overline{e}_n\ \overline{a}_k \to t$, or $N \equiv \boxed{f\ \overline{t}_n \to s}$. We next prove that it cannot be $N \equiv f\ \overline{e}_n\ \overline{a}_k \to t$. The children nodes of N are $N_1 \equiv e_1 \to t_1,\ \ldots,\ N_n \equiv e_n \to t_n,\ N_{n+1} \equiv \boxed{f\ \overline{t}_n \to s}$, $N_{n+2} \equiv s\ \overline{a}_k \to t$, all of them valid in $\mathcal{I}$. Therefore rule $FA_{\mathcal{I}}$ can be used to prove N in $SC_{\mathcal{I}}$ and N cannot be erroneous.

Hence the only possible buggy nodes are of the form: $N \equiv \boxed{f\ \overline{t}_n \to s}$, i.e. they are boxed basic facts.

□

Lemma 10 *Let T a PT and N a buggy node in T. Then N has an associated incorrect rule instance.*

Proof.

By Lemma **??** node N corresponds to a basic fact and hence is a node of the form $\boxed{f\ \overline{t}_n \to s}$ with children C and $r \to s$ and such that $f\ \overline{t}_n \to r \Leftarrow C \in [P]_\perp$, i.e. $f\ \overline{t}_n \to r \Leftarrow C \equiv (f\ \overline{t}'_n \to r' \Leftarrow C')\sigma$ with $\sigma \in Subst_\perp$ and $f\ \overline{t}'_n \to r' \Leftarrow C'$ a program rule. The incorrect rule instance is then $f\ \overline{t}_n \to r \Leftarrow C$. To check that this is true we have to prove the following three points (see the definition of incorrect program rule, considering here s in the role of t, C in the role of $C\theta$, r in that of $r\theta$, and $f\ \overline{t}_n$ as $l\theta$).

1. $e \to p$ valid in $\mathcal{I}$ for any $e \to p \in C$. This is true because all the $e \to p$ are children of the buggy node $\boxed{f\ \overline{t}_n \to s}$ and hence valid in $\mathcal{I}$.
2. $r \to s$ valid in $\mathcal{I}$. Analogous to the previous point.
3. $f\ \overline{t}_n \to s\ \notin \mathcal{I}$, which is true because $f\ \overline{t}_n \to s$ is buggy and hence erroneous.

□

Lemma 11 *Let T a PT and N a buggy node in T. Let T' the APT built from T by means of the algorithm explained in Subsection 5.2. Then there is a buggy node N' in T', which is as descendant of N (both in T and in T').*

Proof.

By Lemma **??** node N corresponds to a basic fact and by the construction of the T' all the basic facts in T appear in T'. Since N is buggy, it is erroneous in T'. Hence either it is buggy in T' or has a descendant N' in T' which is buggy (see the proof of Proposition **??**. By the construction of the APT N' must also appear in T as a descendant of N.

□

Lemma 12 *Let T a PT and N an erroneous node in T. Let S be the set of the closest descendants of N that are boxed basic facts, and suposse that all the approximation statements in S are valid in $\mathcal{I}$. Then N is a buggy node.*

Proof.

Induction on the depth of T.

Basis. depth$(T) = 0$. In this case N is the only node in T and it must be buggy.

Induction Step. depth$(T) = k + 1$, $k \geq 0$. Suposse that N is not buggy. Then N has a child node N' which is erroneous. Moreover, N' is not a boxed basic fact because then it would be in S and would not be erroneous. Then the set S' of closest descendants of N' that are boxed basic facts must verify $S' \subseteq S$ and therefore the induction hypothesis can be applied, and N' is buggy. But this constradicts Lemma **??**, since N' is not a boxed basic fact. Hence N is buggy.

□

Lemma 13 *Let T a PT, T' its associated APT and N' a buggy node in T'. Then N' is buggy also in T.*

Proof.

By the construction of the APT, N' occurs as an erroneous node in T. Also, since all the children of N' in T' are valid in $\mathcal{I}$, we have by lemma **??** that N' is buggy in T.

□

Finally we can prove the main result; the theorem that states the soundness and completeness of the debugging technique.

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.

(a) Soundness.

Let N be a buggy node detected by the debugger in an APT T'. Let T the associated PT. By Lemma **??** N' is also buggy in T and by Lemma **??** N has an associated incorrect rule instance, which corresponds to an incorrect program rule.

(a) Completeness.

Let $e \rightarrow t$ the initial goal and σ the wrong computed answer. The algorithms explained in the paper build an APT T with root $(e \rightarrow t)\sigma$. Since T has an erroneous root, Proposition **??** can be applied and hence a buggy node will be found.

□