Electronic Notes in Theoretical Computer Science 64 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume64.html 38 pages

A Debugging Scheme for Functional Logic Programs [1]

M. Alpuente [2]

Departamento de Sistemas Informáticos y Computación-DSIC
Technical University of Valencia, Camino de Vera s/n, 46022 Valencia, Spain.

F. J. Correa [3]

Departamento de Informática y Sistemas–DIS
University EAFIT, Carrera 49 7 Sur 50, 3300 Medellín, Colombia.

M. Falaschi [4]

Dipartimento di Matematica e Informatica
University of Udine, Via delle Scienze 206, 33100 Udine, Italy.

Abstract

We present a generic scheme for the declarative debugging of functional logic programs which is valid for eager as well as lazy programs. In particular we show that the framework extends naturally some previous work and applies to the most modern lazy strategies, such as needed narrowing. First we associate to our programs a semantics based on a (continuous) immediate consequence operator, $T_{\mathcal{R}}$, which models computed answers. We show that, given the intended specification of a program $\mathcal{R}$, it is possible to check the correctness of $\mathcal{R}$ by a single step of $T_{\mathcal{R}}$. We consider then a more effective methodology which is based on abstract interpretation: by approximating the intended specification of the success set we derive a finitely terminating diagnosis method, which can be used statically and is parametric w.r.t. to the chosen approximation. In order to correct the bugs, we sketch a preliminary deductive approach which uses example-guided unfolding. We specialize the incorrect rules w.r.t. sets of positive and negative examples which are gathered (bottom-up) during the diagnosis process, so that all refutations of negative examples and no refutation of positive examples are excluded. Our debugging framework does not require the user to either provide error symptoms in advance or answer difficult questions concerning program correctness. We extend an implementation of our system to the case of needed narrowing and illustrate it through some examples which demonstrate the practicality of our approach.

[1] This work has been partially supported by CICYT under grant TIC2001-2705-C03-01, by Acción Integrada Hispano-Italiana HI2000-0161, Acción Integrada Hispano-Alemana HA2001-0059 and by Generalitat Valenciana under grant GV01-424.

[2] Email: alpuente@dsic.upv.es
[3] Email: fcorrea@eafit.edu.co
[4] Email: falaschi@dimi.uniud.it

1 Introduction

Functional logic programming combines the advantages of functional languages (efficient and, for particular subclasses, optimal evaluation strategies) and logic languages (logical variables, built–in search). The operational semantics of integrated languages is usually based on *narrowing*, a combination of unification for parameter passing and reduction as evaluation mechanism which subsumes rewriting and SLD-resolution. Essentially, narrowing consists of the instantiation of goal variables, followed by a reduction step on the instantiated goal. Narrowing is complete in the sense of functional programming (computation of normal forms) as well as logic programming (computation of answers). Due to the huge search space of unrestricted narrowing, steadily improved strategies have been proposed (see [34] for a survey).

Finding program bugs is a long-standing problem in software construction. However, current debugging tools do not enforce program correctness adequately as they do not provide means to finding bugs in the source code w.r.t. the intended program semantics. In pure logic programming, [24,23] defined a declarative framework for debugging which extends the methodology in [29,47], based on using the immediate consequences operator T_P to identify program bugs, to diagnosis w.r.t. computed answers. The framework is goal independent–actually it is driven by a set of "most general" atomic goals–and does not require the determination of symptoms in advance.

In this paper, one of the contributions is to further develop a declarative diagnosis method w.r.t. computed answers which generalizes the ideas of [24] to the diagnosis of functional logic programs [2]. We additionally discuss the problem of modifying incorrect components of the initial program in order to form a practical debugging system. The generalization of [24] is far from trivial, since we have to deal with the extra complexity derived from modelling computed answers while handling with (possibly non–strict and partial) functions, nested calls, and lazy evaluation. Similarly to [32], the possibility to deal with partial funtions and infinite data structures leads to the introduction of two notions of equality, which are characterized by two sets of program rules. From the semantics viewpoint, the resulting construction gets much more elaborate in comparison to the declarative diagnosis scheme proposed in [24]. We associate a (continuous) immediate consequence operator to our programs which models computed answers, and uses the two kinds of equality. Then we show that, given the intended specification $\mathcal{I}$ of a program $\mathcal{R}$, we can check the correctness of $\mathcal{R}$ (w.r.t. computed answers) by a single step of this operator. The conditions which we impose on the programs which we consider allow us to define a framework for declarative debugging which works for both eager (*call–by–value*) narrowing as well as for lazy (*call–by–name*) narrowing. We extend an implementation of our system and show how this methodology can be applied to the debugging of errors in modern, non–strict functional logic languages, by means of a classical transformation which compiles pattern matching into *case expressions*, and we report some experimental benchmarks. We also present an effective methodology which is based on abstract interpretation. We approximate the intended specification of the success set. Following an idea inspired in [24,23,20], we use *over* and *under* specifications $\mathcal{I}^+$ and $\mathcal{I}^-$ to correctly over- (resp. under-) approximate the intended semantics. We then use these two sets respectively for the functions in the premises and the consequence of the immediate consequence operator, and by a simple static test we can determine whether some of the clauses are wrong. Finally, we discuss a (preliminary) methodology for repairing some errors which is based on program specialization by example-guided unfolding.

A preliminary version of this work appeared in [2]. This paper extends and improves the results in [2] as follows. As a first improvement, we made our abstract diagnosis methodology to be parametric w.r.t. the evaluation strategy, which can be either eager (innermost narrowing) or lazy (outermost narrowing). We also corrected some misprints and clarified a number of points w.r.t the previous version. We show how our methodology extends to optimal evaluation strategies such as needed narrowing by using Hanus and Prehofer's transformation in [36], which compiles rules into "case expressions". Our prototype debugging system Buggy has been extended as to work with the different instances of the framework discussed in this paper, which we illustrate by adequate examples. The implementation is endowed with preliminary inductive learning capabilities following our ideas for unfolding–guided correction of programs from automatically generated examples. The positive and negative examples needed for this purpose are automatically derived from the over– and under-approximations of the intended program semantics, whose computation is part of our abstract diagnosis method.

Plan of the paper

The rest of the paper is organized as follows. Section 2 briefly presents some preliminary definitions and notations and recalls some technique for implementing needed narrowing. Section 3 recalls a generic immediate consequence operator $T_{\mathcal{R}}^{\varphi}$ for functional logic program $\mathcal{R}$ which is parametric w.r.t. the narrowing strategy φ which can be either eager or lazy, and formulates a flattening procedure which allows to define an implementation which works for needed narrowing. We then define a fixpoint semantics based on $T_{\mathcal{R}}^{\varphi}$ which correctly models the answers and values computed by a narrower which uses the narrowing strategy φ. We also formulate an operational semantics and we show the correspondence with the least fixpoint semantics. In section 4, we introduce the necessary general notions of incorrectness and insufficiency symptoms and uncovered calls. Section 5 provides an abstract semantics which correctly approximates the fixpoint semantics of $\mathcal{R}$. In Section 6, we present our method of abstract diagnosis. Section 7 discusses how to incorporate into our framework some preliminary techniques for correcting bugs. A prototype implementation of the method is described in Section 8. Section 9 concludes.

We also include an Appendix where we show a debugging session of our prototype implementation, Buggy. More details and missing proofs can be found in [3].

2 Preliminaries

Let us briefly recall some known results about rewrite systems [13,38] and functional logic programming (see [34,37] for extensive surveys). For simplicity, definitions are given in the one-sorted case. The extension to many–sorted signatures is straightforward, see [45]. Throughout this paper, V will denote a countably infinite set of variables and Σ denotes a set of function symbols, or signature, each of which has a fixed associated arity. $\tau(\Sigma \cup V)$ and $\tau(\Sigma)$ denote the non-ground word (or term) algebra and the word algebra built on $\Sigma \cup V$ and Σ, respectively. $\tau(\Sigma)$ is usually called the Herbrand universe ($\mathcal{H}_\Sigma$) over Σ and it will be denoted by $\mathcal{H}$. $\mathcal{B}$ denotes the Herbrand base, namely the set of all ground equations which can be built with the elements of $\mathcal{H}$. A Σ-equation $s = t$ is a pair of terms $s, t \in \tau(\Sigma \cup V)$, or *true*.

Terms are viewed as labelled trees in the usual way. Positions are represented by sequences of natural numbers denoting an access path in a term, where Λ denotes the empty sequence. $\overline{O}(t)$ denotes the set of nonvariable positions of a term t. $t_{|u}$ is the subterm at the position u of t. $t[r]_u$ is the term t with the subterm at the position u replaced with r. These notions extend to sequences of equations in a natural way. For instance, the nonvariable position set of a sequence of equations $g \equiv (e_1, \ldots, e_n)$ can be defined as follows: $\overline{O}(g) = \{i.u \mid u \in \overline{O}(e_i), i = 1, \ldots, n\}$. We use the symbol $\top$ as a generic notation for sequences of the form $true, \ldots, true$. By $Var(s)$ we denote the set of variables occurring in the syntactic object s, while $[s]$ denotes the set of ground instances of s. A *fresh* variable is a variable that appears nowhere else.

A *substitution* is a mapping from the set of variables V into the set of terms $\tau(\Sigma \cup V)$. A substitution θ is more general than σ, denoted by $\theta \leq \sigma$, if $\sigma = \theta\gamma$ for some substitution γ. We write $\theta_{\restriction s}$ to denote the restriction of the substitution θ to the set of variables in the syntactic object s. The *empty substitution* is denoted by ϵ. A *renaming* is a substitution ρ for which there exists the inverse ρ^{-1}, such that $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$. A set equation E is unifiable, if there exists ϑ such that, for all $s = t$ in E, we have $s\vartheta \equiv t\vartheta$, and ϑ is called a *unifier* of E. We let $mgu(E)$ denote the *most general unifier* of the equation set E [41]. We write $mgu(\{s_1 = t_1, \ldots, s_n = t_n\}, \{s'_1 = t'_1, \ldots, s'_n = t'_n\})$ to denote the most general unifier of the set of equations $\{s_1 = s'_1, t_1 = t'_1, \ldots, s_n = s'_n, t_n = t'_n\}$.

A *conditional term rewriting system* (CTRS for short) is a pair $(\Sigma, \mathcal{R})$, where $\mathcal{R}$ is a finite set of reduction (or rewrite) rule schemes of the form $(\lambda \to \rho \Leftarrow C)$, $\lambda, \rho \in \tau(\Sigma \cup V)$, $\lambda \notin V$ and $Var(\rho) \subseteq Var(\lambda)$. The condition C is a (possibly empty) sequence $e_1, \ldots, e_n$, $n \geq 0$, of equations which we handle as a set (conjunction) when we find it convenient. Variables in C that do not occur in λ are called *extra-variables*. We will often write just $\mathcal{R}$ instead of $(\Sigma, \mathcal{R})$. If a rewrite rule has no condition, we write $\lambda \to \rho$. A *goal* is a sequence of equations $\Leftarrow C$, i.e. a rule with no head (consequent). We usually leave out the $\Leftarrow$ symbol when we write goals. For CTRS $\mathcal{R}$, $r \ll \mathcal{R}$ denotes that r is a new variant of a rule in $\mathcal{R}$ such that r contains only *fresh* variables, i.e. contains no variable previously met during computation (standardized apart). Given a CTRS $(\Sigma, \mathcal{R})$, we assume that the signature Σ is partitioned into two disjoint sets $\Sigma = \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{D} = \{f \mid (f(t_1, \ldots, t_n) \to r \Leftarrow C) \in \mathcal{R}\}$ and $\mathcal{C} = \Sigma \setminus \mathcal{D}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions*. The elements of $\tau(\mathcal{C} \cup \mathcal{V})$ are called constructor terms. A *pattern* is a term of the form $f(\bar{d})$ where $f/n \in \mathcal{D}$ and $\bar{d}$ are constructor terms. We say that a CTRS is *constructor–based* (CB) if the left hand sides of $\mathcal{R}$ are patterns.

A rewrite step is the application of a rewrite rule to an expression. A term s *conditionally rewrites* to a term t, $s \to_{\mathcal{R}} t$, if there exist $u \in O(s)$, $(\lambda \to \rho \Leftarrow s_1 = t_1, \ldots, s_n = t_n) \in \mathcal{R}$, and substitution σ such that $s_{|u} \equiv \lambda\sigma$, $t \equiv s[\rho\sigma]_u$, and for all $i \in \{1, \ldots, n\}$ there exists a term w_i such that $s_i \to^*_{\mathcal{R}} w_i$ and $t_i \to^*_{\mathcal{R}} w_i$, where $\to^*_{\mathcal{R}}$ is the transitive and reflexive closure of $\to_{\mathcal{R}}$. When no confusion can arise, we omit the subscript $\mathcal{R}$. A term s is a *normal form*, if there is no term t with $s \to_{\mathcal{R}} t$. The program $\mathcal{R}$ is said to be canonical if the binary one-step rewriting relation $\to_{\mathcal{R}}$ defined by $\mathcal{R}$ is noetherian and confluent [38]. A CTRS $\mathcal{R}$ is *noetherian* if there are no infinite sequences of the form $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} t_3 \to_{\mathcal{R}} \ldots$ A CTRS $\mathcal{R}$ is *confluent* if, whenever a term s reduces to two terms t_1 and t_2, both t_1 and t_2 reduce to the same term.

Functional logic languages are extensions of functional languages with principles derived from logic programming [46]. The computation mechanism of functional logic languages is based on *narrowing*, a generalization of term rewriting where unification replaces matching: both the rewrite rule and the term to be rewritten can be instantiated.

Since unrestricted narrowing has quite a large search space, several strategies to control the selection of redexes have been developed. A *narrowing strategy* (or *position constraint*) is any well-defined criterion which obtains a smaller search space by permitting narrowing to reduce only some chosen positions. A narrowing strategy φ can be formalized as a mapping that assigns a subset $\varphi(g)$ of $\overline{O}(g)$ to every goal g (different from $\top$) such that, for all $u \in \varphi(g)$, the goal g is narrowable at position u. An important property of a narrowing strategy φ is completeness, meaning that the narrowing constrained by φ is still complete. There is an inherited tradeoff coming from functional programming, between the benefits of outside-in evaluation of orthogonal, nonterminating rules and those of inner or eager evaluation with terminating, non orthogonal rules. A survey of results about the completeness of narrowing strategies can be found in [12,11,25,26,34]. To simplify our notation, we let $I\!R_\varphi$ denote the class of programs which satisfy the conditions for the completeness of the strategy φ.

We let *inn*(g) (resp. *out*(g)) denote the narrowing strategy which assigns the position p of the leftmost-innermost (resp. leftmost-outermost) narrowing redex of g to the goal g.[5] We formulate a conditional narrower with strategy φ, $\varphi \in \{inn, out\}$, as the smallest relation $\leadsto_\varphi$ satisfying

$$\frac{u = \varphi(g) \;\wedge\; (\lambda \to \rho \Leftarrow C) \ll \mathcal{R}^{\varphi}_{+} \;\wedge\; \sigma = mgu(\{g_{|u} = \lambda\})}{g \overset{\sigma}{\leadsto}_{\varphi} (C, g[\rho]_u)\sigma}.$$

For $\varphi \in \{inn, out\}$, $\mathcal{R}^{\varphi}_{+} = \mathcal{R} \cup \{Eq^{\varphi}\}$, where Eq^{φ} are the rules which model the equality on terms.

Namely, Eq^{out} is the set rules which define the validity of equations as a *strict equality* between terms which is likely when computations may not terminate [43]:

$$\begin{array}{rcll} c \approx c & \to & true & \%\ c/0 \in \mathcal{C} \\ c(x_1,\ldots,x_n) \approx c(y_1,\ldots,y_n) & \to & (x_1 \approx y_1) \wedge \ldots \wedge (x_n \approx y_n) & \%\ c/n \in \mathcal{C} \\ true \wedge x & \to & x & \end{array}$$

whereas Eq^{inn} is the standard equality defined by:

$$x = x \to true \qquad\qquad \%\ x \in \mathcal{V}$$

We also assume that equations in g and C have the form $s = t$ whenever we consider $\varphi = inn$, whereas the equations have the form $s \approx t$ when we consider $\varphi = out$. Note that a non–strict equation like $f(a) = g(a)$ is not an acceptable goal when $\varphi = out$. In the following, this difference will be made explicit by using $=_\varphi$ to denote the standard equality $=$ of terms whenever $\varphi = inn$, whereas $=_\varphi$ is $\approx$ for the case when φ is *out*.

A successful derivation for g is a derivation $g \overset{\theta}{\leadsto}{}^{*}_{\varphi} \top$, and θ is called a *computed answer substitution* (for g in $\mathcal{R}$).

In [49], it is shown that neither *inn* nor *out* are generally complete. For instance, consider $\mathcal{R} = \{\mathtt{f(y,a)} \to \mathtt{true},\ \ \mathtt{f(c,b)} \to \mathtt{true},\ \ \mathtt{g(b)} \to \mathtt{c}\}$ with input goal $\mathtt{f(g(x),x)} =_\varphi \mathtt{true}$. Then innermost narrowing only computes the answer $\{\mathtt{x/b}\}$ for $\mathtt{f(g(x),x)} = \mathtt{true}$ whereas outermost narrowing only computes $\{\mathtt{x/a}\}$ for the considered goal $\mathtt{f(g(x),x)} \approx \mathtt{true}$. For the completeness of narrowing under the strategies $\varphi = inn, out$, the following *uniformity* condition is required [25,26,31,34,45]: a confluent program is uniform iff the position selected by φ is a valid narrowing position for φ for all substitutions in normal form applied to it. Note that the program $\mathcal{R}$ above does not satisfy the uniformity principle since the top position of the term $\mathtt{f(g(x),x)}$ is not a

[5] An *innermost* term is an operation applied to constructor terms, i.e., a term of the form $f(d_1,\ldots,d_k)$, where $f \in \mathcal{F}$ and for all $i = 1,\ldots,k$, $d_i \in \tau(\mathcal{C} \cup V)$. The leftmost-innermost position of g is the leftmost position of g which points to an innermost subterm. A position p is leftmost-outermost in a set of positions O if there is no $p' \in O$ with either p' prefix of p, or $p' = q.i.q'$ and $p = q.j.q''$ and $i < j$.

valid narrowing position if we apply the substitution $\{\mathtt{x/b}\}$ to this term. A sufficient condition for uniformity in constructor–based, canonical programs is as follows [25]: i) functions in $\mathcal{D}$ are completely defined (i.e. the set of normal ground terms is $\tau(\mathcal{C})$), and ii) lhs's of rules in $\mathcal{R}$ are pairwise *not strictly sub–unifiable*, i.e. two subterms at the same position of two lhs's are not unifiable by a nontrivial *mgu* [34]. For instance, $\mathtt{f(y,a)}$ and $\mathtt{f(c,b)}$ are strictly subunifiable since the *mgu* of the first arguments is the nontrivial substitution $\{\mathtt{y/c}\}$. Since the not strictly sub–unifiable requirement is not satisfied by certain programs, [25] contains a method to transform a program satisfying i) into a program satisfying i) and ii) (see [25] for details). Note that *inn* as well as *out* only compute constructor substitutions in programs which satisfy the conditions for the completeness of the corresponding strategy.

Innermost narrowing is the foundation of several functional logic programming languages like SLOG [30], LPG [14,15] and (a subset of) ALF [33]. Innermost narrowing corresponds to the eager evaluation strategies in functional programming. Modern functional logic languages like Curry [35], Escher [39] and Toy [21] are based on lazy evaluation principles, which delay the evaluation of function arguments until their values are needed to compute some result. This avoids unnecessary computations and allows one to deal with infinite data structures [34].

Needed narrowing is a complete lazy narrowing strategy which is optimal w.r.t. the length of the derivations and the number of computed solutions in inductively sequential (IS) programs, that is, programs such that all its defined functions have a definitional tree. Roughly speaking, a definitional tree for a function symbol f is a tree whose leaves (*rule* nodes) contain all (and only) the rules used to define f and whose inner nodes (*branch* nodes) contain information to guide the (optimal) pattern matching during the evaluation of expressions. Each inner node contains a *pattern* and a variable position in this pattern (the *inductive position*) which is further refined in the patterns of its immediate children by using different constructor symbols. The pattern of the root node is simply $f(\bar{x})$, where $\bar{x}$ is a tuple of different variables. Informally, inductive sequentiality amounts to the existence of discriminating left-hand sides, i.e. typical functional programs. A precise definition of this class of programs and the needed narrowing strategy is based on the notion of a definitional tree [10].

Needed narrowing can be easily and efficiently implemented by translating definitional trees into "case expressions" as proposed in [36], which also proves that there is a strong equivalence of needed narrowing derivations in the original program and leftmost-outermost narrowing derivations in the transformed program. A similar transformation is presented in [50], where inductively sequential programs are translated to *uniform* form, which has only flat rules with pairwise non-subunifiable left-hand sides, where the strong equivalence between needed narrowing and leftmost-outermost narrowing derivations also holds.

The following example illustrates the transformation into case expressions of [36]. Roughly speaking, each inductively sequential function f is transformed into a new function (which is called f too) defined by exactly one rewrite rule, whose left hand side is the term $f(\bar{x})$, with $\bar{x}$ a tuple of distinct variables, and where the corresponding right hand side is a "case construct" representing the definitional tree.

Example 2.1 Consider the following inductively sequential program `double/1` for addition and doubling of natural numbers in unary notation.

$$\begin{aligned}
\mathtt{add(0,x)} &\to \mathtt{x}.\\
\mathtt{add(s(x),y)} &\to \mathtt{s(add(x,y))}.\\
\mathtt{double(0)} &\to \mathtt{0}.\\
\mathtt{double(s(x))} &\to \mathtt{s(s(double(x)))}.
\end{aligned}$$

The rules in this program can be represented by the following definitional trees.

$$\begin{array}{ll}
\mathit{branch}(\mathtt{add(x,y)},1, & \mathit{rule}(\mathtt{add(0,x)} \to \mathtt{x}),\\
& \mathit{rule}(\mathtt{add(s(x),y)} \to \mathtt{s(add(x,y))})).\\
\\
\mathit{branch}(\mathtt{double(x)},1, & \mathit{rule}(\mathtt{double(0)} \to \mathtt{0}),\\
& \mathit{rule}(\mathtt{double(s(x))} \to \mathtt{s(s(double(x)))})).
\end{array}$$

The transformed rules with (desugared) case expressions are as follows:

$$\begin{aligned}
\mathtt{add(x,y)} &\to \mathtt{case_1(x,0,y,s(x1),s(add(x1,y)))}.\\
\mathtt{case_1(0,0,z,_,_)} &\to \mathtt{z}.\\
\mathtt{case_1(s(x),_,_,s(x),z)} &\to \mathtt{z}.\\
\mathtt{double(x)} &\to \mathtt{case_2(x,0,0,s(x1),s(s(double(x1))))}.\\
\mathtt{case_2(0,0,z,_,_)} &\to \mathtt{z}.\\
\mathtt{case_2(s(x),_,_,s(x),z)} &\to \mathtt{z}.
\end{aligned}$$

Note that different *case* functions (which we distinguish by using different subindeces) are needed for case expressions with different patterns. The idea behind the transformation is that, after the pattern matching has been compiled into case expressions, definitional trees are no longer necessary to guide the reduction steps (they are simply driven by the case distinction in the right-hand sides, rhs's, of the rules). Hence, via this transformation we don't lose (much) generality by developing our methodology for the simpler leftmost outermost narrowing; this simplifies reasoning about computations, and consequently proving semantic properties, e.g. strong completeness. We prefer to keep our framework simple while not losing, of course, (much) generality. There is only a slender detail here: Hanus & Prehofer's work does not consider conditional rules explicitly but this is not a substantial characterization for our discussion (it can be amended through "deconditionalization", i.e. the usual encoding of conditions by means of the predefined function "if" [11]).

3 The semantic framework

In order to formulate a semantics for functional logic programs modeling computed answers, the usual Herbrand base has to be extended to the set of all (possibly) non-ground equations [27,28]. $\mathcal{H}_V$ denotes the *V-Herbrand universe* which allows variables in its elements, and is defined as $\tau(\Sigma \cup V)/_{\cong}$, where $\cong$ is the equivalence relation induced by the preorder $\leq$ of "relative generality" between terms. For the sake of simplicity, the elements of $\mathcal{H}_V$ (equivalence classes) have the same representation as the elements of $\tau(\Sigma \cup V)$ and are also called terms. $\mathcal{B}_V$ denotes the *V-Herbrand base*, namely, the set of all equations $s =_\varphi t$ modulo variance, where $s, t \in \mathcal{H}_V$. Note that the standard Herbrand base $\mathcal{B}$ is equal to $[\mathcal{B}_V]$. The ordering on $\mathcal{H}_V$ induces an ordering on $\mathcal{B}_V$, namely $s' = t' \leq s = t$ if $s' \leq s$ and $t' \leq t$. The power set of $\mathcal{B}_V$ is a complete lattice under set inclusion.

In the following, we recall a semantics $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ for program $\mathcal{R}$ given in [2] such that the computed answer substitutions of any (possibly conjunctive) goal g can be derived from $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ by unification of the equations in the goal with the equations in the denotation. We assume that the equations in the denotation are renamed apart. Equations in the goal have to be flattened first, i.e. subterms have to be unnested so that the term structure is directly accessible to unification.

Definition 3.1 [flat goal w.r.t. φ] A *flat* equation is an equation of the form $f(d_1, \ldots, d_n) = d$ or $d_1 =_\varphi d_2$, where $d, d_1, \ldots, d_n \in \tau(\mathcal{C} \cup V)$ are constructor terms. A *flat* goal is a set of flat equations.

Note that, for the outermost strategy $\varphi = out$, a flat goal may contain the two kinds of equality, the strict equality $\approx$ which gives to equality the weak meaning of identity of finite objects as is only defined on finite and completely determined data structures, and the standard (non-strict) equality $=$, which is defined even on partially determined or infinite data structures (see [32,43]). Nevertheless, in a flat goal w.r.t. $\varphi = out$ the only non–strict equations are of the form $f(d_1, \ldots, d_n) = x$. This allows, for example, the elimination of $f(a) = x$, whenever $f(a)$ would not have been selected by narrowing (i.e., when its value is not required to reduce $g(f(a))$), since standard equality $=$ is the only one which obeys the reflexivity axiom $x = x$ for all x. This will be apparent below (Section 3.1), when we introduce two different sets of *reflexivity axioms* $\Im^\varphi_\mathcal{R}$ and $\Phi_\mathcal{R}$, as opposed to [37] where the non-strict case is not considered.

Any sequence of equations E can be transformed into an equivalent one, $flat_\varphi(E)$, which is flat. The *flattening* procedures for equation sets which produce flat goals w.r.t. *inn* and *out*, respectively, can be found in [17,32]. For the sake of completeness we recall them here; however, but we prefer to compact them as two cases of a generic flattening transformation.

Following [32], we present terms as applications of a data context (i.e, a constructor term with some *holes*) to operation–headed terms (i.e, terms with defined function symbols at the outermost level). Namely, t is represented as $e[t_1, \ldots, t_n]$, where e is the external part of t containing constructor symbols only (if any) and $t_1, \ldots, t_n$ are the outermost operation–headed subterms of t. In particular, if t is a operation–headed term then it can be obtained as the application of $[\ \ ]$ (the empty context) to t itself. The following definition is auxiliary.

Definition 3.2 [pre-flattening] The function $flat^{\varphi}_{-}(s)$ for an expression s is defined inductively as follows.

$$flat^{\varphi}_{-}(s) =$$

$$\begin{cases}
flat^{\varphi}_{-}(e_1), \ldots, flat^{\varphi}_{-}(e_n) & \text{if } s \equiv e_1, \ldots, e_n \\[2ex]
\begin{array}{l} flat^{\varphi}_{-}(t_1 = z_1), \ldots, flat^{\varphi}_{-}(t_n = z_n), \\ flat^{\varphi}_{-}(t'_1 = y_1), \ldots, flat^{\varphi}_{-}(t'_m = y_m), \\ e[z_1, \ldots, z_n] =_{\varphi} e'[y_1, \ldots, y_m] \end{array} & \begin{array}{l} \text{if } s \equiv e[t_1, \ldots, t_n] =_{\varphi} \\ \quad e'[t'_1, \ldots, t'_m] \\ \quad z_1, \ldots, z_n, y_1, \ldots, y_m, \\ \quad \text{are fresh variables} \end{array} \\[2ex]
flat^{\varphi}_{-}(f(t_1, \ldots, t_n) = x) & \begin{array}{l} \text{if } s \equiv x =_{\varphi} f(t_1, \ldots, t_n), \\ \quad \text{with } x \in \mathcal{V} \end{array} \\[2ex]
\begin{array}{l} flat^{\varphi}_{-}(t_{1,1} = z_{1,1}), \ldots, flat^{\varphi}_{-}(t_{1,m_1} = z_{1,m_1}), \\ \ldots, flat^{\varphi}_{-}(t_{n,1} = z_{n,1}), \ldots, flat^{\varphi}_{-}(t_{n,m_n} = z_{n,m_n}), \\ f(e_1[z_{1,1}, \ldots, z_{1,m_1}], \ldots, e_n[z_{n,1}, \ldots, z_{n,m_n}]) = x \end{array} & \begin{array}{l} \text{if } s \equiv f(e_1[t_{1,1}, \ldots, t_{1,m_1}], \ldots, \\ \quad e_n[t_{n,1}, \ldots, t_{n,m_n}]) =_{\varphi} x \\ \quad z_{1,1}, \ldots, z_{1,m_1}, \ldots, z_{n,1}, \ldots, \\ \quad z_{n,m_1} \text{ are fresh variables} \end{array} \\[2ex]
s & \text{otherwise}
\end{cases}$$

Example 3.4 below illustrates the different outputs of the flattening transformation depending on φ, which (recursively) shows in the case when the input equation is $e[t_1, \ldots, t_n] =_{\varphi} e'[t'_1, \ldots, t'_m]$ (second case of the definition).

By means of flattening, complex unification is broken down into several simple ones. However, some equations which result from the transformation above are trivial and do not contribute to the semantics of our framework. Hence, they can be simply removed after applying the most general unifier to the remainder equations. We formalize this idea as follows.

An equation of the form $x = y$, with $x, y \in V$ is called a *trivial equation*. Note that goals $x \approx y$ are not trivial. Given a set of equations g, we define $split(g) = (g_1, g_2)$ as the function which splits g into two disjoint sets $g = g_1 \uplus g_2$ such that all equations in g_2 are trivial and no equation in g_1 is trivial. Now we are ready to complete the definition of the flattening transformation.

Definition 3.3 [flattening with strategy φ] The function $flat_{\varphi}(s)$ for an expression s is defined as follows. Let $g = flat^{\varphi}_{-}(s)$ be the pre–flattening of s, and $split(g) = (g_1, g_2)$. Then we let $flat_{\varphi}(s) = g_1 mgu(g_2)$.

Modifying a set of equations by flattening results in a flat equation set which cannot be flattened any further. The conversion to flat form subsumes the axioms of transitivity and f-substitutivity (i.e. they become 'built-in')

Example 3.4 Consider the following well-know, nonterminating[6] program `from/1`:

$$\begin{array}{r}\texttt{from(x)} \rightarrow [\texttt{x}|\texttt{from(s(x))}].\\ \texttt{first}([\texttt{x}|\texttt{y}]) \rightarrow \texttt{x}.\end{array}$$

Let $g \equiv \texttt{first}([\texttt{first}([0])]) = \texttt{first}([0,1])$ and $\varphi = inn$. Then the pre-flattening of g is $g_0 = flat_{\_}^{inn}(g) = (\texttt{first}([0]) = \texttt{y},\ \texttt{first}([\texttt{y}]) = \texttt{w},\ \texttt{first}([0,1]) = \texttt{z},\ \texttt{w} = \texttt{z})$, and $split(g_0) = (\{\texttt{first}([0]) = \texttt{y},\ \texttt{first}([\texttt{y}]) = \texttt{w},\ \texttt{first}([0,1]) = \texttt{z}\}, \{\ \texttt{w} = \texttt{z}\})$. Hence $flat_{inn}(g) = (\texttt{first}([0]) = \texttt{y},\ \texttt{first}([\texttt{y}]) = \texttt{w},\ \texttt{first}([0,1]) = \texttt{w})$.

Now, consider a different goal $g' \equiv (\texttt{first(from(s(x)))} \approx \texttt{z})$ and $\varphi = out$. Then the pre-flattening of g' is $g'_0 = flat_{\_}^{out}(g') = (\texttt{from(s(x))} = \texttt{y},\ \texttt{first}([\texttt{y}]) = \texttt{w},\ \texttt{w} \approx \texttt{z})$, and $split(g'_0) = (\{\texttt{from(s(x))} = \texttt{y},\ \texttt{first}([\texttt{y}]) = \texttt{w},\ \texttt{w} \approx \texttt{z}\}, \{\ \})$. Hence $flat_{out}(g') = g'_0$.

It is known that the fixpoint semantics allows for the reconstruction of the top down operational semantics and allows for the (bottom-up) computation of a model which is completely independent from the goal.

3.1 Fixpoint Semantics

Now we consider a generic immediate consequence operator $T_{\mathcal{R}}^{\varphi}$ which models computed answers w.r.t. φ. For any program $\mathcal{R}$, we denote by $\Phi_{\mathcal{R}}$ the set of identical equations $f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$, for each function symbol $f/n \in \mathcal{D}$. We let $\Im_{\mathcal{R}}^{\varphi}$ denote the set of the identical equations $c(x_1, \ldots, x_n) =_{\varphi} c(x_1, \ldots, x_n)$ for the constructor symbols c/n occurring in $\mathcal{R}$ only. As we will see, these *functional reflexivity axioms* play an important role in defining the fixpoint semantics of $\mathcal{R}$.

In non–strict languages, if the compositional character of meaning has to be preserved in presence of infinite data structures and partial functions, then non-normalizable terms, which may occur as subterms within normalizable expressions, also have to be assigned a denotation. Such a denotation is bound to the class of all partial results of the infinite computation along with the usual approximation ordering $\sqsubseteq$ on them [32,43] or, equivalently, the infinite data structure defined as the least upper bound of this class. Following [32,43], we introduce a fresh constant symbol $\perp$ into Σ to represent the value of expressions which would otherwise be undefined.

Definition 3.5 Let $\mathcal{I}$ be a Herbrand interpretation and $\mathcal{R} \in I\!R_{\varphi}$. Then,

[6] When considering the strict equality $\approx$ instead of the non–strict equality $=$, the completeness results for outermost narrowing in [25,26] when we are only interested on the computation of finite and total values of expressions generalize to nonterminating rules with little effort, see [31].

$$
\begin{aligned}
T^{\varphi}_{\mathcal{R}}(\mathcal{I}) = \Phi_{\mathcal{R}} \ \cup \ \Im^{\varphi}_{\mathcal{R}} \ \cup \ \{e \in \mathcal{B}_V \mid\ & (\lambda \to \rho \Leftarrow C) \ll \mathcal{R}^{\varphi}_{++}, \\
& l = r \in \mathcal{I}, C' \subseteq \mathcal{I}^{\mathcal{C}}, \\
& mgu(flat_{\varphi}(C), C') = \sigma, \\
& mgu(\{\lambda = r_{|u}\}\sigma) = \theta, \quad u \in \overline{O}^{\varphi}(r), \\
& e = (l = r[\rho]_u)\sigma\theta \ \}.
\end{aligned}
$$

where $\mathcal{R}^{\varphi}_{++} = \mathcal{R}$ if $\varphi = inn$, whereas $\mathcal{R}^{\varphi}_{++} = \mathcal{R}^{\varphi}_{+} \cup \{f(x_1, \ldots, x_n) \to \bot \mid f/n \in \mathcal{D}$ and $x_1, \ldots, x_n$ are variables$\}$ if $\varphi = out$, and $\mathcal{I}^{\mathcal{C}} = \{l = r \in \mathcal{I} \mid r$ is constructor$\}$

The following proposition allows us to define the fixpoint semantics.

Proposition 3.6 *The $T^{\varphi}_{\mathcal{R}}$ operator is continuous on the complete lattice of Herbrand interpretations, $\varphi \in \{inn, out\}$. The least fixpoint $lfp(T^{\varphi}_{\mathcal{R}}) = T^{\varphi}_{\mathcal{R}} \uparrow \omega$.*

Definition 3.7 The least fixpoint semantics of a program $\mathcal{R}$ in $I\!R_{\varphi}$ is defined as $\mathcal{F}_{\varphi}(\mathcal{R}) = lfp(T^{\varphi}_{\mathcal{R}})$, $\varphi \in \{inn, out\}$.

The following semantics is proven equivalent to the computed answer semantics below.

Definition 3.8 We let $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ denote $\{l = r \in lfp(T^{\varphi}_{\mathcal{R}}) \mid r$ does not contain any defined function symbol $f/n \in \mathcal{D}\}$, $\varphi \in \{inn, out\}$.

A goal of the form $x = y$, with $x, y \in V$ is called a *trivial goal.* Note that goals $x \approx y$ are not trivial.

The following result relates the answer substitutions computed by narrowing w.r.t φ with the $\bot$–free substitutions "computed" in the *computed answers (fixpoint) semantics* by standard unification.

Theorem 3.9 (strong soundness and completeness) *Let $\mathcal{R} \in I\!R_{\varphi}$ and g a (non–trivial) goal for φ. Then θ is a computed answer for g in $\mathcal{R}$ w.r.t. $\leadsto_{\varphi}$ iff there exists $g' \equiv e_1, \ldots, e_n \ll \mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ such that $\theta = mgu(flat_{\varphi}(g), g')_{\restriction Var(g)}$ (up to renaming.)*

Example 3.10 Consider again the program `from/1` of Example[7] 3.4

According to Definition 3.7, the fixpoint[8] semantics is

$$
\begin{aligned}
\mathcal{F}_{out}(\mathcal{R}) = \{\ & [\mathrm{x}|\mathrm{y}] \approx [\mathrm{x}|\mathrm{y}],\ \mathtt{s(x)} \approx \mathtt{s(x)},\ \mathtt{from(x)} = \mathtt{from(x)},\ \mathtt{first(x)} = \mathtt{first(x)}, \\
& \mathtt{from(x)} = \bot,\ \mathtt{from(x)} = [\mathrm{x}|\mathtt{from(s(x))}],\ \mathtt{from(x)} = [\mathrm{x}|\bot], \ldots, \\
& \mathtt{from(x)} = [\mathrm{x}|[\mathtt{s(x)}| \ldots [\mathtt{s^n(x)}|\mathtt{from(s^{n+1}(x))}]]],\ \mathtt{from(x)} = [\mathrm{x}|[\mathtt{s(x)}| \ldots [\mathtt{s^n(x)}|\bot]]], \\
& \ldots,\ \mathtt{first(x)} = \bot,\ \mathtt{first([x|y])} = \mathtt{x}\}
\end{aligned}
$$

Now, by Definition 3.8 computed answers can be distilled from

$$
\begin{aligned}
\mathcal{F}^{ca}_{out}(\mathcal{R}) = \{\ & [\mathrm{x}|\mathrm{y}] \approx [\mathrm{x}|\mathrm{y}], \mathtt{s(x)} \approx \mathtt{s(x)},\ \mathtt{from(x)} = \bot,\ \mathtt{from(x)} = [\mathrm{x}|\bot], \ldots, \\
& \mathtt{from(x)} = [\mathrm{x}|[\mathtt{s(x)}| \ldots [\mathtt{s^n(x)}|\bot]]], \ldots,\ \mathtt{first(x)} = \bot,\ \mathtt{first([x|y])} = \mathtt{x}\}
\end{aligned}
$$

[7] In the examples we use $s^n(x)$ as shorthand for $s(s(\ldots(x)))$.

[8] For the sake of simplicity, we omit the denotation of the 'built–in' defined function symbols "$\approx$" and "$\wedge$", e.g. the equations $(\mathtt{[s^n(x1)|y1]} \approx \mathtt{[s^n(x2)|y2]}) = (\mathtt{x1} \approx \mathtt{x2}) \wedge (\mathtt{y1} \approx \mathtt{y2})$, for $\mathtt{n} > \mathtt{0}$.

with $n \in \omega$. Given the goal $g \equiv$ ($\mathtt{first(from(s(x)))} \approx \mathtt{z}$), outermost narrowing only computes the answer $\{\mathtt{z/s(x)}\}$ in $\mathcal{R}$, which is also the only substitution which can be computed by unifying the flat goal ($\mathtt{from(s(x))} = \mathtt{y}$, $\mathtt{first(y)} = \mathtt{w}$, $\mathtt{w} \approx \mathtt{z}$) in $\mathcal{F}^{ca}_{out}(\mathcal{R})$.

According to Theorem 3.9, $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ can be used to simulate the execution for any (non–trivial) goal g, that is, $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ can be viewed as a (possibly infinite) set of 'unit' clauses, and the computed answer substitutions for g in $\mathcal{R}$ can be determined by 'executing' $flat_{\varphi}(g)$ in the program $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ by standard unification, as if the equality symbol were an ordinary predicate. Also note that, in our denotation, the proper semantic meaning of an equation $l = r$ is equality only in the case that r is total, i.e. without any occurrence of $\perp$.

Roughly speaking, $\perp$ is used in our methodology as an artifact to allow any equation $g(x_1, .., x_n) = x$ to "succeed" (while "computing" in the denotation). This is achieved by simply unifying it with the extra ("fake") equation $g(x_1, .., x_n) = \perp$. This ensures that every (non-strict) equation with a pure variable in its rhs is solvable, which is necessary for completeness. For instance, consider the following example in [32]. Let $f(t_1, .., t_m)$ be a term occurring in a rule body or a goal, and assume that f does not depend (is not strict) on the i-th argument. By definition, if t_i is a functional term $g(x_1, .., x_n)$, then the flattening introduces an equation $g(x_1, .., x_n) = x$ in the goal (and replaces t_i by x in $f(t_1, .., t_m)$). We must allow this equation $g(x_1, .., x_n) = x$ to succeed with an undefined value of x, whenever the value is not required in other equations, since it represents the i-th argument of f. Roughly speaking, solving this equation by means of the rule $g(x_1, .., x_n) = \perp$ has the effect to "reverse" the flattening which would otherwise force the evaluation of the call $g(x_1, .., x_n)$ and might fail (e.g. if g is undefined), while the evaluation was not demanded by f, and is undesired.

In the following, we show the relation between the semantics $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ and a novel operational "computed answer" semantics $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ which correctly models the behavior of single equations, which we introduce in the following.

3.2 Success Set Semantics

The operational success set semantics $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ of a program $\mathcal{R}$ w.r.t. narrowing semantics φ is defined by considering the answers computed for "most general calls", as shown by the following definition.

Definition 3.11 Let $\mathcal{R}$ be a program in $I\!R_{\varphi}$. Then,
$\mathcal{O}^{ca}_{\varphi}(\mathcal{R}) = \Im^{\varphi}_{\mathcal{R}} \cup \{(f(x_1, \ldots, x_n) = x_{n+1})\theta \mid (f(x_1, \ldots, x_n) =_{\varphi} x_{n+1}) \overset{\theta}{\leadsto}{}^{*}_{\varphi} \top$ where $f/n \in \mathcal{D}$, x_{n+1} and x_i are distinct variables, for $i = 1, \ldots, n\}$.

The following auxiliary operator $inprogress(S)$ is helpful. $inprogress(S)$ selects those equations of S which do not model successful computations, i.e. computations which are still incomplete or don't terminate.

Definition 3.12 Let S be a set of equations and Σ be the considered signature. We define:

$$inprogress(S) = \{\lambda = \rho \in S \mid \bot \text{ occurs in } \rho, \text{ or } \rho \text{ contains a defined function symbol of } \Sigma\}$$

By definition, $inprogress(\mathcal{O}_\varphi^{ca}(\mathcal{R})) = \emptyset$. The following result summarizes the relation between the operational and fixpoint denotations of a program.

Theorem 3.13 *The following relation holds:*

$$\mathcal{O}_\varphi^{ca}(\mathcal{R}) = \mathcal{F}_\varphi(\mathcal{R}) - inprogress(\mathcal{F}_\varphi(\mathcal{R}))$$

For the sake of clarity, let us summarize the relation among the three different program denotations $\mathcal{F}_\varphi(\mathcal{R})$, $\mathcal{F}_\varphi^{ca}(\mathcal{R})$ and $\mathcal{O}_\varphi^{ca}(\mathcal{R})$ introduced in this section. The compositional, fixpoint semantics $\mathcal{F}_\varphi(\mathcal{R})$ which models successful as well as partial (intermediate as well as nonterminating) computations is obtained by computing the lfp of the immediate consequences operator $T_\mathcal{R}^\varphi$. A subset of the denotation $\mathcal{F}_\varphi(\mathcal{R})$ is the *computed answer (fixpoint) semantics* $\mathcal{F}_\varphi^{ca}(\mathcal{R})$, which is obtained from $\mathcal{F}_\varphi(\mathcal{R})$ by removing the equations which model intermediate computations (i.e. those equations $f(\bar{t}) = s$ where s "has not reached its value") and is the only semantics which allows us to run (nontrivial) goals g by simply unifying $flat(g)$ with the equations in the denotation. Note that the semantics $\mathcal{F}_\varphi^{ca}(\mathcal{R})$ still models nonterminating functions, which are denoted by $\bot$. Finally, the operational *success set semantics* $\mathcal{O}_\varphi^{ca}(\mathcal{R})$ just catches successful derivations, that is, it only models the computed answers observable. Therefore, we have $\mathcal{O}_\varphi^{ca}(\mathcal{R}) \subseteq \mathcal{F}_\varphi^{ca}(\mathcal{R}) \subseteq \mathcal{F}_\varphi(\mathcal{R})$.

4 Declarative diagnosis of functional logic programs

We now introduce some basic definitions on the diagnosis of declarative programs [24]. As operational semantics we consider the success set semantics. Since we consider two different semantics $\mathcal{O}_\varphi^{ca}(\mathcal{R})$ and $\mathcal{F}_\varphi(\mathcal{R})$, in the sequel we also distinguish two different intended semantics: $\mathcal{I}_{ca}$ and $\mathcal{I}_\mathcal{F}$. While $\mathcal{I}_{ca}$ is the reference semantics from a programmer perspective, $\mathcal{I}_\mathcal{F}$ is suitable for technical reasons [20], as we explain in the following.

Figure 1 below summarizes our first definition.

Definition 4.1 Let $\mathcal{I}_{ca}$ be the specification of the intended success set semantics for $\mathcal{R}$.

(i) $\mathcal{R}$ is partially correct w.r.t. $\mathcal{I}_{ca}$, if $\mathcal{O}_\varphi^{ca}(\mathcal{R}) \subseteq \mathcal{I}_{ca}$.

(ii) $\mathcal{R}$ is complete w.r.t. $\mathcal{I}_{ca}$, if $\mathcal{I}_{ca} \subseteq \mathcal{O}_\varphi^{ca}(\mathcal{R})$.

(iii) $\mathcal{R}$ is totally correct w.r.t. $\mathcal{I}_{ca}$, if $\mathcal{O}_\varphi^{ca}(\mathcal{R}) = \mathcal{I}_{ca}$.

If a program contains errors, these are signalled by corresponding *symptoms*. The "intended success set semantics" allows us to establish the validity

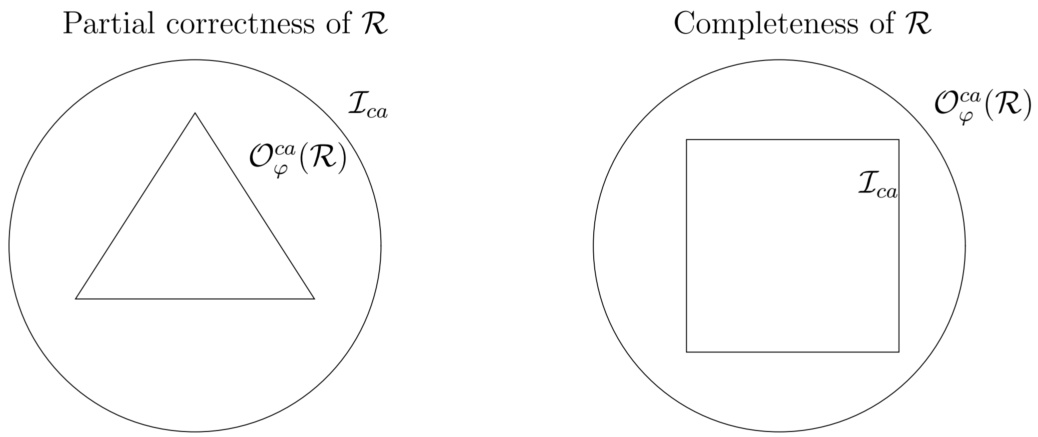

Fig. 1. Correctness and completeness of $\mathcal{R}$ w.r.t. computed answers

of an atomic equation by a simple "membership" test, in the style of the s-semantics.

Definition 4.2 Let $\mathcal{I}_{ca}$ be the specification of the intended success set semantics for $\mathcal{R}$. An *incorrectness symptom* is an equation e such that $e \in \mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ and $e \notin \mathcal{I}_{ca}$. An *incompleteness symptom* is an equation e such that $e \in \mathcal{I}_{ca}$ and $e \notin \mathcal{O}^{ca}_{\varphi}(\mathcal{R})$.

For the diagnosis, however, we need to consider a "well-provided" intended semantics $\mathcal{I}_{\mathcal{F}}$ (such that $\mathcal{I}_{ca} \subseteq \mathcal{I}_{\mathcal{F}}$), which models successful as well as "in progress" computations, and enjoys the semantic properties of the denotation formalized in Definition 3.7, that is, $\mathcal{I}_{\mathcal{F}}$ should correspond to the fixpoint semantics of the correct program and $\mathcal{I}_{ca} = \mathcal{I}_{\mathcal{F}} - \mathit{inprogress}(\mathcal{I}_{\mathcal{F}})$. In a practical system, of course, these descriptions would not be provided by the user but it would be automatically inferred from a finite set of input equations, e.g. a possibly inefficient (correct) version of the program, or a (executable) specification. This is the approach which we follow in the abstract diagnosis methodology that we present in Section 5.

In case of errors, in order to determine the faulty rules, the following definitions will be useful.

Definition 4.3 Let $\mathcal{I}_{\mathcal{F}}$ be the specification of the intended fixpoint semantics for $\mathcal{R}$. If there exists an equation $e \in T^{\varphi}_{\{r\}}(\mathcal{I}_{\mathcal{F}})$ and $e \notin \mathcal{I}_{\mathcal{F}}$, then the rule $r \in \mathcal{R}$ is incorrect on e. We also say that e is *incorrectly covered* by r.

Therefore, the incorrectness of rule r is signalled by a simple transformation of the intended semantics $\mathcal{I}_{\mathcal{F}}$.

Definition 4.4 Let $\mathcal{I}_{\mathcal{F}}$ be the specification of the intended fixpoint semantics for $\mathcal{R}$. An equation e is *uncovered* if $e \in \mathcal{I}_{\mathcal{F}}$ and $e \notin T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$.

By the above definition, an equation e is uncovered if it cannot be derived by any program rule using the intended fixpoint semantics. However, we are typically interested in the equations of $\mathcal{I}_{ca} \subseteq \mathcal{I}_{\mathcal{F}}$ which are uncovered, i.e. $e \in \mathcal{I}_{ca}$ and $e \notin T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$.

Proposition 4.5 *If there are no incorrect rules in $\mathcal{R}$ w.r.t the specification of the intended fixpoint semantics, then $\mathcal{R}$ is partially correct w.r.t. the intended success set semantics.*

Proposition 4.5 shows a simple methodology to prove partial correctness. Completeness is harder: some incompleteness cannot be detected by comparing the specification of the intended fixpoint semantics $\mathcal{I}_{\mathcal{F}}$ and $T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$. That is, the absence of uncovered equations does not allow us to derive that the program is complete. Let us consider the following counterexample:

Example 4.6 Consider $\varphi = out$ and the program $\mathcal{R} = \{\mathtt{f(x)} \rightarrow \mathtt{a} \Leftarrow \mathtt{f(x)} = \mathtt{a}\}$ and the specification $\mathcal{I}_{\mathcal{F}} = \{\mathtt{a} \approx \mathtt{a}, \mathtt{f(x)} = \mathtt{f(x)}, \mathtt{f(x)} = \bot, \mathtt{f(x)} = \mathtt{a}\}$. Then $\mathcal{R}$ is not complete since $\mathcal{I}_{ca} \subseteq \mathcal{O}^{ca}_{out}(\mathcal{R}) = \{\}$. However $\mathcal{I}_{\mathcal{F}} \subseteq T^{out}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}}) = \{\mathtt{a} \approx \mathtt{a}, \mathtt{f(x)} = \mathtt{f(x)}, \mathtt{f(x)} = \bot, \mathtt{f(x)} = \mathtt{a}\}$ and there are no uncovered equations.

The problem is related to the existence of several fixed points for the $T_{\mathcal{R}}$ operator. See [24] for details.

5 Abstract semantics

In this section, starting from the fixpoint semantics in Section 3, we develop an abstract semantics which approximates the observable behavior of the program and is adequate for modular data-flow analysis, such as the analysis of unsatisfiability of equation sets or any analysis which is based on the program success set. We assume the framework of abstract interpretation for analysis of equational unsatisfiability as defined in [5]. Another approach to constructing an abstract term rewriting system is followed in [16]. We think that another approximation of the fixpoint semantics given in the previous section which is different from the one that we describe in this section can be characterized by following an approach similar to that in [16]. We recall some previous principal definition about the abstract domains and the associated abstract operators, see [2,5,8] for details. Then we describe the novel abstract immediate consequence operator $T^{\sharp\varphi}_{\mathcal{R}}$, which approximates $T^{\varphi}_{\mathcal{R}}$, and the corresponding abstract fixpoint semantics, $\mathcal{F}^{\sharp}_{\varphi}(\mathcal{R})$ and $\mathcal{F}^{ca\sharp}_{\varphi}(\mathcal{R})$. An abstract success set semantics $\mathcal{O}^{\sharp\varphi}$ can also be systematically derived from the concrete one, by replacing the considered narrowing calculus by a corresponding abstract version, see e.g. [8]. In the following, we denote the abstract analog of a concrete object O by $O^{\sharp}$.

The abstract methodology in this section generalizes the results in [2] by making them parametric w.r.t. $\varphi = inn, out$.

5.1 Abstract Programs and Operators

A *description* is the association of an *abstract domain* $(D, \leq)$ (a poset) with a *concrete domain* $(E, \leq)$ (a poset). The correspondence between the abstract and concrete domain is established through a 'concretization' function $\gamma : D \to 2^E$. We say that d *approximates* e, written $d \propto e$, iff $e \in \gamma(d)$.

Abstract substitutions are introduced for the purpose of describing the computed answer substitutions for a given goal. The domains for equations and substitutions are based in a notion of the abstract Herbrand universe $\mathcal{H}_V^\sharp = (\tau(\Sigma \cup V \cup \{\sharp\}), \preceq)$, which introduces an irreducible symbol $\sharp$, such that $\sharp \notin \Sigma$ (see [2,5,8].)

Our analysis is based on a form of simplified (abstract) program which always terminates and in which the query can be executed efficiently. Our notion of abstract program is parametric with respect to a loop-check. Two different instances can be found in [5,8].

Definition 5.1 A loop-check is a graph $\mathcal{G}_\mathcal{R}$ associated with a program $\mathcal{R}$, i.e. a relation consisting of a set of pairs of terms, such that:
(1) the transitive closure $\mathcal{G}_\mathcal{R}^+$ is decidable and
(2) Let $\overset{\circ}{t} = t'$ be a function which assigns to a term t some node t' in $\mathcal{G}_\mathcal{R}$. If there is an infinite sequence:

$$g_0 \overset{\theta_0}{\leadsto}_\varphi g_1 \overset{\theta_1}{\leadsto}_\varphi \dots$$

then $\exists i \geq 0.$ $\langle \overset{\circ}{t_i}, \overset{\circ}{t_i} \rangle \in \mathcal{G}_\mathcal{R}^+$, where $t_i = g_{i|u}$, $u \in \overline{O}(g_i)$. (we refer to $\langle \overset{\circ}{t_i}, \overset{\circ}{t_i} \rangle$ as a 'cycle' of $\mathcal{G}_\mathcal{R}$.)

A program is abstracted by simplifying the right-hand side and the body of each clause. This definition is given inductively on the structure of terms and equations. The main idea is that terms whose corresponding nodes in $\mathcal{G}_\mathcal{R}$ have a cycle are drastically simplified by replacing them by $\sharp$. We use this definition in a iterative manner. We first abstract a concrete rule r obtaining $r^\sharp$ (we select a rule with direct recursion if any; otherwise we choose any rule in the program). Then we replace r by $r^\sharp$ in $\mathcal{R}$, and recompute the loop–check before proceeding to abstract the next rule.

Definition 5.2 [abstract rule] Let $\mathcal{R}$ be a program and let $r = (\lambda \to \rho \Leftarrow C) \in \mathcal{R}$. Let $\mathcal{G}_\mathcal{R}$ be a loop-check for $\mathcal{R}$. We define the abstraction of r as follows: $r^\sharp = (\lambda \to sh(\rho, \mathcal{G}_\mathcal{R}) \Leftarrow sh(C, \mathcal{G}_\mathcal{R}))$ where the shell $sh(x, \mathcal{G})$ of an expression x according to a loop-check $\mathcal{G}$ is defined inductively

$$sh(x,\mathcal{G}) = \begin{cases} x & \text{if } x \in V \\ f(sh(t_1,\mathcal{G}),\dots,sh(t_k,\mathcal{G})) & \text{if } x \equiv f(t_1,\dots,t_k) \text{ and } \langle \overset{\circ}{x}, \overset{\circ}{x} \rangle \notin \mathcal{G}^+ \\ sh(l,\mathcal{G}) = sh(r,\mathcal{G}) & \text{if } x \equiv (l = r) \\ sh(e_1,\mathcal{G}),\dots,sh(e_n,\mathcal{G}) & \text{if } x \equiv e_1,\dots,e_n \\ \sharp & \text{otherwise} \end{cases}$$

We can now formalize the abstract semantics.

5.1.1 Abstract Fixpoint Semantics

We define an abstract fixpoint semantics in terms of the least fixpoint of a continuous transformation $T^{\sharp\varphi}_{\mathcal{R}}$ based on abstract unification and the operation of abstraction of a program. The idea is to provide a finitely computable approximation of the concrete denotation of the program $\mathcal{R}$. In the following, we define the abstract transformation $T^{\sharp\varphi}_{\mathcal{R}}$.

Definition 5.3 [abstract Herbrand base, abstract Herbrand interpretation] The abstract Herbrand base of equations $\mathcal{B}^{\sharp}_{V}$ is defined as the set of equations over the abstract Herbrand universe $\mathcal{H}^{\sharp}_{V}$. An abstract Herbrand interpretation is any element of $2^{\mathcal{B}^{\sharp}_{V}}$.

We can show that the set of abstract interpretations is a complete lattice w.r.t. $\subseteq$. An abstract trivial equation is an equation $\sharp = x$, $x = \sharp$ or $\sharp = \sharp$.

Definition 5.4 Let $\mathcal{R}$ be a program in $I\!R_{\varphi}$, $\mathcal{G}_{\mathcal{R}}$ be a loop-check for $\mathcal{R}$ and $\mathcal{R}^{\sharp}$ be the abstraction of $\mathcal{R}$ using $\mathcal{G}_{\mathcal{R}}$ where we also drop any abstract trivial equation from the body of the rules if there exist. Let $\mathcal{I}$ be an abstract interpretation. Then,

$$\begin{aligned} T^{\sharp\varphi}_{\mathcal{R}}(\mathcal{I}) = \Phi_{\mathcal{R}} \;\cup\; \Im^{\varphi}_{\mathcal{R}} \;\cup\; \{e \in \mathcal{B}^{\sharp}_{V} \mid\ & (\lambda \to \rho \Leftarrow C) \ll \mathcal{R}^{\sharp\varphi}_{++}, \\ & \{l = r\} \in \mathcal{I}, C' \subseteq \mathcal{I}^{\mathcal{C}}, \\ & mgu^{\sharp}(flat_{\varphi}(C), C') = \sigma, \\ & mgu^{\sharp}(\{\lambda = (r_{|u})\}\sigma) = \theta, \\ & u \in \overline{O}^{\varphi}(r),\quad e = (l = r[\rho]_u)\sigma\theta\ \}. \end{aligned}$$

where $\mathcal{R}^{\sharp\varphi}_{++} = \mathcal{R}^{\sharp}$ if $\varphi = inn$, whereas $\mathcal{R}^{\sharp\varphi}_{++} = \mathcal{R}^{\sharp\varphi}_{+} \cup \{f(x_1, \ldots, x_n) \to \bot \mid f/n \in \mathcal{D}$ and $x_1, \ldots, x_n$ are variables$\}$ if $\varphi = out$, and $\mathcal{I}^{\mathcal{C}} = \{l = r \in \mathcal{I} \mid r$ is constructor$\}$.

Proposition 5.5 *The $T^{\sharp\varphi}_{\mathcal{R}}$ operator is continuous on the complete lattice of abstract Herbrand interpretations.*

We can define $\mathcal{F}^{\sharp}_{\varphi}(\mathcal{R})$ and $\mathcal{F}^{ca\sharp}_{\varphi}(\mathcal{R})$ in a way similar to the concrete constructions of $\mathcal{F}_{\varphi}(\mathcal{R})$ and $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$, as in Section 3.

Definition 5.6 [abstract least fixpoint semantics] The abstract least fixpoint semantics of a program $\mathcal{R}$ is $\mathcal{F}^{\sharp}_{\varphi}(\mathcal{R}) = lfp(T^{\sharp\varphi}_{\mathcal{R}})$. Let $\mathcal{F}^{ca\sharp}_{\varphi}(\mathcal{R}) = \{l = r \in \mathcal{F}^{\sharp}_{\varphi}(\mathcal{R}) \mid r$ does not contain any defined function symbol $f/n \in \mathcal{D}\}$, $\varphi \in \{inn, out\}$

The following theorem states that $\mathcal{F}^{\sharp}_{\varphi}(\mathcal{R})$ and $\mathcal{F}^{ca\sharp}_{\varphi}(\mathcal{R})$ are finitely computable.

Theorem 5.7 *There exists a finite positive number k such that $\mathcal{F}^{\sharp}_{\varphi}(\mathcal{R}) = T^{\sharp\varphi}_{\mathcal{R}} \uparrow k$, $\varphi \in \{inn, out\}$.*

From a semantics viewpoint, given a program $\mathcal{R}$, the fixpoint semantics $\mathcal{F}_{\varphi}(\mathcal{R})$ (resp. $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$) is approximated by the corresponding abstract fixpoint semantics $\mathcal{F}^{\sharp}_{\varphi}(\mathcal{R})$ (resp. $\mathcal{F}^{ca\sharp}_{\varphi}(\mathcal{R})$). That is, we can compute an abstract approximation of the concrete semantics in a finite number of steps. The correctness of the abstract fixpoint semantics with respect to the concrete semantics is proved by the following:

Theorem 5.8 $\mathcal{F}^{\sharp}_{\varphi}(\mathcal{R}) \propto \mathcal{F}_{\varphi}(\mathcal{R})$ *and* $\mathcal{F}^{ca\sharp}_{\varphi}(\mathcal{R}) \propto \mathcal{F}^{ca}_{\varphi}(\mathcal{R})$.

The semantics $\mathcal{F}^{ca\sharp}_{\varphi}(\mathcal{R})$ collects goal-independent information about success patterns of a given program. The relation between the abstract fixpoint and the concrete operational semantics (success set) is given by the following theorem. Roughly speaking, given a goal g, we obtain a description of the set of the computed answers of g by abstract unification of the equations in $flat_{\varphi}(g)$ with equations in the approximated semantics $\mathcal{F}^{ca\sharp}_{\varphi}(\mathcal{R})$.

Theorem 5.9 (completeness) *Let $\mathcal{R}$ be a program in $I\!R_{\varphi}$ and g be a (non-trivial) goal. If θ is a computed answer substitution for g in $\mathcal{R}$ w.r.t. φ, then there exists $g' \equiv e_1, \ldots, e_m \ll \mathcal{F}^{ca\sharp}_{\varphi}(\mathcal{R})$ such that $\theta' = mgu^{\sharp}(flat_{\varphi}(g), g')$, and $(\theta' \preceq \theta)_{\lceil Var(g)}$.*

Example 5.10 Let us consider the following program, built up with pieces of code from previous examples:

$$\begin{aligned}
\mathtt{from}(\mathtt{x}) &\to [\mathtt{x}|\mathtt{from}(\mathtt{s}(\mathtt{x}))].\\
\mathtt{first}([\mathtt{x}|\mathtt{y}])) &\to \mathtt{x}.\\
\mathtt{add}(\mathtt{0},\mathtt{x}) &\to \mathtt{x}.\\
\mathtt{add}(\mathtt{s}(\mathtt{x}),\mathtt{y}) &\to \mathtt{s}(\mathtt{add}(\mathtt{x},\mathtt{y})).\\
\mathtt{double}(\mathtt{x}) &\to \mathtt{add}(\mathtt{x},\mathtt{x}).
\end{aligned}$$

We consider the outermost strategy, i.e. $\varphi = out$. Let us consider the loop-check [8] $\mathcal{G}_{\mathcal{R}} = \{\langle\mathtt{from}(\mathtt{x}),\mathtt{from}(\mathtt{x})\rangle, \langle\mathtt{add}(\mathtt{x}),\mathtt{add}(\mathtt{x})\rangle\}$, and let $t \overset{\circ}{=} t'$ be the (partial) function which, given a graph, assigns to a term t some node t' in the graph such that t' unifies with t, if one such node t' exists. Then, the abstraction of the program $\mathcal{R}$ is $\mathcal{R}^{\sharp}$:

$$\begin{aligned}
\mathtt{from}(\mathtt{x}) &\to [\mathtt{x}|\sharp].\\
\mathtt{first}([\mathtt{x}|\mathtt{y}]) &\to \mathtt{x}.\\
\mathtt{add}(\mathtt{0},\mathtt{x}) &\to \mathtt{x}.\\
\mathtt{add}(\mathtt{s}(\mathtt{x}),\mathtt{y}) &\to \mathtt{s}(\sharp).\\
\mathtt{double}(\mathtt{x}) &\to \mathtt{add}(\mathtt{x},\mathtt{x}).
\end{aligned}$$

Then, the fixpoint semantics is $\mathcal{F}_{out}(\mathcal{R}) =$

$$\begin{array}{l}
\{\mathtt{0} \approx \mathtt{0},\ \mathtt{s}(\mathtt{x}) \approx \mathtt{s}(\mathtt{x}),\ [\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}|\mathtt{y}],\ \mathtt{double}(\mathtt{x}) = \mathtt{double}(\mathtt{x}),\ \mathtt{add}(\mathtt{x},\mathtt{y}) = \mathtt{add}(\mathtt{x},\mathtt{y}),\\
\mathtt{from}(\mathtt{x}) = \mathtt{from}(\mathtt{x}),\ \mathtt{first}(\mathtt{x}) = \mathtt{first}(\mathtt{x}),\ \mathtt{and}(\mathtt{x},\mathtt{y}) = \mathtt{and}(\mathtt{x},\mathtt{y}),\ (\mathtt{x} \approx \mathtt{y}) = (\mathtt{x} \approx \mathtt{y}),\\
\mathtt{and}(\mathtt{true},\mathtt{x}) = \mathtt{x},\ \mathtt{and}(\mathtt{x},\mathtt{y}) = \bot,\ ([\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}_1|\mathtt{y}_1]) = \mathtt{and}(\mathtt{x} \approx \mathtt{x}_1, \mathtt{y} \approx \mathtt{y}_1),\ ([\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}_1|\mathtt{y}_1]) = \bot,\\
(\mathtt{s}(\mathtt{x}) \approx \mathtt{s}(\mathtt{y})) = (\mathtt{x} \approx \mathtt{y}),\ \mathtt{from}(\mathtt{x}) = \bot,\ \mathtt{from}(\mathtt{x}) = [\mathtt{x}|\mathtt{from}(\mathtt{s}(\mathtt{x}))],\ \mathtt{from}(\mathtt{x}) = [\mathtt{x}|\bot], \ldots,\\
\mathtt{from}(\mathtt{x}) = [\mathtt{x}|[\mathtt{s}(\mathtt{x})|\ldots[\mathtt{s}^{\mathtt{n}}(\mathtt{x})|\mathtt{from}(\mathtt{s}^{\mathtt{n}+1}(\mathtt{x}))]]], \ldots, \mathtt{from}(\mathtt{x}) = [\mathtt{x}|[\mathtt{s}(\mathtt{x})|\ldots[\mathtt{s}^{\mathtt{n}}(\mathtt{x})|\bot]]], \ldots,\\
\mathtt{first}(\mathtt{x}) = \bot,\ \mathtt{first}([\mathtt{x}|\mathtt{y}]) = \mathtt{x}, (\mathtt{x} \approx \mathtt{y}) = \bot,\ (\mathtt{0} \approx \mathtt{0}) = \mathtt{true},\ (\mathtt{s}(\mathtt{x}) \approx \mathtt{s}(\mathtt{y})) = \bot, \ldots,\\
(\mathtt{s}^{\mathtt{n}}(\mathtt{x}) \approx \mathtt{s}^{\mathtt{n}}(\mathtt{y})) = \bot, \ldots,\ (\mathtt{s}(\mathtt{x}) \approx \mathtt{s}(\mathtt{y})) = (\mathtt{x} \approx \mathtt{y}), \ldots,\ (\mathtt{s}^{\mathtt{n}}(\mathtt{x}) \approx \mathtt{s}^{\mathtt{n}}(\mathtt{y})) = (\mathtt{x} \approx \mathtt{y}), \ldots,\\
(\mathtt{s}(\mathtt{0}) \approx \mathtt{s}(\mathtt{0})) = \mathtt{true}, \ldots,\ (\mathtt{s}^{\mathtt{n}}(\mathtt{0}) \approx \mathtt{s}^{\mathtt{n}}(\mathtt{0})) = \mathtt{true}, \ldots\ \mathtt{add}(\mathtt{x},\mathtt{y}) = \bot,\ \mathtt{add}(\mathtt{s}(\mathtt{x}),\mathtt{y}) = \mathtt{s}(\bot),\\
\ldots, \mathtt{add}(\mathtt{s}^{\mathtt{n}}(\mathtt{x}),\mathtt{y}) = \mathtt{s}^{\mathtt{n}}(\bot), \ldots,\ \mathtt{add}(\mathtt{0},\mathtt{x}) = \mathtt{x},\ \mathtt{add}(\mathtt{s}(\mathtt{0}),\mathtt{x}) = \mathtt{s}(\mathtt{x}), \ldots,\ \mathtt{add}(\mathtt{s}^{\mathtt{n}}(\mathtt{0}),\mathtt{x}) = \mathtt{s}^{\mathtt{n}}(\mathtt{x}),\\
\ldots, \mathtt{add}(\mathtt{s}(\mathtt{x}),\mathtt{y}) = \mathtt{s}(\mathtt{add}(\mathtt{x},\mathtt{y})), \ldots, \mathtt{add}(\mathtt{s}^{\mathtt{n}}(\mathtt{x}),\mathtt{y}) = \mathtt{s}^{\mathtt{n}}(\mathtt{add}(\mathtt{x},\mathtt{y})), \ldots, \mathtt{double}(\mathtt{x}) = \bot, \ldots,\\
\mathtt{double}(\mathtt{s}^{\mathtt{n}}(\mathtt{x})) = \mathtt{s}^{\mathtt{n}}(\bot), \ldots, \mathtt{double}(\mathtt{x}) = \mathtt{add}(\mathtt{x},\mathtt{x}),\ \mathtt{double}(\mathtt{s}(\mathtt{x})) = \mathtt{s}(\mathtt{add}(\mathtt{x},\mathtt{s}(\mathtt{x}))), \ldots,\\
\mathtt{double}(\mathtt{s}^{\mathtt{n}}(\mathtt{x})) = \mathtt{s}^{\mathtt{n}}(\mathtt{add}(\mathtt{x},\mathtt{s}^{\mathtt{n}}(\mathtt{x}))), \ldots, \mathtt{double}(\mathtt{0}) = \mathtt{0},\ \mathtt{double}(\mathtt{s}(\mathtt{0})) = \mathtt{s}^{2}(\mathtt{0}), \ldots,\\
\mathtt{double}(\mathtt{s}^{\mathtt{n}}(\mathtt{0})) = \mathtt{s}^{2*\mathtt{n}}(\mathtt{0}), \ldots\}
\end{array}$$

and the corresponding abstract fixpoint semantics is the finite set $\mathcal{F}^{\sharp}_{out}(\mathcal{R}) =$
$\{\mathtt{0} \approx \mathtt{0},\ \mathtt{s(x)} \approx \mathtt{s(x)},\ [\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}|\mathtt{y}],\ \mathtt{add(x,y)} = \mathtt{add(x,y)},\ \mathtt{from(x)} = \mathtt{from(x)},$
$\mathtt{double(x)} = \mathtt{double(x)},\ \mathtt{first(x)} = \mathtt{first(x)},\ (\mathtt{x} \approx \mathtt{y}) = (\mathtt{x} \approx \mathtt{y}),\ \mathtt{and(x,y)} = \mathtt{and(x,y)},$
$([\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}|\mathtt{y}]) = \sharp,\ ([\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}|\mathtt{y}]) = \perp,\ (\mathtt{0} \approx \mathtt{0}) = \mathtt{true},\ (\mathtt{x} \approx \mathtt{y}) = \perp,\ (\mathtt{s(x)} \approx \mathtt{s(y)}) = \sharp,$
$([\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}|\mathtt{y}]) = \mathtt{and}(\sharp,\sharp),\ \mathtt{and(true,x)} = \mathtt{x},\ \mathtt{and(x,y)} = \perp,\ \mathtt{from(x)} = \perp,\ \mathtt{from(x)} = [\mathtt{x}|\sharp],$
$\mathtt{first(x)} = \perp,\ \mathtt{first}([\mathtt{x}|\mathtt{y}]) = \mathtt{x},\ \mathtt{add(x,y)} = \perp,\ \mathtt{add(0,x)} = \mathtt{x},\ \mathtt{add(s(x),y)} = \mathtt{s}(\sharp),$
$\mathtt{double(0)} = \mathtt{0},\ \mathtt{double(x)} = \perp,\ \mathtt{double(x)} = \mathtt{add(x,x)},\ \mathtt{double(s(x))} = \mathtt{s}(\sharp)\}$
which approximates the program success set.

Given a goal $g \equiv\ \mathtt{first}([\mathtt{double(x)}|\mathtt{from(x)}]) \approx \mathtt{y}$, outermost narrowing computes the infinite set of substitutions $\{\{\mathtt{x/0},\ \mathtt{y/0}\}, \{\mathtt{x/s(0)},\ \mathtt{y/s^2(0)}\}, \ldots,$ $\{\mathtt{x/s^n(0)},\ \mathtt{y/s^{2*n}(0)}\}\}$. Now, the abstract substitutions computed by abstract unification in $\mathcal{F}^{ca\sharp}_{out}(\mathcal{R})$ of the equations of flattened goal $flat_{out}(g) = (\mathtt{double(x)} =$ $\mathtt{z_1},\ \mathtt{from(x)} = \mathtt{z_2},\ \mathtt{first}([\mathtt{z_1}|\mathtt{z_2}]) = \mathtt{z_3},\ \mathtt{z_3} \approx \mathtt{y})$ are $\{\{\mathtt{x/0},\ \mathtt{y/0}\}, \{\ \mathtt{x/s(x_1)},\ \mathtt{y/s}(\sharp)\},$ $\{\mathtt{x/x'}, \mathtt{y}/\perp\}\}$, which approximate the computed answers of g.

We would like to note that the two different symbols $\sharp$ and $\perp$ which are related, in our framework, with the "lack of information" may appear simultaneously in the semantics $\mathcal{F}^{\sharp}_{out}(\mathcal{R})$. So we need to distinguish between them because these symbols do behave differently. The "default value" $\perp$, which is introduced to ensure completeness of our fixpoint description, is handled by the rule $f(x_1, \ldots, x_n) = \perp$ as a special data constructor (constant) symbol. On the other hand, the symbol $\sharp$, which is introduced in our construction to ensure termination of the approximate computations, abstractly unifies with every term and bears the additional information to "stop the computation rather than to continue looking for a better approximation of the result".

6 Abstract diagnosis

An efficient debugger can be based on the notion of over-approximation and under-approximation for the intended fixpoint semantics that we have introduced. The basic idea is to consider two sets to verify partial correctness: $\mathcal{I}^+$ which over-approximates the intended fixpoint semantics $\mathcal{I}_{\mathcal{F}}$ (that is, $\mathcal{I}_{\mathcal{F}} \subseteq \gamma(\mathcal{I}^+)$) and $\mathcal{I}^-$ which under-approximates $\mathcal{I}_{\mathcal{F}}$ (that is, $\mathcal{I}_{\mathcal{F}} \supseteq \gamma(\mathcal{I}^-)$). We can then use such sets as shown below, where the immediate consequences operators (w.r.t. the program $\mathcal{R}$) are applied once to $\mathcal{I}^-$ (resp. $\mathcal{I}^+$) to check (in-)correctness –resp. (in-)completeness– w.r.t. $\mathcal{I}_{ca}$.

The following results hold. These propositions correct a misprint and also generalize the results in [2]. Define $[\![S]\!] = \exists\bar{x}.S' = S''$, where the n-tuple of occurrences of $\sharp$ in S is replaced by an n-tuple of (existentially quantified) fresh variables in S'. By abuse, in the following, we sometimes disregard the existential quantifiers and confuse S' and S''. .

Proposition 6.1 *If there exists an equation e such that $(\exists\mathcal{I}' \subseteq \gamma(\mathcal{I}^+))(e \notin \mathcal{I}')$ and $e \in T^{\varphi}_{\{r\}}([\![\mathcal{I}^-]\!])$, then the rule $r \in \mathcal{R}$ is incorrect on e.*

Proposition 6.2 *If there exists an (abstract) equation e such that $e \in \mathcal{I}^-$ and $[\![e]\!] \notin \gamma(T^{\sharp\varphi}_{\mathcal{R}}(\mathcal{I}^+))$, then $[\![e]\!]$ is uncovered in $\mathcal{R}$.*

Here we would like to clarify that the use of γ in the previous results does not prevent us from having a finite debugging methodology, as the test that e ($[\![e]\!]$) is not in $\gamma(\mathcal{I}^+)$ (resp. $\gamma(T^{\sharp\varphi}_{\mathcal{R}}(\mathcal{I}^+))$ can be efficiently and safely implemented by simply checking whether the considered equation abstractly unifies with some element of $\mathcal{I}^+$ or $T^{\sharp\varphi}_{\mathcal{R}}(\mathcal{I}^+)$, and then performing an easy test on the abstract mgu.

In the following, by abuse we let $\mathcal{I}$ denote the program that specifies the intended semantics. In the following, we consider $\mathcal{I}^+ = lfp(T^{\sharp\varphi}_{\mathcal{I}})$, i.e. we consider the abstract success set that we have defined in previous section as overapproximation of the success set of a program. We can consider any of the sets defined in the works of [20,23] as underapproximation of $\mathcal{I}$. Alternatively, we can simply take the set which results from a finite number of iterations of the $T^{\varphi}_{\mathcal{I}}$ function (the concrete operator). This provides a simple albeit useful debugging scheme which is satisfactory in practice. Let us illustrate this method by an example.

Example 6.3 Let us consider now the following (wrong) program $\mathcal{R}$ for doubling.

$$\begin{aligned}
\mathtt{add(0,x)} &\rightarrow \mathtt{x.}\\
\mathtt{add(s(x),y)} &\rightarrow \mathtt{s(add(x,y)).}\\
\mathtt{double(0)} &\rightarrow \mathtt{0.}\\
\mathtt{double(s(x))} &\rightarrow \mathtt{double(x).}
\end{aligned}$$

The intended specification is given by the following program which uses addition for doubling:

$$\begin{aligned}
\mathtt{add(0,x)} &\rightarrow \mathtt{x.}\\
\mathtt{add(s(x),y)} &\rightarrow \mathtt{s(add(x,y)).}\\
\mathtt{double(x)} &\rightarrow \mathtt{add(x,x).}
\end{aligned}$$

Let $\varphi = out$; then $\mathcal{I}^\sharp =$

$$\begin{aligned}
\mathtt{add(0,x)} &\rightarrow \mathtt{x.}\\
\mathtt{add(s(x),y)} &\rightarrow \mathtt{s(\sharp).}\\
\mathtt{double(x)} &\rightarrow \mathtt{add(x,x).}\\
\mathtt{0} \approx \mathtt{0} &\rightarrow \mathtt{true.}\\
\mathtt{s(x)} \approx \mathtt{s(y)} &\rightarrow \sharp.\\
\mathtt{add(x,y)} &\rightarrow \bot.\\
\mathtt{double(x)} &\rightarrow \bot.\\
\mathtt{x} \approx \mathtt{y} &\rightarrow \bot.
\end{aligned}$$

The overapproximation $\mathcal{I}^+$ is given by the following set of equations (after three iterations of the $T^{out\sharp}_{\mathcal{I}}$ operator, we get the fixpoint):

$$\begin{aligned}
\mathcal{I}^+ = \mathcal{F}^\sharp_{out}(\mathcal{I}) = lfp(T^{out\sharp}_{\mathcal{I}}) = \{&\mathtt{double(0)} = \mathtt{0},\ \mathtt{double(s(x))} = \mathtt{s(\sharp)},\ \mathtt{add(x,y)} = \bot,\\
&\mathtt{double(x)} = \bot,\ (\mathtt{x} \approx \mathtt{y}) = \bot,\ (\mathtt{s(x)} \approx \mathtt{s(y)}) = \sharp,\ (\mathtt{0} \approx \mathtt{0}) = \mathtt{true},\ \mathtt{add(0,x)} = \mathtt{x},\\
&\mathtt{double(x)} = \mathtt{add(x,x)},\ \mathtt{add(s(x),y)} = \mathtt{s(\sharp)},\ \mathtt{add(x,y)} = \mathtt{add(x,y)},\ \mathtt{0} \approx \mathtt{0},\ \mathtt{s(x)} \approx \mathtt{s(x)},\\
&\mathtt{double(x)} = \mathtt{double(x)},\ (\mathtt{x} \approx \mathtt{y}) = (\mathtt{x} \approx \mathtt{y})\}
\end{aligned}$$

After four iterations of the $T^{out}_{\mathcal{I}}$ operator, we get to the underapproximation

$\mathcal{I}^- = \{\mathtt{add(x,y)} = \mathtt{add(x,y)},\ \mathtt{0} \approx \mathtt{0},\ \mathtt{s(x)} \approx \mathtt{s(x)},\ \mathtt{double(x)} = \mathtt{double(x)},\ (\mathtt{x} \approx \mathtt{y}) = (\mathtt{x} \approx \mathtt{y}),$
$(\mathtt{x} \approx \mathtt{y}) = \bot,\ (\mathtt{s(x)} \approx \mathtt{s(y)}) = \bot,\ (\mathtt{s}^2\mathtt{(x)} \approx \mathtt{s}^2\mathtt{(y)}) = \bot,\ (\mathtt{s}^3\mathtt{(x)} \approx \mathtt{s}^3\mathtt{(y)}) = \bot,$
$(\mathtt{s(x)} \approx \mathtt{s(y)}) = (\mathtt{x} \approx \mathtt{y}),\ (\mathtt{s}^2\mathtt{(x)} \approx \mathtt{s}^2\mathtt{(y)}) = (\mathtt{x} \approx \mathtt{y}),\ (\mathtt{s}^3\mathtt{(x)} \approx \mathtt{s}^3\mathtt{(y)}) = (\mathtt{x} \approx \mathtt{y}),$
$(\mathtt{s}^4\mathtt{(x)} \approx \mathtt{s}^4\mathtt{(y)}) = (\mathtt{x} \approx \mathtt{y}),\ (\mathtt{0} \approx \mathtt{0}) = \mathtt{true},\ (\mathtt{s(0)} \approx \mathtt{s(0)}) = \mathtt{true},\ (\mathtt{s}^2\mathtt{(0)} \approx \mathtt{s}^2\mathtt{(0)}) = \mathtt{true},$
$(\mathtt{s}^3\mathtt{(0)} \approx \mathtt{s}^3\mathtt{(0)}) = \mathtt{true},\ \mathtt{add(x,y)} = \bot,\ \mathtt{add(s(x),y)} = \mathtt{s}(\bot),\ \mathtt{add(s}^2\mathtt{(x),y)} = \mathtt{s}^2(\bot),$
$\mathtt{add(s}^3\mathtt{(x),y)} = \mathtt{s}^3(\bot),\ \mathtt{add(0,x)} = \mathtt{x},\ \mathtt{add(s(0),x)} = \mathtt{s(x)},\ \mathtt{add(s}^2\mathtt{(0),x)} = \mathtt{s}^2\mathtt{(x)},$
$\mathtt{add(s}^3\mathtt{(0),y)} = \mathtt{s}^3\mathtt{(y)},\ \mathtt{add(s(x),y)} = \mathtt{s(add(x,y))},\ \mathtt{add(s}^2\mathtt{(x),y)} = \mathtt{s}^2\mathtt{(add(x,y))},$
$\mathtt{add(s}^3\mathtt{(x),y)} = \mathtt{s}^3\mathtt{(add(x,y))},\ \mathtt{add(s}^4\mathtt{(x),y)} = \mathtt{s}^4\mathtt{(add(x,y))},\ \mathtt{double(x)} = \bot,$
$\mathtt{double(s(x))} = \mathtt{s}(\bot),\ \mathtt{double(s}^2\mathtt{(x))} = \mathtt{s}^2(\bot),\ \mathtt{double(x)} = \mathtt{add(x,x)},$
$\mathtt{double(s(x))} = \mathtt{s(add(x,s(x)))},\ \mathtt{double(s}^2\mathtt{(x))} = \mathtt{s}^2\mathtt{(add(x,s}^2\mathtt{(x)))},$
$\mathtt{double(s}^3\mathtt{(x))} = \mathtt{s}^3\mathtt{(add(x,s}^3\mathtt{(x)))}, \mathtt{double(0)} = \mathtt{0},\ \mathtt{double(s(0))} = \mathtt{s(s(0))},$
$\mathtt{double(s}^2\mathtt{(0))} = \mathtt{s}^4\mathtt{(0)}\}$

Then,

$T_{\mathcal{R}}\ (\mathcal{I}^-) = \{\mathtt{0} \approx \mathtt{0},\ \mathtt{s(x)} \approx \mathtt{s(x)},\ \mathtt{add(x,y)} = \mathtt{add(x,y)},\ \mathtt{double(x)} = \mathtt{double(x)},$
$\mathtt{aprox(x,y)} = \mathtt{aprox(x,y)},\ (\mathtt{0} \approx \mathtt{0}) = \mathtt{true},\ (\mathtt{s(0)} \approx \mathtt{s(0)}) = \mathtt{true},\ (\mathtt{s}^2\mathtt{(0)} \approx \mathtt{s}^2\mathtt{(0)}) = \mathtt{true},$
$(\mathtt{s}^3\mathtt{(0)} \approx \mathtt{s}^3\mathtt{(0)}) = \mathtt{true},\ (\mathtt{s}^4\mathtt{(0)} \approx \mathtt{s}^4\mathtt{(0)}) = \mathtt{true},\ (\mathtt{s(x)} \approx \mathtt{s(y)}) = (\mathtt{x} \approx \mathtt{y}),$
$(\mathtt{s}^2\mathtt{(x)} \approx \mathtt{s}^2\mathtt{(y)}) = (\mathtt{x} \approx \mathtt{y}),\ (\mathtt{s}^3\mathtt{(x)} \approx \mathtt{s}^3\mathtt{(y)}) = (\mathtt{x} \approx \mathtt{y}),\ (\mathtt{s}^4\mathtt{(x)} \approx \mathtt{s}^4\mathtt{(y)}) = (\mathtt{x} \approx \mathtt{y}),$
$(\mathtt{s}^5\mathtt{(x)} \approx \mathtt{s}^5\mathtt{(y)}) = (\mathtt{x} \approx \mathtt{y}),\ \mathtt{double(s(x))} = \mathtt{double(x)},\ \mathtt{double(s(x))} = \mathtt{s(add(x,s(x)))},$
$\mathtt{double(s}^2\mathtt{(x))} = \mathtt{s}^2\mathtt{(add(x,s}^2\mathtt{(x)))},\ \mathtt{double(s}^3\mathtt{(x))} = \mathtt{s}^3\mathtt{(add(x,s}^3\mathtt{(x)))},$
$\mathtt{double(s}^4\mathtt{(x))} = \mathtt{s}^4\mathtt{(add(x,s}^4\mathtt{(x)))},\ \mathtt{double(0)} = \mathtt{0},\ \mathtt{double(s(0))} = \mathtt{s}^2\mathtt{(0)},$
$\mathtt{double(s}^2\mathtt{(0))} = \mathtt{s}^4\mathtt{(0)},\ \mathtt{double(s}^3\mathtt{(0))} = \mathtt{s}^6\mathtt{(0))},\ \mathtt{add(s(x),y)} = \mathtt{s(add(x,y))},$
$\mathtt{add(s}^2\mathtt{(x),y)} = \mathtt{s}^2\mathtt{(add(x,y))},\ \mathtt{add(s}^3\mathtt{(x),y)} = \mathtt{s}^3\mathtt{(add(x,y))},\ \mathtt{add(s}^4\mathtt{(x),y)} = \mathtt{s}^4\mathtt{(add(x,y))},$
$\mathtt{add(s}^5\mathtt{(x),y)} = \mathtt{s}^5\mathtt{(add(x,y))},\ \mathtt{add(0,x)} = \mathtt{x},\ \mathtt{add(s(0),x)} = \mathtt{s(x)},\ \mathtt{add(s}^2\mathtt{(0),x)} = \mathtt{s}^2\mathtt{(x)},$
$\mathtt{add(s}^3\mathtt{(0),y)} = \mathtt{s}^3\mathtt{(y)},\ \mathtt{add(s}^4\mathtt{(0),y)} = \mathtt{s}^4\mathtt{(y)}\}$

Now the system detects that the program rule $\mathtt{double(s(x))} = \mathtt{double(x)}$ is wrong. If the programmer replaces it with the correct equation $\mathtt{double(s(x))} = \mathtt{s(s(double(x)))}$, the program becomes correct and complete according to our conditions.

In the following section, a bug–correction strategy is presented which tries to modify the erroneous components of the initial code in order to infer a correct program. Then, we show how this mechanism can be combined within our diagnosis method in order to form a practical debugging system. We are only concerned with obtaining a partially correct program; completeness is not considered.

7 Correction of programs by example-guided unfolding

Inductive Logic Programming (ILP) is the field of Machine Learning concerned with the task of learning logic programs from positive and negative examples, generally in the form of ground literals [44]. A challenging subfield of Inductive Logic Programming (ILP) is known as *inductive theory revision* which is close to program debugging under the *competent programmer* assumption of [47]. In other words, the initial program is assumed to be written with the intention of being correct and, if it is not correct, then a close variant of it is. The debugging technique we develop tries to find such variant.

Formally, in our ILP approach to debugging, an initial hypothesis $\mathcal{R}$ is provided, under the constraint that the final hypothesis $\mathcal{R}^c$ should be as close a variation thereof as possible, in the sense that only the bugs of $\mathcal{R}$ should be detected, located and corrected, in order to produce $\mathcal{R}^c$. So we make the assumption that each component of the program appears there for some reason. This implies that if a piece of code is found to be incorrect, we cannot just throw it away; rather, we have just to repair it while keeping the part of the code which is right. However, note that our approach has an important difference w.r.t. [47] and similar work in that we don't require the user to interact with the debugger by either providing example evidences, answering correctness questions, establishing equivalence classes among the rules, or manually correcting code.

The automatic search for a new rule in an induction process can be performed either bottom-up (i.e. from an overly specific rule to a more general) or top-down (i.e. from an overly general rule to a more specific). We follow a deductive approach which is known as *example-guided unfolding* [19] which uses unfolding as specialization operator, focusing on discriminating positive from negative examples.

The problem can be formulated in our setting as follows:
Given: a program $\mathcal{R}$, such that $\mathcal{R}' \subseteq \mathcal{R}$ is incorrect w.r.t. the intended success set specification $\mathcal{I}_{ca}$
Find: a set of rules $\mathcal{X}$ such that the revised program $\mathcal{R}^c = (\mathcal{R} \setminus \mathcal{R}') \cup \mathcal{X}$ is (partially) correct w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$, that is, no incorrectness or incompleteness can be derived, where $\mathcal{I}^+$ and $\mathcal{I}^-$ are respectively the under– and over– approximation of the intended fixpoint semantics $\mathcal{I}_{\mathcal{F}}$ for $\mathcal{R}$. The revised program $\mathcal{R}^c$ will be called *correct* program.

7.1 Example-guided unfolding

An incorrect rule of program $\mathcal{R}$ overly covers a set E^+ of positive examples (that is, equations which belong to the intended success set semantics $\mathcal{I}_{ca}$) and a set E^- of negative examples (that is, equations in $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ which do not belong to $\mathcal{I}_{ca}$). What we need is some way of *guessing* a set of correct rules which cover all equations in E^+ and no equation in E^-. For example, let E^+ contain $\mathtt{even(0)} = \mathtt{true}$, $\mathtt{even(s^2(0))} = \mathtt{true}$, $\mathtt{even(s^4(0))} = \mathtt{true}, \ldots$ and let E^- contain $\mathtt{even(s(0))} = \mathtt{true}$, $\mathtt{even(s^3(0))} = \mathtt{true}$, $\mathtt{even(s^5(0))} = \mathtt{true}, \ldots$. Then $\{\mathtt{even(0)} = \mathtt{true},\ \mathtt{even(s^2(x))} = \mathtt{even(x)}\}$ would be a correct program.

Example–guided unfolding [1,19] is commonly applied to specialize the incorrect program $\mathcal{R}$ in order to exclude the negative examples without excluding the positive ones [19]. The basic idea of the method is as follows. We first specialize an incorrect rule r of $\mathcal{R}$ by unfolding one of the calls in the rhs or the body of the rule. Then, we delete some new rules from the program which have to do only with the negative examples. Hopefully, after repeating these two steps a number of times, we may find rules which can be removed from the program without harming its behavior on the positive examples. The motivation behind the method, formerly introduced in [18], is the following:

- Unfolding removes some internal nodes from the specialization tree. This tends to separate the positive from the negative examples, and also brings them closer to the root of the tree.

- If a negative example hangs directly from the root of the specialization tree, and its input rule r is not used elsewhere in the tree for a positive example, then the program can be specialized by deleting r.

We illustrate this by an example.

Example 7.1 Consider the program $\mathcal{R}$, consisting of the following rules $\mathcal{R} = \{\mathtt{f(x)} = \mathtt{g(x)},\ \mathtt{g(a)} = \mathtt{a},\ \mathtt{g(b)} = \mathtt{a}\}$. Let the intended success set specification $\mathcal{I}_{ca} = \{\mathtt{f(a)} = \mathtt{a},\ \mathtt{g(a)} = \mathtt{a},\ \mathtt{g(b)} = \mathtt{a}\}$. Hence, according to our abstract diagnosis method in section 5, the rule $r : \mathtt{f(x)} = \mathtt{g(x)}$ is incorrect. We fix $E^+ = \{\mathtt{f(a)} = \mathtt{a}\}$, and $E^- = \{\mathtt{f(b)} = \mathtt{a}\}$. Since r covers both positive and negative examples, unfolding is applied. Unfolding r upon $\mathtt{g(x)}$ replaces the rule with the following set of rules: $\mathcal{X} = \{\mathtt{f(a)} = \mathtt{a},\ \mathtt{f(b)} = \mathtt{a}\}$.

The first rule covers the positive example only, while the second rule covers the negative one. Since the negative example is directly connected to the root of the narrowing tree for $\mathtt{f(x)}$ in the program $\mathcal{R} \setminus \{r\} \cup \mathcal{X}$, it can be pruned by simply deleting the rule $\mathtt{f(b)} = \mathtt{a}$ from the program, which does not affect the positive example.

In the remainder of this section we particularize this method to our debugging framework.

7.2 Selection of the sets of examples

When a negative example is covered by the current version of the program, there is at least one rule which is responsible for the incorrect covering. In order to motivate our method, let us suppose that, as a first step we simply eliminate the incorrect rules from $\mathcal{R}$. By doing so, we would immediately get a partially correct program $\mathcal{R}^-$, since it does not contain incorrect rules. However, it might be incomplete w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$ as there can be equations which are covered in $\mathcal{I}$ [9], but not in $\mathcal{R}^-$. Equations of this kind are sensible positive examples, since the computed correction has to cover them. Hence we define the finite set $E^+ = \{e \mid e \in \mathcal{I}^- \text{ and } [\![e]\!] \not\in \gamma(T^{\sharp\varphi}_{\mathcal{R}^-}(\mathcal{I}^+)\}$.

Similarly, we let E^- define the set of equations e which, according to Proposition 6.1, allow the debugger to prove that each rule r of $\mathcal{R}'$ is incorrect (i.e. those equations e s.t. $e \in T^{\varphi}_{\{r\}}([\![\mathcal{I}^-]\!])$ and $e \not\in \gamma(\mathcal{I}^+)$).

7.3 Correction algorithm

In the context of functional logic programs, a natural way to specialize programs is to use a form of narrowing–driven unfolding, i.e. the expansion, by means of narrowing, of program subexpressions using the corresponding definitions (see [9] for a complete description). A complete characterization of un-

[9] Here we consider the specification of the intended (fixpoint) semantics to be provided by means of a program $\mathcal{I}$.

Input: $\mathcal{R}$, $\mathcal{I}$, and example sets E^+, E^-
Output: program $\mathcal{R}_c$ (a specialization of $\mathcal{R}$ w.r.t. E^+ and E^-).
Initialization $k := 0$; $\mathcal{R}_k := \mathcal{R}$.
While there is $e^- \in E^-$ s.t. $\mathcal{R}_k$ does not fail on e^- **do**
    begin
        Unfold the incorrect rule $r \in \mathcal{R}_k$
        Let $\mathcal{X}$ be the set of one-step resultants of r;
                Remove from $\mathcal{X}$ all those rules that do not occur
                in refutations of equations in E^+
        Let $\mathcal{R}_{k+1} = \mathcal{R}_k \setminus \{r\} \cup \mathcal{X}$
        Let $k := k + 1$
    end While
Return $\mathcal{R}_k$

Fig. 2. Basic algorithm for program correction

folding w.r.t. computed answers in functional logic languages with eager/lazy semantics can be found in [6,7]. Following [19], we use the unfolding operator for program correction as follows.

We formulate a basic algorithm which specializes programs w.r.t. positive and negative examples by applying the transformation rule unfolding together with rule removal. The (backbone of the) algorithm, to be used for program correction, is described in Figure 2. The procedure is inspired by [1], which is known to produce (with some extra outfit which is needed to specialize recursive definitions [19]) a correct specialization when the program originally covers all positive examples and some incorrect ones. The algorithm consists of one main loop which continues until no negative examples are incorrectly covered by the program rules. Informally, this is done in the following way. As long as there is a rule in the program that covers a negative example, it is checked whether it covers any positive example or not. If it covers no positive examples, then it is removed. Otherwise, it is is specialized w.r.t. the sets of positive and negative examples which are built as described above. Hence, incorrect rules are split in a number of rules that taken together should be equivalent to the rule which is split. Then, the process is repeated for those rules which still cover negative examples until a termination condition is reached.

Example 7.2 Assume that the following definition of the program `even/1` is given (note that the bug in this fragment causes the program to deliver any natural number as even):

$$\texttt{even(0)} = \texttt{true}. \qquad\qquad \texttt{even(s(x))} = \texttt{even(x)}.$$

together with the following positive and negative examples:

$$E^{+} = \{\texttt{even(0)} = \texttt{true},\ \texttt{even(s(s(0)))} = \texttt{true}\}$$

$$E^{-} = \{\texttt{even(s(0))} = \texttt{true}\}$$

The faulty rule which is responsible for the negative example is the second rule in the definition. According to our correction algorithm, the rule is unfolded and then discarded, since it is not used to refute any other negative example. Hence we get the desired, correct program

$$\texttt{even(0)} = \texttt{true} \qquad\qquad \texttt{even(s(s(x))} = \texttt{even(x)}$$

Unfortunately, unfolding and rule deletion are not generally sufficient for a complete correction method, as witnessed by the following example.

Example 7.3 Consider again the program $\mathcal{R}$, consisting of the rules for natural addition together with the rule $\{\texttt{double(x)} = \texttt{x} + \texttt{0}\}$. Let the intended specification $\mathcal{I}$ be the program which contains the rules for the addition together with the rules $\{\texttt{double(0)} = \texttt{0},\ \ \texttt{double(s(x))} = \texttt{s(s(double(x)))}\}$.

The rule $\{\texttt{double(x)} = \texttt{x} + \texttt{0}\}$ covers one positive example, $\texttt{double(0)} = \texttt{0}$ (note that it does not cover the other ones), and one negative example: $\texttt{double(s(x))} = \texttt{s(x)}$. So we unfold it getting the specialization $\{\texttt{double(0)} = \texttt{0},\ \ \texttt{double(s(x))} = \texttt{s(x+0)}\}$. Now, the second rule covers negative examples only, so it is simply removed; as a consequence, there is no chance to infer the correct rule $\texttt{double(x)} = \texttt{x} + \texttt{x}$ by example–guided unfolding.

As anyone acquainted with fold/unfold methodologies would expect, some mechanism for generalizing calls is needed. This was already pointed out in [18,19], where a solution to this problem was proposed which applies the transformation rules unfolding, definition, and folding. In the context of functional logic languages, a formal definition of these rules as well as proofs of their meaning preserving property (when they are combined) w.r.t. computed answers can be found in [6,7]. We plan to pursue an "example–guided fold/unfold" methodology for program correction as further research.

8 Implementation

The basic methodology presented so far has been implemented by a prototype system Buggy [2,4], which is written in SICStus Prolog and available at `http://www.dsic.upv.es/users/elp/soft.html`. The system description can be found in [4]. Here we just point out the major issues involved in such an endeavor.

Buggy includes a parser for a conditional functional logic language which can be executed by leftmost innermost narrowing [30], (innermost) basic narrowing [37] (a better strategy which does not require functions to be completely defined [17,37]), leftmost outermost narrowing [25] and needed narrowing [12] (these two lazy strategies are new w.r.t. the previous implementation reported in [2,4]). The module which computes the abstract program is an improving of a previous implementation (reported in [2]) by iteratively recomputing the loop check after abstracting each single rule. The debugger requires the user to fix some parameters, such as the narrowing strategy and the number n of iterations for approximating the success set. Then the errors are automatically found by the debugger. Then the user can try to correct the bug by specialization and he has also the possibility to indicate manually the corrections to be made on the wrong rules. In the current Buggy implementation, the intended semantics is entered as a program (that is, an executable specification), although this is just an implementation decision aimed at easing the experimentation and by no means should considered a drawback of the theoretical framework. Assisting the user in the task of (manually) entering the (under and over)-aproximations $\mathcal{I}^-$ and $\mathcal{I}^+$ is a possible extension not yet implemented.

In order to debug programs under the needed narrowing strategy we follow the transformational approach based on [36]. That is, functional nestings in the left-hand sides, lhs's, of the (IS) rules are removed by replacing them by a "case distinction" in the rhs of the rules. Then, the translated program is executed by using the leftmost outermost strategy which is strongly equivalent to needed narrowing on the translated programs, as mentioned in Section 2.

9 Conclusions

We have presented a generic scheme for the declarative debugging of functional logic programs. Our approach presented here is an elaboration of the preliminary method for abstract diagnosis introduced in [2,4]. We have presented a fixpoint semantics $\mathcal{T}_{\mathcal{R}}^{\varphi}$ for functional logic program $\mathcal{R}$, which is parametric w.r.t. the narrowing strategy φ. It works also for optimal strategies like needed narrowing. Our semantics allows us to model the set of computed answers in a bottom-up manner. Thus, it is a suitable basis for dataflow analyses based on abstract interpretation as we illustrated. Our dataflow analysis is built in a similar way to the concrete one and is parametric w.r.t. the narrowing strategy. We have extended an implementation of our system (Buggy) by using case expressions in order to be able to debug a modern language with needed narrowing.

Then we presented an algorithm for program correction based on example-guided unfolding and have analyzed some non trivial examples which can be corrected by using it. Our methodology does not require the user to provide a symptom (a known bug in the program) to start. Rather, our diagnoser discovers whether there is one such bug and then tries to correct it automatically, without asking the user to either provide further evidences or answer difficult questions about program semantics. We have discussed the successful experiments which have been performed with a prototypical implementation of our debugging system which is publicly available. Some topics for further research are to develop sophisticated heuristics to select what calls to apply unfolding upon, and how to handle cases when some positive examples are not included in the specification of the original program.

Acknowledgments We thank Stefano Marson, Sergio Lara–Pons and Santiago Escobar for their help in implementing the declarative debugger Buggy. We also thank the anonymous referees for their very valuable comments which have greatly improved the presentation. Part of this work was done when Francisco Correa was visiting DSIC-UPV.

References

[1] Z. Alexin, T. Gyimóthy, and H. Bostrom. Integrating Algorithmic Debugging and Unfolding Transformation in an Interactive Learner. In *Proc. ECAI'96*. John Wiley & Sons, 1996.

[2] M. Alpuente, F. Correa, and M. Falaschi. Declarative Debugging of Functional Logic Programs. In B. Gramlich and S. Lucas, editors, *Proc. of the International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, volume 57 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[3] M. Alpuente, F. Correa, and M. Falaschi. A Debugging Scheme for Functional Logic Programs. Technical Report DSIC-II/10/02, UPV, 2002. Available at URL: `http://www.dsic.upv.es/users/elp/papers.html`.

[4] M. Alpuente, F. Correa, M. Falaschi, and S. Marson. The Debugging System buggy. Technical Report DSIC-II/1/01, UPV, 2001. Available at URL: `http://www.dsic.upv.es/users/elp/papers.html`.

[5] M. Alpuente, M. Falaschi, and F. Manzo. Analyses of Unsatisfiability for Equational Logic Programming. *Journal of Logic Programming*, 22(3):221–252, 1995.

[6] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe folding/unfolding with conditional narrowing. In H. Heering M. Hanus and K. Meinke, editors, *Proc. of the International Conference on Algebraic and Logic Programming, ALP'97, Southampton (England)*, pages 1–15. Springer LNCS 1298, 1997.

[7] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs. In A. Middeldorp and T. Sato, editors, *Proc. of the 4th Fuji International Symposyum on Functional and Logic Programming, FLOPS'99, Tsukuba (Japan)*, pages 147–162. Springer LNCS 1722, 1999.

[8] M. Alpuente, M. Falaschi, and G. Vidal. A Compositional Semantic Basis for the Analysis of Equational Horn Programs. *Theoretical Computer Science*, 165(1):97–131, 1996.

[9] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.

[10] S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, pages 143–157. Springer LNCS 632, 1992.

[11] S. Antoy. Evaluation Strategies for Functional Logic Programming. In *Int'l Workshop on Reduction Strategies in Rewriting and Programming WRS'01*, volume 57. Electronic Notes in Theoretical Computer Science, 2001.

[12] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000. Preliminary version appeared in Proc. 21st ACM Symp. on Principles of Programming Languages, pages 268-279. ACM Press, New York, 1994.

[13] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[14] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *Proc. of First European Symp. on Programming, ESOP'86*, pages 119–132. Springer LNCS 213, 1986.

[15] D. Bert and R. Echahed. On the Operational Semantics of the Algebraic and Logic Programming Language LPG. In *Recent Trends in Data Type Specifications*, pages 132–152. Springer LNCS 906, 1995.

[16] D. Bert, R. Echahed, and B.M. Østvold. Abstract Rewriting. In *Proc. of Third Int'l Workshop on Static Analysis, WSA'93*, pages 178–192. Springer LNCS 724, 1993.

[17] P. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23, 1988.

[18] H. Bostrom and P. Idestam-Alquist. Specialization of Logic Programs by Pruning sld-trees. In *Proc of 4th Int'l Workshop on Inductive Logic Programming (ILP-94)*, pages 31–47, 1994.

[19] H. Bostrom and P. Idestam-Alquist. Induction of Logic Programs by Example–guided Unfolding. *Journal of Logic Programming*, 40:159–183, 1999.

[20] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M Hermenegildo, J. Maluszyński, and G. Puebla. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging-AADEBUG'97*, pages 155–170. U. of Linkoping Press, 1997.

[21] R. Caballero-Roldán, F.J. López-Fraguas, and J. Sánchez-Hernández. User's manual for Toy. Technical Report SIP-5797, UCM, Madrid (Spain), 1997.

[22] M. Codish, M. Falaschi, and K. Marriott. Suspension Analysis for Concurrent Logic Programs. In K. Furukawa, editor, *Proc. of Eighth Int'l Conf. on Logic Programming*, pages 331–345. The MIT Press, Cambridge, MA, 1991.

[23] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.

[24] M. Comini, G. Levi, and G. Vitiello. Declarative Diagnosis Revisited. In John W. Lloyd, editor, *Proceedings of the 1995 Int'l Symposium on Logic Programming*, pages 275–287. The MIT Press, 1995.

[25] R. Echahed. On completeness of narrowing strategies. In *Proc. of CAAP'88*, pages 89–101. Springer LNCS 299, 1988.

[26] R. Echahed. Uniform narrowing strategies. In *Proc. of ICALP'92*, pages 259–275. Springer LNCS 632, 1992.

[27] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[28] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 103(1):86–113, 1993.

[29] G. Ferrand. Error Diagnosis in Logic Programming, and Adaptation of E.Y.Shapiro's Method. *Journal of Logic Programming*, 4(3):177–198, 1987.

[30] L. Fribourg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. of Second IEEE Int'l Symp. on Logic Programming*, pages 172–185. IEEE, New York, 1985.

[31] D. Frutos-Escrig and M.I. Fernández-Camacho. On Narrowing Strategies for Partial Non-Strict Functions. In S. Abramsky and T. Maibaum, editors, *Proc. TAPSOFT'01*, pages 416–437. Springer LNCS 493, 1991.

[32] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377, 1991.

[33] M. Hanus. Compiling Logic Programs with Equality. In *Proc. of 2nd Int'l Workshop on Programming Language Implementation and Logic Programming*, pages 387–401. Springer LNCS 456, 1990.

[34] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[35] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[36] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999. Preliminary version in Proc. RTA'96, pages 138-152. Springer LNCS 1103.

[37] S. Hölldobler. *Foundations of Equational Logic Programming.* Springer LNAI 353, 1989.

[38] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.

[39] J. W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3, 1999.

[40] M.J. Maher. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. In *Proc. of Third IEEE Symp. on Logic In Computer Science*, pages 348–357. Computer Science Press, New York, 1988.

[41] M.J. Maher. Equivalences of Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, Los Altos, Ca., 1988.

[42] M.J. Maher. On Parameterized Substitutions. Technical Report RC 16042, IBM - T.J. Watson Research Center, Yorktown Heights, NY, 1990.

[43] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.

[44] S. Muggleton and L. de Raedt. Inductive Logic Programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[45] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.

[46] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. of Second IEEE Int'l Symp. on Logic Programming*, pages 138–151. IEEE, New York, 1985.

[47] E. Y. Shaphiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Massachusetts, 1982. ACM Distinguished Dissertation.

[48] G. Vidal. *Semantics-Based Analysis and Transformation of Functional Logic Programs*. PhD thesis, DSIC-UPV, Sept. 1996. En español.

[49] Y.H. You. Enumerating outer narrowing derivations for constructor-based term rewriting systems. *Journal of Symbolic Computation*, 7:319–343, 1989.

[50] F. Zartmann. Denotational Abstract Interpretation of Functional Logic Programs. In P. Van Hentenryck, editor, *Proc. of the 4th Int'l Static Analysis Symposium, SAS'97*, pages 141–159. Springer LNCS 1302, 1997.

A Appendix. A Buggy session

We show a Buggy session where we diagnose and correct the errors in two example programs.

Once the system is loaded, the following menu of commands is displayed with option *help.*:

```
*******************************************************************
       DECLARATIVE DEBUGGER FOR FUNCTIONAL LOGIC PROGRAMS.
*******************************************************************
*******************************************************************
*** help.      -> display menu                                  ***
*** clean.     -> clean program and specification               ***
*** loadprg.   -> read in program to be corrected from file     ***
*** loadspe.   -> load specification of the semantics from file ***
*** listprg.   -> display program                               ***
*** listspe.   -> display specification                         ***
*** debugger.  -> run debugger                                  ***
*** save.      -> save program to file                          ***
*** exit.      -> leave program                                 ***
*******************************************************************
>> clean.
>> loadprg.
Write the name of the program: 'evenprg.pl'.
Program loaded.

>> loadspe.
Write the name of the program: 'evenspe.pl'.
Specification loaded.

>> listprg.
even(0)=true.
even(s(X))=even(X).

>> listspe.
even(0)=true.
even(s(X))=odd(X).
odd(s(0))=true.
odd(s(s(X)))=odd(X).

>> debugger.
************************************************
          NARROWING STRATEGY  OPTIONS
************************************************
    1. Basic Narrowing.
    2. Leftmost-Innermost Narrowing.
    3. Leftmost-Outermost Narrowing.
    4. Needed Narrowing.
    Introduce the option:4.

Translating program to case version:
even(X)  =  case_5(X,0,true,s(X_1),even(X_1)).
            case_5(s(X),_,_,s(X),T) = T.
```

```
            case_5(0,0,T,_,_) = T.
(X ≈ Y) =  case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
                  s(X_1),case_3(Y,s(Y_1),(X_1 ≈ Y_1))).
            case_4(true,true,T,_,_,_,_) = T.
            case_4(0,_,_,0,T,_,_) = T.
            case_4(s(X),_,_,_,_,s(X),T) = T.
            case_1(true,true,T) = T.
            case_2(0,0,T) = T.
            case_3(s(X),s(X),T) = T.

Translating intended specification to case version:
even(X)  =  case_5(X,0,true,s(X_1),odd(X_1)).
            case_5(0,0,T,_,_) = T.
            case_5(s(X),_,_,s(X),T) = T.
odd(X)   =  case_7(X,s(X_1),case_6(X_1,0,true,s(X_2),odd(X_2))).
            case_7(s(X),s(X),T) = T.
            case_6(0,0,T,_,_) = T.
            case_6(s(X),_,_,s(X),T) = T.
(X ≈ Y) =  case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
                  s(X_1),case_3(Y,s(Y_1),(X_1 ≈ Y_1))).
            case_4(true,true,T,_,_,_,_) = T.
            case_4(0,_,_,0,T,_,_) = T.
            case_4(s(X),_,_,_,_,s(X),T) = T.
            case_1(true,true,T) = T.
            case_2(0,0,T) = T.
            case_3(s(X),s(X),T) = T.
************************************************
***          OVER APPROXIMATION   I+         ***
************************************************
true ≈ true.              0 ≈ 0.                    (X ≈ Y) = (X ≈ Y).
s(X) ≈ s(X).              odd(X) = odd(X).          even(X) = even(X).
case_1(X,Y,Z) = case_1(X,Y,Z).  %Iteration 0, omitting equations:
                                %case_n(X,...,Z) = case_n(X,...,Z), with 1 < n
even(X) = case_5(X,0,true,s(X_1),odd(X_1)).
case_5(0,0,T,_,_) = T.
case_5(s(X),_,_,s(X),T) = T.
odd(X) = case_7(X,s(X_1),case_6(X_1,0,true,s(X_2),♯))).
case_7(s(X),s(X),T) = T.
case_6(0,0,T,_,_) = T.
case_6(s(X),_,_,s(X),T) = T.
(X ≈ Y) = case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
                 s(X_1),case_3(Y,s(Y_1),♯)).
case_4(true,true,T,_,_,_,_) = T.
case_4(0,_,_,0,T,_,_) = T.
case_4(s(X),_,_,_,_,s(X),T) = T.
case_1(true,true,T) = T.
case_2(0,0,T) = T.
case_3(s(X),s(X),T) = T.
case_1(_,_,_) = bottom.           %Iteration 1, omitting equations:
                                  %case_n(_,..,_) = bottom.  1 < n
even(X) = bottom.
odd(X) = bottom.
(X ≈ Y) = bottom.
```

```
odd(s(X)) = case_6(X,0,true,s(X_1),♯).
even(s(X)) = odd(X).
even(0) = true.
(true ≈ Y) = case_1(Y,true,true).
(0 ≈ Y) = case_2(Y,0,true).
(s(X) ≈ Y) = case_3(Y,s(Y_1),♯).                         % Iteration 2.
odd(s(0)) = true.
odd(s(s(X))) = ♯.
odd(s(X)) = bottom.
even(s(X)) = case_7(X,s(X_1),case_6(X_1,0,true,s(X_2),♯))).
even(s(X)) = bottom.
(true ≈ true) = true.
(true ≈ Y) = bottom.
(0 ≈ 0) = true.
(0 ≈ Y) = bottom.
(s(X) ≈ s(Y) = ♯.
(s(X) ≈ Y) = bottom.                                      % Iteration 3.
even(s(s(X))) = case_6(X,0,true,s(X_1),♯).               % Iteration 4.
even(s(s(0))) = true.
even(s(s(s(X)))) = ♯.
even(s(s(X))) = bottom.                                   % Iteration 5.
************************************************
The Over Approximation  𝓘^+ has been computed in 5 iterations.
Enter the number K of iterations to compute  𝓘^- : 4.
************************************************
***          UNDER APPROXIMATION  𝓘^-          ***
************************************************
true ≈ true.             0 ≈ 0.                        (X ≈ Y) = (X ≈ Y).
s(X) ≈ s(X).             odd(X) = odd(X).              even(X) = even(X).
case_1(X,Y,Z) = case_1(X,Y,Z).  %Iteration 0, omitting equations:
                                %case_n(X,...,Z) = case_n(X,...,Z), with 1 < n
even(X) = case_5(X,0,true,s(X_1),odd(X_1)).
case_5(0,0,T,_,_) = T.
case_5(s(X),_,_,s(X),T) = T.
odd(X) = case_7(X,s(X_1),case_6(X_1,0,true,s(X_2),odd(X_2)))).
case_7(s(X),s(X),T) = T.
case_6(0,0,T,_,_) = T.
case_6(s(X),_,_,s(X),T) = T.
(X ≈ Y) = case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
               s(X_1),case_3(Y,s(Y_1),(X_1 ≈ Y_1))).
case_4(true,true,T,_,_,_,_) = T.
case_4(0,_,_,0,T,_,_) = T.
case_4(s(X),_,_,_,_,s(X),T) = T.
case_1(true,true,T) = T.
case_2(0,0,T) = T.
case_3(s(X),s(X),T) = T.
odd(X) = bottom.
even(X) = bottom.
(X ≈ Y) = bottom.
case_1(_,_,_) = bottom.           %Iteration 1, omitting equations:
                                  %case_n(_,..,_) = bottom.  1 < n
odd(s(X)) = case_6(X,0,true,s(X_1),odd(X_1)).
even(s(X)) = odd(X).
```

```
even(0) = true.
(true ≈ Y) = case_1(Y,true,true).
(0 ≈ Y) = case_2(Y,0,true).
(s(X) ≈ Y) = case_3(Y,s(Y_1),(X ≈ Y_1)).                % Iteration 2.
odd(s(0)) = true.
odd(s(s(X))) = odd(X).
odd(s(X)) = bottom.
even(s(X)) = case_7(X,s(X_1),case_6(X_1,0,true,s(X_2),odd(X_2))).
even(s(X)) = bottom.
(true ≈ true) = true.
(true ≈ Y) = bottom.
(0 ≈ 0) = true.
(0 ≈ Y) = bottom.
(s(X) ≈ s(Y) = (X ≈ Y).
(s(X) ≈ Y) = bottom.                                     % Iteration 3.
odd(s(s(X))) = case_7(X,s(X_1),case_6(X_1,0,true,s(X_2),odd(X_2))).
odd(s(s(X))) = bottom.
even(s(s(X))) = case_6(X,0,true,s(X_1),odd(X_1)).
(s(X) ≈ s(Y)) = case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
                       s(X_1),case_3(Y,s(Y_1),(X_1 ≈ Y_1))).
(s(X) ≈ s(Y)) = bottom.                                  % Iteration 4.
************************************************
***               T_R(I^-)                    ***
************************************************
true ≈ true.             0 ≈ 0.                        (X ≈ Y) = (X ≈ Y).
s(X) ≈ s(X).             even(X) = even(X).
case_1(X,Y,Z) = case_1(X,Y,Z).  %Iteration 0, omitting equations:
                                %case_n(X,...,Z) = case_n(X,...,Z), with 1 < n
even(X) = case_5(X,0,true,s(X_1),even(X_1)).
case_5(s(X),_,_,s(X),T) = T.
case_5(0,0,T,_,_) = T.
even(0) = true.
even(s(X)) = even(X).
(X ≈ Y) = case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
                  s(X_1),case_3(Y,s(Y_1),(X_1 ≈ Y_1))).
case_4(true,true,T,_,_,_,_) = T.
case_4(0,_,_,0,T,_,_) = T.
case_4(s(X),_,_,_,_,s(X),T) = T.
case_1(true,true,T) = T.
case_2(0,0,T) = T.
case_3(s(X),s(X),T) = T.
(X ≈ Y) = bottom.
(true ≈ Y) = case_1(Y,true,true).
(0 ≈ Y) = case_2(Y,0,true).
(s(X) ≈ Y) = case_3(Y,s(Y_1),(X ≈ Y_1)).
(true ≈ true) = true.
(true ≈ Y) = bottom.
(0 ≈ 0) = true.
(0 ≈ Y) = bottom.
(s(X) ≈ s(Y) = (X ≈ Y).
(s(X) ≈ Y) = bottom.
(s(X) ≈ s(Y)) = case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
                       s(X_1),case_3(Y,s(Y_1),(X_1 ≈ Y_1))).
```

```
(s(X) ≈ s(Y)) = bottom.
(s(0) ≈ s(Y)) = case_2(Y,0,true).
(s(s(X)) ≈ s(Y)) = case_3(Y,s(Y_1),(X ≈ Y_1)).
***********************************************
***         DIAGNOSIS:  ERROR FOUND         ***
***********************************************
The equation: even(s(X)) = even(X) in T_R(I^-) is incorrect.
The incorrect rule is: even(s(X)) = even(X).

            CORRECTION OPTIONS
***********************************************
1. Automatic correction.
2. Manual correction.

Enter option: 1.
***********************************************
***           GENERATING EXAMPLES           ***
***********************************************
Positive examples: even(s(s(0)) = true.
Negative examples: even(s(0)) = true.
***********************************************
***           CORRECTED PROGRAM             ***
***********************************************
even(0) = true.
even(s(s(X))) = even(X).

>> clean.
>> loadprg.
Write the name of the program: 'lastprg.pl'.
Program loaded.

>> loadspe.
Write the name of the program: 'lastspe.pl'.
Specification loaded.

>> listprg.
last([])=[].
last([X|Y])=last(Y).

>> listspe.
last(X)=Y :- append(Z,[Y])=X.
append([],B)=B.
append([A|B],C)=[A|append(B,C)].

>> debugger.
***********************************************
          NARROWING STRATEGY  OPTIONS
***********************************************
    1. Basic Narrowing.
    2. Leftmost-Innermost Narrowing.
    3. Leftmost-Outermost Narrowing.
    4. Needed Narrowing.
    Introduce the option:2.
```

```
**********************************************
***          OVER APPROXIMATION   I+        ***
**********************************************
[] = [].                          [A|B] = [A|B].
append(A,B) = append(A,B).      last(X) = last(X).              % Iteration 0
append([],B) = B.
append([A|B],C) = [A|#].                                        % Iteration 1
last([X]) = X.                                                  % Iteration 2
last([X|Y]) = #.                                                % Iteration 3
**********************************************
The Over Approximation  I+ has been computed in 3 iterations.
Enter the number K of iterations to compute  I- : 4.
**********************************************
***          UNDER APPROXIMATION  I-        ***
**********************************************
[] = [].                          [A|B] = [A|B].
append(A,B) = append(A,B).      last(X) = last(X).              % Iteration 0
append([],B) = B.
append([A|B],C) = [A|append(B,C)].                              % Iteration 1
last([A]) = A.
append([A],B) = [A|B].
last([A,B]) = B.
append([A,B|C],D) = [A,B|append(C,D)].                          % Iteration 2
append([A,B],C) = [A,B|C].
last([A,B,C]) = C.
append([A,B,C|D],E) = [A,B,C|append(D,E)].                      % Iteration 3
append([A,B,C],D) = [A,B,C|D].
append([A,B,C,D|E],F) = [A,B,C,D|append(E,F)].                  % Iteration 4
**********************************************
***                T_R(I-)                 ***
**********************************************
[] = [].                                [A|B] = [A|B].
append(A,B) = append(A,B).              last(X) = last(X).
last([]) = [].
**********************************************
***         DIAGNOSIS:  ERROR FOUND        ***
**********************************************
The equation: last([]) = []. in T_R(I-) is incorrect.
The incorrect rule is: last([]) = [].

            CORRECTION OPTIONS
**********************************************
1. Automatic correction.
2. Manual correction.

Enter option:  2.
Enter the position of the rule: 1.
Enter the correct clause: last([X])=X.
**********************************************
***                  T#_R(I+)              ***
**********************************************
last([A|B]) = last(B).
```

```
last([A,B,C|D]) = last(D).
last([A,B,C,D|E]) = last(E).
last([A,B|C]) = last(C).
last([A]) = A.
last([A,B,C]) = C.
last([A,B,C,D]) = D.
last([A,B]) = B.
[] = [].
[A|B] = [A|B].
last(X) = last(X).
************************************************
>> Analyzing completeness...
************************************************
***            UNCOVERED EQUATIONS          ****
************************************************
append([A,B,C,D|E],F) = [A,B,C,D|append(E,F)].
append([A,B,C],D) = [A,B,C|D].
append([A,B,C|D],E) = [A,B,C|append(D,E)].
append([A,B],C) = [A,B|C].
append([A,B|C],D) = [A,B|append(C,D)].
append([A],B) = [A|B].
append([A|B],C) = [A|append(B,C)].
append([],B) = B.

                    OPTION
************************************************
1. Adding one rule
2. Correcting one rule

Introduce the option: 1.
Introduce the position of the rule: 3.
Introduce the new rule: append([],B)=B.
...
************************************************
1. Adding one rule
2. Correcting one rule

Introduce the option: 1.
Introduce the position of the rule: 4.
Introduce the new rule: append([A|B],C)=[A|append(B,C)].
...
************************************************
>> Analyzing completeness...
************************************************
        ... the program is totally correct w.r.t. I+, I-.
        (no incorrectness or incompleteness bug found)
```