Theoretical Computer Science 411 (2010) 4055–4101

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

An integrated framework for the diagnosis and correction of rule-based programs☆

M. Alpuente [a], D. Ballis [b,*], F. Correa [c], M. Falaschi [d]

[a] *ELP-DSIC, Universidad Politécnica de Valencia, Valencia, Spain*
[b] *Università di Udine, Udine, Italy*
[c] *U. EAFIT, Medellin, Colombia*
[d] *Università di Siena, Siena, Italy*

ARTICLE INFO

Article history:
Received 23 July 2008
Received in revised form 21 June 2010
Accepted 20 July 2010
Communicated by G. Levi

Keywords:
Debugging
Narrowing
Functional logic programming
Program transformation

ABSTRACT

We present a generic scheme for the declarative debugging of programs that are written in rewriting-based languages that are equipped with narrowing. Our aim is to provide an integrated development environment in which it is possible to debug a program and then correct it automatically. Our methodology is based on the combination (in a single framework) of a semantics-based diagnoser that identifies those parts of the code that contain errors and an inductive learner that tries to repair them, once the bugs have been located in the program. We develop our methodology in several steps. First, we associate with our programs a semantics that is based on a (continuous) immediate consequence operator, $T_{\mathcal{R}}$, which models the answers computed by narrowing and is parametric w.r.t. the evaluation strategy, which can be eager or lazy. Then, 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 $T_{\mathcal{R}}$. In order to develop an effective debugging method, we approximate the *computed answers semantics* of $\mathcal{R}$ and derive a finitely terminating bottom-up abstract diagnosis method, which can be used statically. Finally, a bug-correction program synthesis methodology attempts to correct the erroneous components of the wrong code. We propose a hybrid, top-down (unfolding-based) as well as bottom-up (induction-based), correction approach that is driven by a set of evidence examples which are automatically produced as an outcome by the diagnoser. The resulting program is proven to be correct and complete w.r.t. the considered example sets. Our debugging framework does not require the user to provide error symptoms in advance or to answer difficult questions concerning program correctness. An implementation of our debugging system has been undertaken which demonstrates the workability of our approach.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

Functional logic languages combine the most important features of functional programming (expressivity of functions and types, higher-order functions, nested expressions, efficient reduction strategies, sophisticated abstraction facilities) and logic programming (unification, logical variables, partial data-structures, built-in search). The operational principle of

☆ This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2007-68093-C02-02, EAFIT project 173-000137, and the Italian MUR under grant RBIN04M8S8, FIRB project, Internationalization 2004.

* Corresponding author. Tel.: +39 0432 558484; fax: +39 0432 558499.
E-mail addresses: alpuente@dsic.upv.es (M. Alpuente), demis.ballis@dimi.uniud.it (D. Ballis), fcorrea@eafit.edu.co (F. Correa), moreno.falaschi@unisi.it (M. Falaschi).

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.07.009

integrated languages with a complete semantics is usually based on *narrowing* [67], which consists of the instantiation of variables in expressions, followed by a reduction step on the instantiated function call. *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, with innermost narrowing and needed narrowing being of main interest (see [78] for a survey.) Innermost narrowing was the basis of pioneer functional logic languages such as SLOG [71], LPG [29,32] and (a subset of) ALF [77], whereas needed narrowing is the core engine of modern functional logic languages like Curry [83] and Toy [96]. Nevertheless, innermost narrowing has recently regained much attention as it proves to be very useful for analysing security protocols and access control policies in rewriting-based languages such as Elan [35,91] and Maude [103,50].

The main purpose of this work is to provide a methodology for developing effective diagnosis and correction tools for functional logic programs or, more generally, for rewriting-based programs that can be executed by narrowing. Functional logic programming is now a mature paradigm and as such there exist modern environments that assist in the design, development and debugging of integrated programs. However, there is no theoretical foundation for integrating diagnosis and automated correction into a single unified framework. We believe that such an integration can be quite productive and hence develop useful techniques and new results for the process of automatically synthesizing correct programs.

Debugging programs with the combination of user-defined functions and logic variables is a difficult but important task which has received considerable interest in recent years, and different debugging techniques have been proposed. Abstract diagnosis [54,55] is a declarative debugging framework that extends the methodology in [68,117] (which is based on using the immediate consequence operator to identify bugs in logic programs) to diagnoses w.r.t. computed answers. An important advantage of this framework is that it is goal-independent and does not require the determination of symptoms in advance. In [8,9], we generalized the declarative diagnosis methodology of [54,55] to the debugging of wrong as well as missing answers of functional logic programs. Recently, other related abstractions of term rewriting systems have been proposed [6,80], which apply to nondeterministic TRSs but do not approximate computed answers.

This paper offers an up-to-date, comprehensive, and uniform presentation of the declarative debugging of functional logic programs as developed in [8,9]; a short overview can be found in [5]. We additionally address the problem of modifying incorrect components of the initial program in order to form an integrated debugging framework in which it is possible to detect program bugs and correct them automatically, which we first outlined in [4]. The generalization of [55] is far from trivial since we have to deal with the extra complexity derived from modeling computed answers while handling (possibly non-strict and partial) functions, nested calls, and lazy evaluation. In order to achieve this, we develop our framework in a stepwise manner.

(1) First, we define a (continuous) immediate consequence operator which models computed answers. This provides a fixpoint characterization of the operational semantics of integrated programs that is parametric w.r.t. the evaluation strategy, which can be either eager or lazy. Similarly to [75], the possibility of dealing with partial functions and infinite data structures leads to introducing two notions of equality, which are characterized by two sets of program rules. From the semantics viewpoint, the resulting construction gets more elaborate but also becomes richer in comparison to the scheme proposed in [55].

(2) 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 specification $\mathcal{I}$ may be complete or partial, which is useful for modular programming. Without loss of generality, we assume in this work that a complete specification is available which is expressed by another (simpler) program [8,9], but could be alternatively expressed by an assertion language [53] or by equation sets (in the case when it is finite). The diagnosis is based on the detection of *incorrect rules* and *uncovered equations*, which both have a bottom-up definition (in terms of a single application of the immediate consequence operator to the program specification). It is worth noting that no fixpoint computation is required since the semantics does not need to be computed.

The conditions that we impose on the considered programs 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 show how our methodology can be extended to optimal lazy evaluation strategies such as needed narrowing by using Hanus and Prehofer's transformation in [84], which compiles pattern matching into "case expressions". Our technique could be used as a basis for developing abstract debugging tools for different multi-paradigm languages equipped with a form of narrowing, including e.g., Curry [83], Elan [36], LPG [32,33], Maude [51,102,103], and Toy [96].

(3) In order to provide a practical implementation, we also present an effective debugging methodology that is based on abstract interpretation. Following an idea inspired by [55,54,42], we use *over* and *under* specifications $\mathcal{I}^+$ and $\mathcal{I}^-$ to correctly over (resp. under)- approximate the intended specification $\mathcal{I}$ of the success set. We then use these two sets respectively for the functions in the premises and the consequences of the immediate consequence operator, and by a simple static test we can determine whether some of the clauses are wrong. The method is sound in the sense that each error which is found and repaired by using $\mathcal{I}^+$, $\mathcal{I}^-$ is really a bug w.r.t. $\mathcal{I}$.

(4) Finally, we discuss our methodology for repairing some errors which is based on program specialization by example-guided unfolding. Informally, our correction procedure works as follows. Starting from an *overly general* program (that is, a program which proves all the positive examples as well as some negative ones), the algorithm unfolds the program and deletes program rules until a suitable specialization of the original program is reached which still implies all the

positive examples and does not prove any negative one. If the original wrong program does not initially prove all positive examples, we first invoke a bottom-up procedure, which "generalizes" the program in order to fulfil the applicability conditions. After introducing the new method, we prove its correctness and completeness w.r.t. the considered example sets.

Our prototype debugging system Buggy has been extended to work with the different instances of the framework discussed in this paper, which we illustrate by a number of examples. The implementation is endowed with inductive learning capabilities following our ideas for unfolding-based correction of programs from automatically generated examples. The positive and negative examples needed for this purpose are automatically derived from the approximations of the intended program semantics, which are computed by our abstract diagnosis method.

The idea of considering declarative specifications as programs goes back to the origins of declarative programming. In software development, a specification is often seen as the starting point for the subsequent program development and as the criterion for judging the correctness of the final software product. In general, it also happens that some parts of the software need to be improved during the software life cycle, e.g., in order to obtain better performance. Then the old programs (or large parts of them) can be usefully (and automatically) used as a specification for the new ones. This is not only common practice in logic programming but also in term-rewriting and functional languages, and a tool for checking the user's program w.r.t. suitable specifications is considered important in this context. For example, in QuickCheck [49], formal specifications are used to describe properties of Haskell programs (which are also written in Haskell) that are automatically tested. Recently, the convenience of the specification-oriented approach to program development based on prototype refinement has also been advocated in [23]. Nevertheless, we go one step further because, in our methodology, the intended specification can be automatically abstracted and used to automatically repair the programs under examination.

Related work

Finding program bugs is a long-standing problem in software construction. Unfortunately, the debugging support is rather poor for functional languages (see [121,85,100] and references therein), and there are no good general-purpose semantics-based debuggers available.

In the field of multi-paradigm declarative languages, standard trace debuggers are based on suitably extended box models which help to display the execution [81,25]. Due to the complexity of the operational semantics of (functional) logic programs, the information obtained by tracing the execution is difficult to understand. To improve understandability, a graphic debugger for the multi-paradigm language Curry is provided within a graphical environment [82] which visualizes the evaluation of expressions and is based on tracing. TeaBag [24] is both a tracer and a runtime debugger that is provided as an accessory of a Curry virtual machine which handles non-deterministic programs. For Mercury, a visual debugging environment is ViMer [48], which borrows techniques from standard tracers, such as the use of spypoints. In [109], the functional logic programming language NUE-Prolog was endowed with a more declarative, algorithmic debugger which uses the declarative semantics of the program and works in the style proposed by Shapiro [117]: an oracle (typically the user) is supposed to provide the debugger with error symptoms, as well as to correctly answer oracle questions driven by proof trees aimed at locating the actual source of errors. A similar declarative debugger for the functional logic language Escher is proposed in [94]. Unfortunately, when debugging the real code, the questions are often textually large and may be difficult to answer. Following the generic scheme which is based on proof trees of [108], a procedure for the declarative debugging of wrong answers in higher-order functional logic programs is proposed in [46]. This is a semi-automatic debugging technique where the debugger tries to locate the node in an execution tree which is ultimately responsible for a visible bug symptom. A declarative debugger (for wrong answers) based on this methodology was developed for the lazy functional logic language Toy [45] and adapted to Curry in [47]. The methodology in [46,45] includes a formalization of computation trees which is precise enough to prove the logical correctness of the debugger and also helps to simplify oracle questions. Missing answers are debugged in [44]. Braßel et al. [41] extended the idea (known as observational debugging [74]) of letting the programmer see the intermediate data structures that are passed between functions to functional logic languages, resulting in a new kind of algorithmic debugger. The application of declarative debugging, in Shapiro's style, to Maude was studied in [43,101].

As far as we know, none of the above-mentioned debuggers integrates both diagnosis and correction capabilities in a uniform and seamless way. As a matter of fact, program correction has scarcely been studied in the context of declarative programming. In [117], a theory revision framework for correction purposes has been proposed; however, it requires the user either to strongly interact with the debugger or to manually correct the code. Automated correction of faulty codes has been investigated in concurrent logic programming. In [1,2], a framework for the diagnosis and the correction of Moded flat GHC programs [120] has been developed. This framework exploits strong mode/typing and constraint analysis in order to locate bugs; then, symbols which are likely sources of error are syntactically replaced by other program symbols so that new slightly different programs (mutations) are produced. Finally, mutations are newly checked for correctness. This approach is essentially able to correct *near misses* (i.e., wrong variable/constant occurrences), but no mistakes involving predicates or function symbols can be repaired. Moreover, only modes and types are employed to come up with a corrected program; no finer semantic information is taken into consideration which might improve the quality of the repair. To the best of our knowledge, our approach is the first attempt to endow a declarative debugger with a repair methodology which is both *fully automatic* and *semantics-guided*.

To summarize, the main advantage of our approach to error debugging is that the debugger itself has a simple and elegant semantics, and that the programmer only needs to provide the intended interpretation of the erroneous program to debug

it. A limitation of our approach is that it does not cope with some advanced features such as higher order. This means that a programmer can use a diagnoser like ours for debugging only the first-order part of the program. Of course, some important practical problems well known for declarative diagnosis tools in LP, CLP, and FP also arise in our context, such as the problem of finding a compact representation of the semantics, which we have recently investigated in [6]. In spite of these difficulties, the prototype works reasonably well in all cases and we believe that it can be very useful for detecting many programming bugs in practice.

Plan of the paper

After some preliminaries in Section 2, we present a fixpoint characterization of functional logic computations in Section 3. This is done by means of a generic, narrowing-based immediate consequence operator $T^{\varphi}_{\mathcal{R}}$ which is parametric w.r.t. the narrowing strategy φ which can be either eager or lazy [9]. Based on $T^{\varphi}_{\mathcal{R}}$, we then define a fixpoint semantics which correctly models the answers and values computed by a narrower which uses the narrowing strategy φ. In the case of the eager strategy, it is enough to introduce a flattening transformation which eliminates nesting calls. This allows us to see the semantics (when it might be convenient) as an "efficient program" where it is still possible to execute goals just by using standard unification, as in [38]. However, the lazy strategy is more involved and we need to introduce two kinds of equality in the definition of $T^{\varphi}_{\mathcal{R}}$: the strict equality $\approx$ which models the equality on data terms, and the non-strict equality $=$ which holds even if the arguments are both undefined or partially defined, similarly to [75,105]. 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 incorrect rules and uncoveredness. Section 5 provides an abstract semantics that correctly approximates the fixpoint semantics of $\mathcal{R}$. The abstract semantics is computed by first removing from $\mathcal{R}$ those calls that may give rise to an infinite narrowing computation, which is done by computing a sort of *estimated narrowing dependency graph* for the CTRS $\mathcal{R}$. Using this semantics, our abstract diagnosis methodology is developed in Section 6. Section 7 endows the diagnosis method with a bug-correction program synthesis methodology which, after diagnosing the buggy program, tries to correct the erroneous components of the wrong code automatically. A prototype implementation of the method together with a debugging session is described in Section 8. Section 9 concludes and discusses some lines for future work. Appendix A describes a flattening procedure for equational goals along with the program transformation which we use to implement the needed-narrowing strategy. Proofs of all technical results are given in Appendix B. Finally, Appendix C provides some additional information regarding the debugging session given in Section 8.

2. Preliminaries

Conditional term rewriting systems (CTRSs) provide an adequate computational model for rule-based languages which allow the definition of functions by means of rules that can be activated by conditions in a set of data. In this paper, we consider the class of rule-based languages that combine a rule-based syntax for programs with the goal-solving operational principle of narrowing. This class includes functional logic programs as well as other rewriting-based languages equipped with the narrowing mechanism such as those mentioned above.

Let us briefly recall some known results about conditional rewrite systems [27,92] and functional logic programming (see [78,89] for extensive surveys). For simplicity, definitions are given in the one-sorted case. The extension to many-sorted signatures is straightforward, see [110]. Throughout this paper, V will denote a countably infinite set of variables and Σ denotes a non-empty, finite set of function symbols, or signature, each of which has a fixed associated arity. The signature Σ contains a special constant symbol $\bot$ intended to denote an undefined data value. Throughout the paper, we will use the following notation: lowercase letters from the end of the alphabet x, y, z, possibly with subindices, denote variables, and we often write $f/n \in \Sigma$ to denote that f is a function symbol of arity n. $\tau(\Sigma \cup V)$ and $\tau(\Sigma)$ denote the non-ground term algebra and the term 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 $\mathcal{H}$. A Σ-equation is either a pair of terms $s, t \in \tau(\Sigma \cup V)$ (denoted $s = t$), or the constants *true* or *fail*. $\mathcal{B}$ denotes the Herbrand base, namely the set of all ground equations that can be built with the elements of $\mathcal{H}$ (note that $\mathcal{B}$ disallows predicate symbols other than "=", similarly to [88]). A Herbrand interpretation $\mathcal{I}$ is a subset of $\mathcal{B}$. Identity of syntactic objects is denoted by $\equiv$.

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. $\mathcal{O}(t)$ (resp. $\overline{\mathcal{O}}(t)$) denotes the set of positions (resp. 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 $g \equiv (t_1 = t'_1, \ldots, t_n = t'_n)$ can be defined as follows: $\overline{\mathcal{O}}(g) = \{i.1.u \mid i \in \{1, \ldots, n\}, u \in \overline{\mathcal{O}}(t_i)\} \cup \{i.2.u \mid i \in \{1, \ldots, n\}, u \in \overline{\mathcal{O}}(t'_i)\}$. 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. We use $\bar{t}$ as a shorthand for $t_1, \ldots, t_n$.

Let *Eqn* denote the set of possibly existentially quantified finite sets of equations over terms [52]. We write $E \leq E'$ if E' logically implies E. Thus, *Eqn* is a lattice ordered by $\leq$ with bottom element *true* and top element *fail*. The elements of *Eqn* are often regarded as (quantified) conjunctions of equations (written as sequences) and treated modulo logical equivalence. An equation set is *solved* if it is either *fail* or it has the form $\exists y_1 \ldots \exists y_m.\ \{x_1 = t_1, \ldots, x_n = t_n\}$, where each x_i is a distinct

variable not occurring in any of the terms t_i and each y_i occurs in some t_j. Any set of equations E can be transformed into an equivalent one, $solve(E)$, which is solved. In our framework, existential quantifiers do not appear in concrete expressions like program rules or input equations, but only when we deal with semantic properties and abstract interpretations. A *substitution* is a mapping from the set of variables V into the set of terms $\tau(\Sigma \cup V)$. We restrict our interest to the set of idempotent substitutions over $\tau(\Sigma \cup V)$, which is denoted by *Sub*. A substitution θ is more general than σ, denoted by $\theta \leq \sigma$, if $\sigma = \theta\gamma$ for some substitution γ. We write $\theta_{\lceil s}$ 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 $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$. There is a natural isomorphism between substitutions $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ and unquantified equation sets in the solved form $\widehat{\theta} = \{x_1 = t_1, \ldots, x_n = t_n\}$. A set of equations E is unifiable if there exists θ such that, for all $s = t$ in E, we have $s\theta \equiv t\theta$, and θ is called a *unifier* of E. We let $mgu(E)$ denote the *most general unifier* of the (unquantified) equation set E [98]. When E is not unifiable, by abuse we define $mgu(E) = fail$. 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 $\{s_1 = s'_1, t_1 = t'_1, \ldots, s_n = s'_n, t_n = t'_n\}$. Given two substitutions θ_1 and θ_2, we define the *parallel composition* of θ_1 and θ_2 as $\theta_1 \Uparrow \theta_2 = mgu(\hat{\theta}_1 \cup \hat{\theta}_2)$.

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 $(\lambda \to \rho \Leftarrow C)$, $\lambda, \rho \in \tau(\Sigma \cup V)$ and $\lambda \notin V$. The condition C is a (possibly empty) finite 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 or ρ 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 TRS is a CTRS whose rules have no conditions. 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 $\Sigma = \mathcal{C} \uplus \mathcal{F}$, where $\mathcal{F} = \{f \mid (f(t_1, \ldots, t_n) \to r \Leftarrow C) \in \mathcal{R}\}$ and $\mathcal{C} = \Sigma \setminus \mathcal{F}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{F}$ are called *defined functions*. The elements of $\tau(\mathcal{C} \cup \mathcal{V})$ are called *constructor terms*. A *constructor* substitution $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ is a substitution such that each t_i, $i = 1, \ldots, n$ is a constructor term. A term is linear if it does not contain multiple occurrences of the same variable. A CTRS is left-linear if the left-hand sides of all rules are linear terms. A *pattern* is a term of the form $f(\bar{d})$ where $f/n \in \mathcal{F}$ 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*[1] *rewrites* to a term t, $s \to_{\mathcal{R}} t$, if there exist $u \in \overline{O}(s)$, $(\lambda \to \rho \Leftarrow s_1 = t_1, \ldots, s_n = t_n) \ll \mathcal{R}$, and a substitution σ such that $s_{|u} \equiv \lambda\sigma$, $t \equiv s[\rho\sigma]_u$, and there exists a term w_i such that $s_i\sigma \to^*_{\mathcal{R}} w_i$ and $t_i\sigma \to^*_{\mathcal{R}} w_i$, where $\to^*_{\mathcal{R}}$ is the transitive and reflexive closure of $\to_{\mathcal{R}}$ [76,104]. The term $s_{|u}$ is said to be a *redex* of s. When no confusion can arise, we omit the subscript $\mathcal{R}$. When we want to emphasize the rule and the redex chosen for the rewrite step, we write $s \overset{r,u}{\to} t$. The length of a rewrite sequence $\mathcal{D} : t_0 \to t_1 \to t_2 \to \cdots t_n$ is the number n of rewrite steps occurring in $\mathcal{D}$ and is denoted by $|\mathcal{D}|$. A term s is a *normal form* if there is no term t with $s \to_{\mathcal{R}} t$. A CTRS $\mathcal{R}$ is *noetherian* if there are no infinite sequences of the form $t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots$. 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 common term. The program $\mathcal{R}$ is said to be canonical if the binary one-step rewrite relation $\to_{\mathcal{R}}$ defined by $\mathcal{R}$ is noetherian and confluent [92]. The rewrite relation on terms can be extended to equations in the natural way, by considering the equality as a standard function symbol which is modeled by a set of rewrite rules (see Section 2.1).

2.1. Functional logic programming

Functional logic languages are extensions of functional languages with principles derived from logic programming [57,93, 105,113]. The computation mechanism of functional logic languages is based on *narrowing* [67,118], a generalization of term rewriting where unification replaces matching: both the rewrite rule and the term to be rewritten can be instantiated. Under the narrowing mechanism, functional programs behave like logic programs: narrowing solves equations by computing solutions with respect to a given CTRS, which is henceforth called the "program".

Definition 1 (*Narrowing*)**.** Let $\mathcal{R}$ be a program and g be a goal. We say that g conditionally narrows into g' if there exists a position $u \in \overline{O}(g)$, a standardized apart variant $r \equiv (\lambda \to \rho \Leftarrow C)$ of a rewrite rule in $\mathcal{R}$, and a substitution σ such that σ is the most general unifier of $g_{|u}$ and λ, and $g' \equiv (C, g[\rho]_u)\sigma$.

We write $g \overset{u,r,\sigma}{\leadsto} g'$. The relation $\leadsto$ is called (*unrestricted* or *ordinary*) conditional narrowing. Sometimes, we simply write $g \overset{r,\sigma}{\leadsto} g'$, or $g \overset{\sigma}{\leadsto} g'$.

[1] This definition corresponds to the TRSs of Type II (join systems) in the terminology of Bergstra and Klop [92]. As noted in [92], $\to_R$ is well defined since all conditions are positive. The conditional rewrite relation $\to_R$ induced by a CTRS $\mathcal{R}$ can be defined inductively as $\to_R = \bigcup_{j\geq 0} \to_{R_j}$, where $R_0 = \emptyset$ and $R_{j+1} = \{l\sigma \to r\sigma \mid (l \to r \Leftarrow s_1 = t_1, \ldots, s_n = t_n) \ll \mathcal{R}$ and for all $i \in \{1, \ldots, n\}$, there exists a term w_i such that $s_i\sigma \to^*_{\mathcal{R}_j} w_i$ and $t_i\sigma \to^*_{\mathcal{R}_j} w_i\}$ [104].

The extension of a CTRS $\mathcal{R}$ with the rewrite rules for dealing with the equality is denoted by $\mathcal{R}_+$. In the case of unrestricted narrowing, $\mathcal{R}_+$ denotes $\mathcal{R} \cup \{x = x \to true\}, x \in V$. This allows us to treat syntactical unification as a narrowing step, by using the rule $(x = x \to true)$ to compute *mgu*'s. Then, $s = t \overset{\sigma}{\leadsto} true$ holds iff $\sigma = mgu(\{s = t\})$.

Definition 2 (*Narrowing derivation*)**.** A narrowing *derivation* for g in $\mathcal{R}$ is defined by $g \overset{\theta}{\leadsto}{}^* g'$ iff $\exists\theta_1, \ldots, \exists\theta_n.\ g \overset{\theta_1}{\leadsto} \cdots \overset{\theta_n}{\leadsto} g'$ and $\theta = \theta_1 \ldots \theta_n$. We say that the derivation has length n. If $n = 0$, then $\theta = \epsilon$. We use the symbol $\top$ as a generic notation for sequences of the form *true*, ..., *true*. A *successful* derivation (or refutation) in $\mathcal{R}$ is a narrowing derivation $g \overset{\theta}{\leadsto}{}^* \top$ in $\mathcal{R}^+$, and $\theta_{|Var(g)}$ is called a *computed answer substitution (cas)* for g in $\mathcal{R}$.

The *narrowing* mechanism is a powerful tool for constructing complete equational unification algorithms for useful classes of TRSs, including canonical TRSs [90]. Similarly to logic programming, completeness means the ability to compute representatives of all solutions for one or more equations.

Example 3. Consider the following program $\mathcal{R}$ which defines the `last` element of a list in a logic programming style, by using the list concatenation function `append` (list constructors are [ ] (empty list) and [_|_] (non-empty list)):

$$\begin{array}{lrl} r_1: & \texttt{last(xs)} & \to \texttt{y} \Leftarrow \texttt{append(zs, [y])} = \texttt{xs} \\ r_2: & \texttt{append([ ], xs)} & \to \texttt{xs} \\ r_3: & \texttt{append([x|xs], ys)} & \to \texttt{[x|append(xs, ys)]} \end{array}$$

Given the input goal $\texttt{last(ys)} = \texttt{0}$, narrowing is able to compute in $\mathcal{R}$ infinitely many answers of the form $\{\texttt{ys} \mapsto \texttt{[0]}\}, \{\texttt{ys} \mapsto \texttt{[\_|0]}\}, \ldots$. For instance, the first answer is computed by the following narrowing derivation (at each step, the narrowing relation $\leadsto$ is labelled with the applied substitution[2] and rule, and the reduced subterm is underlined):

$$\begin{array}{rl} \underline{\texttt{last(ys)}} = \texttt{0} & \\ \overset{r_1,\{\texttt{ys/xs}\}}{\leadsto} & \underline{\texttt{append(zs, [y])}} = \texttt{xs}, \texttt{y} = \texttt{0} \\ \overset{r_2,\{\texttt{zs/[ ]}\}}{\leadsto} & (\texttt{[y]} = \texttt{xs}, \underline{\texttt{y} = \texttt{0}}) \\ \overset{(\texttt{x=x}\to\texttt{true}),\{\texttt{y/0}\}}{\leadsto} & (\underline{\texttt{[0]} = \texttt{xs}}, \texttt{true}) \\ \overset{(\texttt{x=x}\to\texttt{true}),\{\texttt{xs/[0]}\}}{\leadsto} & \top \end{array}$$

For more in-depth understanding of the power and subtleties of narrowing, please refer to [23,79].

2.1.1. Narrowing strategies

Since unrestricted narrowing has quite a large search space,[3] 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 input expression g (e.g. a goal 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 outer evaluation of orthogonal (i.e. left-linear and overlap-free), nonterminating rules and those of inner or eager evaluation with terminating, non-orthogonal rules. Also, under the eager strategy, programs are required not to contain extra-variables, that is, each program rule $\lambda \to \rho \Leftarrow C$ satisfies $Var(\rho) \cup Var(C) \subset Var(\lambda)$, whereas the weaker condition $Var(\rho) \subset Var(\lambda) \cup Var(C)$ is demanded in lazy programs. A survey of results about the completeness of narrowing strategies can be found in [22,21,60,61,78]. To simplify our notation, we let $\mathbb{R}_\varphi$ denote the class of programs that satisfy the conditions for the completeness of the strategy φ. Under these conditions, the narrowing strategies defined below only compute constructor substitutions.

We need the following notions and notation. An *innermost term* is 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 that 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 $i < j$.

Definition 4 (*Narrowing strategies*)**.** 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.

We formulate a conditional narrower with strategy $\varphi, \varphi \in \{inn, out\}$, 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 the set of rules Eq^φ models the equality on terms.

2 Substitutions are restricted to the input variables.

3 Actually, there are three sources of non-determinism in (conditional) narrowing: the choice of the equation within the goal, the choice of the redex within the equation, and the choice of the rewrite rule.

Namely, Eq^{out} is the set of rules that define the validity of equations as a *strict equality* between terms which is appropriate when computations may not terminate [105]:

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

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

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

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 an input 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.

It is known 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 $\mathtt{f(g(x),\,x)} \approx \mathtt{true}$.

As for canonical TRSs, the completeness of a narrowing strategy is achieved by enforcing the following *uniformity* condition [60,61,72,110]: a narrowing strategy φ is *uniform* iff for any goal g and every grounding normalized substitution σ for g (i.e. a substitution that only contains terms in normal form such that $g\sigma$ is ground), the subterm of $g\sigma$ occurring at any position in $\varphi(g)$ is reducible. Note that in the program $\mathcal{R}$ above, the strategy $\varphi = out$ does not satisfy the uniformity principle since the term $\mathtt{g(x)}$ is not reducible when instantiated with the grounding substitution $\{\mathtt{x/a}\}$.

The uniformity condition for canonical TRSs has been extended to conditional CTRSs in [30] by extending to goals the rewrite relation $\to_{\mathcal{R}}$. Namely, they use the non-deterministic rewriting relation without evaluation of conditions $\mapsto_{\mathcal{R}}$ first defined by Bockmayr and Werner in [34], which provides a direct correspondence between conditional narrowing and conditional rewriting. We write $g \mapsto_{\mathcal{R}} g'$ if there exist a position $p \in \varphi(g)$, a rule $\lambda \to \rho \Leftarrow C \ll \mathcal{R}$ and a substitution σ such that (i) $g_{|p} = \lambda\sigma$, (ii) $g' = (C\sigma, g[\rho\sigma]_p)$. Then, given a constructor-based program $\mathcal{R}$ such that $\mapsto_{\mathcal{R}}$ is canonical,[4] a sufficient condition for uniformity is given by [30,60]: (i) functions in $\mathcal{F}$ are completely defined (i.e., the set of normal ground terms is $\tau(\mathcal{C})$), and (ii) left-hand sides of rules in $\mathcal{R}$ are pairwise *not strictly subunifiable*, i.e., two subterms at the same position of two left-hand sides are not unifiable by a nontrivial *mgu* (i.e., a mgu θ such that $\theta\theta^{-1} \neq \epsilon$). 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 $\{\mathtt{y/c}\}$. We denote by $\mathbb{R}^u$ the class of *uniform* CTRSs that satisfy the conditions (i) and (ii) above. For the case $\varphi = out$, we also require left-linearity of $\mathcal{R}$.

Since the not strictly subunifiable requirement is not satisfied by certain programs, [60] contains a method to transform a program satisfying (i) into a program satisfying (i) and (ii) (see [60] for details).

The following example borrowed from [13] illustrates the transformation of a non-uniform program to a uniform one.

Example 5. Given the following (non-uniform) program $\mathcal{R}$:

$$\begin{array}{rcll} \mathtt{f(0,\,0)} & \to & \mathtt{0} & (R_1) \\ \mathtt{f(s(x),\,0)} & \to & \mathtt{s(0)} & (R_2) \\ \mathtt{f(x,\,s(0))} & \to & \mathtt{0} & (R_3) \\ \mathtt{f(x,\,s(s(y)))} & \to & \mathtt{s(s(0))} & (R_3) \end{array}$$

we can get the following uniform program by applying the transformation procedure of [60]:

$$\begin{array}{rcl} \mathtt{f(x,\,0)} & \to & \mathtt{h(x)} \\ \mathtt{f(x,\,s(y))} & \to & \mathtt{g(y)} \end{array} \qquad \begin{array}{rcl} \mathtt{h(0)} & \to & \mathtt{0} \\ \mathtt{h(s(x))} & \to & \mathtt{s(0)} \\ \mathtt{g(0)} & \to & \mathtt{0} \\ \mathtt{g(s(y))} & \to & \mathtt{s(s(0))}, \end{array}$$

where g and h are new function symbols not appearing in the signature of the original program.

Innermost narrowing is the foundation of several functional logic programming languages like SLOG [71], LPG [29,32] and (a subset of) ALF [77]. Also, the multi-paradigm language Maude [51,102] is equipped with a (kind of) innermost narrowing strategy (called *variant narrowing* [63,62]) that is part of an equational unification procedure. Moreover, reachability analyses for programs written in Maude rely on the so-called *topmost theories* [103], where the innermost strategy is often advantageous. Recently, the notion of *strategic narrowing* has been proposed as the main mechanism for the analysis of security policies in the strategy language Elan, relying on the confluence, termination and sufficient completeness of the underlying rewrite system [91]. In this context, innermost narrowing, innermost priority narrowing (i.e., innermost narrowing with a partial ordering on the program rules) and outermost narrowing have proven to be of prime interest [91].

[4] In the case of unconditional programs, this boils down to require canonicity of $\to_{\mathcal{R}}$.

Modern functional logic languages like Curry [83] and Toy [96] are based on lazy evaluation principles instead, which delay the evaluation of function arguments until their values are needed to compute a result. This allows one to deal with infinite data structures and avoids some unnecessary computations [78]. The strategy of contemporary implementations of lazy functional logic languages is needed narrowing. Needed narrowing can be easily and efficiently implemented by means of a transformation proposed in [84], which preserves the answers computed by needed narrowing in the original program. Thanks to the possibility of using this transformation, we do not 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. completeness. For the sake of completeness, the transformation of [84] can be found in Appendix A.

3. The semantic framework

In this section, we develop a compositional, *fixpoint semantics* $\mathcal{F}_\varphi(\mathcal{R})$ for program $\mathcal{R}$, which models successful as well as partial (unfinished) computations and is parametric w.r.t. the evaluation strategy. Then, we provide a subset $\mathcal{O}^{ca}_\varphi(\mathcal{R})$ of the denotation $\mathcal{F}_\varphi(\mathcal{R})$ that only models the successful computations. The *computed answer semantics* $\mathcal{O}^{ca}_\varphi(\mathcal{R})$ allows us to formalize the correctness and completeness of a program w.r.t. a given specification, whereas the former, fixpoint semantics $\mathcal{F}_\varphi(\mathcal{R})$ is used for the diagnosis. In order to formalize the precise relationship between these two semantics, we distinguish between two kinds of partial computations (intermediate computations and nonterminating computations) and we introduce an intermediate, auxiliary semantics $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ which models success as well as non-termination. The *computer answer (fixpoint) semantics* $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ is instrumental and provides a useful, purely syntactic characterization of the narrowing computations and can be automatically approximated, as we will show in Section 4.

Let $\lfloor\mathcal{R}\rfloor$ denote the set of ground instances of the rules of $\mathcal{R}$. For canonical $\mathcal{R}$, the standard (Herbrand) semantics of $\mathcal{R}$, which is given by the "ground success set" (i.e., the set of all ground equations $s = t$ such that s and t have a common $\mathcal{R}$-normal form), can be reconstructed as the least fixpoint $T_\mathcal{R} \uparrow \omega$ of the following immediate consequence operator $T_\mathcal{R}$, which is continuous on the complete lattice of Herbrand interpretations ordered by set inclusion [89].

$$T_\mathcal{R}(I) = \{t = t \in \mathcal{B}\} \cup \{e \in \mathcal{B} \mid (\lambda \to \rho \Leftarrow C) \in \lfloor\mathcal{R}\rfloor,\ \{e[\rho]_u\} \cup C \subseteq I,\ u \in \overline{O}(e),\ e_{|u} = \lambda\}$$

Informally, $T_\mathcal{R}(I)$ contains the set of all ground instances of the reflexivity axiom and the set of all ground equations that can be 'constructed' from elements of the Herbrand interpretation I by replacing one occurrence of the right-hand side of the head of a rule in $\mathcal{R}$ by the corresponding left-hand side.

In order to formulate a semantics for functional logic programs that models computed answers, the usual Herbrand base has to be extended to the set of all (possibly) non-ground equations [65,66].

Definition 6 (*V-Herbrand Universe, V-Herbrand φ-base*)**.** $\mathcal{H}_V$ denotes the *V-Herbrand universe* that allows variables in its elements and is defined as $\tau(\Sigma \cup V)/_{\cong}$, where $\cong$ is the equivalence (renaming) 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 φ-base*, namely, the set of all (unquantified) equations $s =_\varphi t$ modulo renaming, where $s, t \in \mathcal{H}_V$.

Note that the case when $\varphi = inn$, all equations in $\mathcal{B}_V$ have the form $t = s$, whereas for the lazy strategy, we need to distinguish between two kinds of equality: the strict equality $\approx$ which models the equality on data terms, and the non-strict equality $=$ which holds even if the arguments are both undefined or partially defined, similarly to [75,105]. That is, equations have the form $t = s$ or $t \approx s$ when $\varphi = out$. Note that the standard Herbrand base $\mathcal{B}$ is equal to $\lfloor\mathcal{B}_V\rfloor$. The ordering on $\mathcal{H}_V$ induces an ordering on $\mathcal{B}_V$, namely $s' =_\varphi t' \leq s =_\varphi t$ if $s' \leq s$ and $t' \leq t$. The power set of $\mathcal{B}_V$ is a complete lattice under set inclusion. A V-Herbrand φ-interpretation $\mathcal{I}$ is a subset of $\mathcal{B}_V$, which we simply refer to as Herbrand interpretation.

The idea of using syntactic domains for describing program semantics, and in particular the use of non-ground atoms in the denotation, is inspired in the literature of logic programming (see [38]) where it is commonly used to capture various observables like *computed answers* or call patterns in a goal-independent way, so that goals can be simply solved in the semantics by syntactic unification.

Following [38], we are interested in developing a semantics $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ for program $\mathcal{R}$ 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 standardized apart. In order for the term structure to be directly accessible to unification, equations in the goal have to be flattened first, i.e., subterms need to be unnested.

Definition 7 (*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 $d, d_1, \ldots, d_n \in \tau(\mathcal{C} \cup V)$. 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 [75,105]). 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 $\Phi_{\mathcal{R}}$. This is novel w.r.t. the standard literature, where the non-strict case is not considered (see e.g. in [89] the characterization of some equational inference rules such as ordinary narrowing and paramodulation).

Any sequence of equations E can be transformed into a flat one, $flat_{\varphi}(E)$, which is equivalent in the following sense: E is equivalent to the existential quantification of $flat_{\varphi}(E)$ w.r.t. the new, auxiliary variables introduced by the flattening transformation. By abuse of notation, we disregard the existential quantification of these new variables as we only consider the unification problem $flat_{\varphi}(E)$ for the variables in E. The *flattening* procedures for equation sets which produce flat goals w.r.t. *inn* and *out*, respectively, can be found in [37,75]. For the sake of completeness, we recall them in Appendix A, where we compact them as two cases of a generic flattening transformation.

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 that is completely independent of the goal [17,65]. In the following section, we provide a fixpoint characterization of the operational semantics of integrated programs that is also parametric w.r.t. the evaluation strategy, which can be either eager or lazy.

As mentioned above, we are going to introduce three different program denotations $\mathcal{F}_{\varphi}(\mathcal{R})$, $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ and $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ for program $\mathcal{R}$. The 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 least fixpoint of an immediate consequences operator $T^{\varphi}_{\mathcal{R}}$. A subset of the denotation $\mathcal{F}_{\varphi}(\mathcal{R})$ is the *computed answer (fixpoint) semantics* $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$, which is obtained from $\mathcal{F}_{\varphi}(\mathcal{R})$ by removing the equations that model intermediate computations (i.e., those equations $f(\bar{t}) = s$ where s "has not reached its value") and is the only semantics that allows us to execute (nontrivial) goals g by simply unifying $flat(g)$ with the equations in the denotation and obtain the very same answers as computed by narrowing. The semantics $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ is purely instrumental; also note that it still models nonterminating functions, which are denoted by $\bot$. Finally, the operational *success set semantics* $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ just catches successful derivations, that is, it only catches the computed answers. Therefore, we have $\mathcal{O}^{ca}_{\varphi}(\mathcal{R}) \subseteq \mathcal{F}^{ca}_{\varphi}(\mathcal{R}) \subseteq \mathcal{F}_{\varphi}(\mathcal{R})$.

3.1. Fixpoint semantics

Now we consider a generic immediate consequence operator $T^{\varphi}_{\mathcal{R}}$ which models computed answers w.r.t. φ. In non-strict languages, if the compositional character of meaning has to be preserved in the 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 [75,105] or, equivalently, the infinite data structure defined as the least upper bound of this class. Following [75,105], the constant symbol $\bot \in \Sigma$ is used to approximate the value of expressions which would otherwise be undefined.

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 $\Im^{\varphi}_{\mathcal{R}}$ 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. These *functional reflexivity axioms* play an important role in defining the fixpoint semantics of $\mathcal{R}$.

Definition 8 (*Immediate consequence operator*)**.** Let $\mathcal{I}$ be a φ-Herbrand interpretation and $\mathcal{R} \in \mathbb{R}_{\varphi}$. Then,

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

where $\mathcal{R}^{inn}_{++} = \mathcal{R}$, whereas $\mathcal{R}^{out}_{++} = \mathcal{R}^{out}_{+} \cup \{f(x_1, \ldots, x_n) \to \bot \mid f/n \in \mathcal{F}$ and $x_1, \ldots, x_n \in V\}$, and $\mathcal{I}^{c} = \{l =_{\varphi} r \in \mathcal{I} \mid r$ is a constructor term$\}$.

In the case when $\varphi = out$, the rules $f(x_1, \ldots, x_n) \to \bot$ are necessary to associate a denotation with all input expressions, including those that yield non-termination. Note that only equations with equality symbol $=$ are derived at each application of the immediate consequences operator.

We are ready to formalize our notion of fixpoint semantics in the fixpoint style. As usual, we consider the chain of iterations of $T^{\varphi}_{\mathcal{R}}$ starting from bottom, by defining $T^{\varphi}_{\mathcal{R}} \uparrow 0 = \emptyset$; $T^{\varphi}_{\mathcal{R}} \uparrow (k+1) = T^{\varphi}_{\mathcal{R}}(T^{\varphi}_{\mathcal{R}} \uparrow k)$, for $k \geq 0$; and $T^{\varphi}_{\mathcal{R}} \uparrow \omega = \bigcup_{k \geq 0} T^{\varphi}_{\mathcal{R}} \uparrow k$.

The following proposition is instrumental to define the fixpoint semantics.

Proposition 9. *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 10 (*Fixpoint semantics*)**.** The least fixpoint semantics of a program $\mathcal{R}$ in $\mathbb{R}_{\varphi}$ is defined as $\mathcal{F}_{\varphi}(\mathcal{R}) = lfp(T^{\varphi}_{\mathcal{R}})$, $\varphi \in \{inn, out\}$.

Let us now illustrate the fixpoint semantics by some examples.

Example 11. Let $\varphi = inn$, and consider the following program R that defines the predecessor function `pre` for natural numbers that are generated by means of function `nat`:

$$\begin{array}{l} \mathtt{pre(s(x))} \to \mathtt{x} \Leftarrow \mathtt{nat(x)} = \mathtt{x} \\ \mathtt{nat(0)} \quad\;\; \to \mathtt{0} \\ \mathtt{nat(s(x))} \to \mathtt{s(nat(x))} \end{array}$$

Then,

$$\begin{array}{l} T^{inn}_{\mathcal{R}} \uparrow 0 = \emptyset \\ T^{inn}_{\mathcal{R}} \uparrow 1 = \{\mathtt{0} = \mathtt{0},\ \mathtt{s(x)} = \mathtt{s(x)},\ \mathtt{nat(x)} = \mathtt{nat(x)},\ \mathtt{pre(x)} = \mathtt{pre(x)}\} \\ T^{inn}_{\mathcal{R}} \uparrow 2 = T^{inn}_{\mathcal{R}} \uparrow 1 \cup \{\mathtt{nat(0)} = \mathtt{0},\ \mathtt{nat(s(x))} = \mathtt{s(nat(x))}\} \\ T^{inn}_{\mathcal{R}} \uparrow 3 = T^{inn}_{\mathcal{R}} \uparrow 2 \cup \{\mathtt{nat(s(0))} = \mathtt{s(0)},\ \mathtt{nat(s^2(x))} = \mathtt{s^2(nat(x))}, \mathtt{pre(s(0))} = \mathtt{0}\} \\ T^{inn}_{\mathcal{R}} \uparrow 4 = T^{inn}_{\mathcal{R}} \uparrow 3 \cup \{\mathtt{nat(s^2(0))} = \mathtt{s^2(0)},\ \mathtt{nat(s^3(x))} = \mathtt{s^3(nat(x))}, \mathtt{pre(s^2(0))} = \mathtt{s(0)}\} \\ \qquad\vdots \\ T^{inn}_{\mathcal{R}} \uparrow \omega = \{\mathtt{0} = \mathtt{0},\ \mathtt{s(x)} = \mathtt{s(x)},\ \mathtt{nat(x)} = \mathtt{nat(x)},\ \mathtt{pre(x)} = \mathtt{pre(x)}, \mathtt{nat(0)} = \mathtt{0}, \mathtt{nat(s(0))} = \mathtt{s(0)}, \\ \qquad\quad \mathtt{nat(s^2(0))} = \mathtt{s^2(0)}, \ldots, \mathtt{nat(s^n(0))} = \mathtt{s^n(0)}, \ldots,\ \mathtt{nat(s(x))} = \mathtt{s(nat(x))}, \ldots, \\ \qquad\quad \mathtt{nat(s^n(x))} = \mathtt{s^n(nat(x))}, \ldots, \mathtt{pre(s(0))} = \mathtt{0}, \mathtt{pre(s^2(0))} = \mathtt{s(0)}, \ldots, \\ \qquad\quad \mathtt{pre(s^n(0))} = \mathtt{s^{n-1}(0)}, \ldots\} = \mathcal{F}_{\mathtt{inn}}(\mathcal{R}) \end{array}$$

Example 12. Let $\varphi = out$, and consider the non-terminating[5] program R defining the `first` element of a list, together with the list `from(x)` of natural numbers starting from `x`:

$$\begin{array}{l} \mathtt{from(x)} \quad\;\; \to [\mathtt{x}|\mathtt{from(s(x))}] \\ \mathtt{first([x|y])} \to \mathtt{x} \end{array}$$

Then[6]:

$$\begin{array}{l} T^{out}_{\mathcal{R}} \uparrow 0 = \emptyset \\ T^{out}_{\mathcal{R}} \uparrow 1 = \{\mathtt{s(x)} \approx \mathtt{s(x)},\ [\,] \approx [\,],\ [\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}|\mathtt{y}],\ \mathtt{from(x)} = \mathtt{from(x)},\ \mathtt{first(x)} = \mathtt{first(x)}\} \\ T^{out}_{\mathcal{R}} \uparrow 2 = T^{out}_{\mathcal{R}} \uparrow 1 \cup \{\mathtt{first([x|y])} = \mathtt{x},\ \mathtt{first(x)} = \bot,\ \mathtt{from(x)} = \bot, \mathtt{from(x)} = [\mathtt{x}|\mathtt{from(s(x))}]\} \\ \qquad\vdots \\ T^{out}_{\mathcal{R}} \uparrow \omega = \{\mathtt{s(x)} \approx \mathtt{s(x)}, [\,] \approx [\,], [\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}|\mathtt{y}], \mathtt{first(x)} = \mathtt{first(x)}, \mathtt{first(x)} = \bot, \mathtt{first([x|y])} = \mathtt{x}, \\ \qquad\quad \mathtt{from(x)} = \mathtt{from(x)},\ \mathtt{from(x)} = \bot,\ \mathtt{from(x)} = [\mathtt{x}|\mathtt{from(s(x))}],\ \mathtt{from(x)} = [\mathtt{x}|\bot],\ \ldots, \\ \qquad\quad \mathtt{from(x)} = [\mathtt{x}|[\mathtt{s(x)}|\ldots[\mathtt{s^n(x)}|\mathtt{from(s^{n+1}(x))}]]],\ \mathtt{from(x)} = [\mathtt{x}|[\mathtt{s(x)}|\ldots[\mathtt{s^n(x)}|\bot]]], \ldots\} \end{array}$$

According to Definition 10, the fixpoint semantics is[7]:

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

By disregarding the equations that denote partial computations, we obtain an intermediate semantics that can be thought of as an evaluation semantics for functional programs.

Definition 13 (*Computed answers fixpoint semantics*)**.** We let $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ denote the set $\{e \in lfp(T^{\varphi}_{\mathcal{R}}) |$ the right-hand side of e does not contain any defined function symbol $f/n \in \mathcal{F}\}$.

The semantics $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ is called *computed answers fixpoint semantics* because it provides a declarative characterization of the narrowing computations, where the answers are "computed in the denotation" by syntactic unification. The following result formalizes the precise relationship between the answer substitutions computed by narrowing w.r.t. φ with the $\bot$-free substitutions that can be "extracted" from the *computed answers (fixpoint) semantics* by syntactic unification.

Definition 14 (*Closed goal by a set of equations*)**.** Let $\mathcal{R} \in \mathbb{R}_{\varphi}$ and g be a (non-trivial) goal for φ. Let S be a set of equations. We say that g is closed by S (with substitution θ) iff there exists $g' \equiv e_1, \ldots, e_n \subseteq S$ such that $\theta = mgu(flat_{\varphi}(g), g')_{|Var(g)}$.

Note that, in the case when θ is a variable renaming, then $g \in S$.

---

[5] When considering the strict equality $\approx$ instead of the non-strict equality $=$, the completeness results for outermost narrowing in [60,61] (when we are only interested in the computation of finite and total values of expressions) generalize to nonterminating rules with little effort, see [72].

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

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

Theorem 15 (*Strong soundness and completeness*). *Let $\mathcal{R} \in \mathbb{R}_\varphi$ and g be a (non-trivial) goal for φ. Then, θ is a computed answer for g in $\mathcal{R}$ w.r.t. $\leadsto_\varphi$ iff g is closed by $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ with substitution θ.*

According to Theorem 15, $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ 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.

Example 16. Consider again the nonterminating program of Example 12. Using Definition 13, the computed answers fixpoint semantics is given by

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

with $n \in \omega$. Let us now show how the computed answers for a given goal can be distilled from this semantics by unification. 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 that 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 $\mathcal{F}^{ca}_{out}(\mathcal{R})$.

Note that, in our denotation, the proper semantic meaning of an equation $l = r$ is equality only in the case that l and r are total, i.e., without any occurrence of $\bot$. Roughly speaking, $\bot$ is used in our methodology as an artifact to allow any equation $g(x_1, \ldots, x_n) = x$ to "succeed" (when it is executed in the denotation). This is achieved by simply unifying it with the extra ("fake") equation $g(x_1, \ldots, x_n) = \bot$. This ensures that every (non-strict) equation with a pure variable in its right-hand side is solvable, which is necessary for completeness. For instance, consider the following example in [75]. Let $f(t_1, \ldots, t_m)$ be a term occurring in a rule body or a goal, and assume that f is not strict on the ith argument. By definition, if t_i is a term $g(x_1, \ldots, x_n)$, then the flattening introduces an equation $g(x_1, \ldots, x_n) = x$ in the goal (and replaces t_i by x in $f(t_1, \ldots, t_m)$). We must allow this equation $g(x_1, \ldots, x_n) = x$ to succeed with an undefined value of x, whenever the value is not required in other equations since it represents the ith argument of f. Roughly speaking, solving this equation by means of the rule $g(x_1, \ldots, x_n) = \bot$ has the effect to "undo" the flattening, which would otherwise force the evaluation of the call $g(x_1, \ldots, x_n)$, even though the evaluation was not demanded by f. Moreover, note that such a call $g(x_1, \ldots, x_n)$ could fail (e.g. if g is undefined), which would be certainly undesired.

In the following, we show the relation between the semantics $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ and a novel operational "computed answer" semantics $\mathcal{O}^{ca}_\varphi(\mathcal{R})$ that 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 $\mathcal{R}$ w.r.t. narrowing strategy φ is defined in the style of [18,38] by considering the answers computed by narrowing for "most general calls".

Definition 17 (*Success set semantics*). Let $\mathcal{R}$ be a program in $\mathbb{R}_\varphi$. Then,

$$\mathcal{O}^{ca}_\varphi(\mathcal{R}) = \Im^\varphi_\mathcal{R} \cup \{(f(x_1, \ldots, x_n) = x_{n+1})\theta \mid (f(x_1, \ldots, x_n) =_\varphi x_{n+1}) \overset{\theta}{\leadsto}{}^*_\varphi \top \text{ where } f/n \in \mathcal{F}, \text{ and } x_1, \ldots, x_{n+1} \text{ are distinct variables}\}.$$

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

Definition 18. Let S be a set of equations and Σ be the considered signature. We define

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

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

Theorem 19. *The following relation holds:*

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

Theorem 19 implies that, in the case when $\varphi = inn$, $\mathcal{O}^{ca}_\varphi(\mathcal{R}) = \mathcal{F}^{ca}_\varphi(\mathcal{R})$, whereas they only differ in the denotation of the non-terminating computations in the case when $\varphi = out$.

Example 20. Consider again the program of Example 16. According to Definition 18, we have that

$$partial(\mathcal{F}_{out}(\mathcal{R})) = \{\mathtt{from(x)} = \mathtt{from(x)},\ \mathtt{from(x)} = \bot,\ \mathtt{from(x)} = \mathtt{[x|from(s(x))]}, \ldots, \mathtt{from(x)} = \mathtt{[x|[s(x)|}\ldots\mathtt{[s^n(x)|from(s^{n+1}(x))]]]},\ \mathtt{from(x)} = \mathtt{[x|[s(x)|}\ldots\mathtt{[s^n(x)|}\bot\mathtt{]]]}, \ldots,\ \mathtt{first(x)} = \mathtt{first(x)},\ \mathtt{first(x)} = \bot\}.$$

Now, by Theorem 19, the computed answer semantics is as follows:

$$\mathcal{O}^{ca}_{out}(\mathcal{R}) = \{[\,] \approx [\,],\ \mathtt{[x|y]} \approx \mathtt{[x|y]},\ \mathtt{s(x)} \approx \mathtt{s(x)},\ \mathtt{first([x|y])} = \mathtt{x}\}.$$

Example 21. Let us consider the program $\mathcal{R} = \{\mathtt{g(x)} \to \mathtt{0},\ \mathtt{f(0)} \to \mathtt{0},\ \mathtt{f(s(x))} \to \mathtt{f(x)}\}$. According to Definition 10,

$$\mathcal{F}_{inn}(\mathcal{R}) = \{\mathtt{0} = \mathtt{0},\ \mathtt{s(x)} = \mathtt{s(x)},\ \mathtt{g(x)} = \mathtt{0},\ \mathtt{f(0)} = \mathtt{0},\ \mathtt{f(s(x))} = \mathtt{f(x)}, \ldots,\ \mathtt{f(s^n(x))} = \mathtt{f(x)}, \ldots,\ \mathtt{f(s(0))} = \mathtt{0}, \ldots, \mathtt{f(s^n(0))} = \mathtt{0}, \ldots\}.$$

From Definition 18,

$$\mathit{partial}(\mathcal{F}_{inn}(\mathcal{R})) = \{\mathtt{f(s(x))} = \mathtt{f(x)}, \ldots, \mathtt{f(s^n(x))} = \mathtt{f(x)}, \ldots\}.$$

Now, by Theorem 19,

$$\mathcal{O}^{ca}_{inn}(\mathcal{R}) = \mathcal{F}^{ca}_{inn}(\mathcal{R}) = \{\mathtt{0} = \mathtt{0},\ \mathtt{s(x)} = \mathtt{s(x)},\ \mathtt{g(x)} = \mathtt{0},\ \mathtt{f(0)} = \mathtt{0},\ \mathtt{f(s(0))} = \mathtt{0}, \ldots, \mathtt{f(s^n(0))} = \mathtt{0}, \ldots\}$$

In the following section, we use the (greater) fixpoint semantics $\mathcal{F}_\varphi(\mathcal{R})$ to define the diagnosis methodology, whereas the smaller, operational semantics $\mathcal{O}^{ca}_\varphi(\mathcal{R})$ is used to formalize the notions of correctness and completeness of a program w.r.t. a given specification.

4. Declarative diagnosis of functional logic programs

The idea behind declarative error diagnosis is to collect information about what the program is intended to do and compare this with what it actually does. Starting from these premises, a diagnoser can find errors. The information needed can be found in many different ways. It can be built by asking the user (as an oracle), or by means of a formal specification (or an older, correct, version of the program), or some combination of both.

Declarative debugging as defined in [117,68] is concerned with model-theoretic properties (the least Herbrand model in [117] and the set of atomic logical consequences in [68]). In his seminal work [95], Lloyd extended Shapiro's algorithmic debugging [117] in order to deal with logic programs with negation and non-standard computation rules, but he considers only model-theoretic properties. In [55], Comini, Levi and Vitiello extended the definitions given in [117,95,68] to diagnosis w.r.t. computed answers in order to provide more precise diagnoses. In the following, we extend the diagnosis framework of [55] in order to deal with functional logic programs.

As operational semantics, we consider the success set semantics. Since we consider two different semantics for the program $\mathcal{R}$, operational $\mathcal{O}^{ca}_\varphi(\mathcal{R})$ and fixpoint $\mathcal{F}_\varphi(\mathcal{R})$, in the sequel, we also distinguish between two different denotations (V-Herbrand interpretations) representing the intended meaning of the program: $\mathcal{I}_{ca}$ and $\mathcal{I}_{\mathcal{F}}$. Both $\mathcal{I}_{ca}$ and $\mathcal{I}_{\mathcal{F}}$ consist of standard equations as well as strict equations (in the case of the outer strategy). The symbol $\bot$ never occurs in $\mathcal{I}_{ca}$. We also note that the equality symbol $=$ in the denotation does not have the mathematical meaning of equality, in the sense that equational reasoning does not apply to equations containing the symbol $\bot$: otherwise, one would infer the equation $[x|\bot] = [x|[s(x)|\bot]]$ from the equations $\mathit{from}(x) = [x|\bot]$ and $\mathit{from}(x) = [x|[s(x)|\bot]]$, which is false under the semantics discussed in this paper. In our framework, the only meaning of the $=$ symbol is given, by extension, in the denotation itself: all equations in the denotation, and only those, hold. Standard equality properties such as symmetry of the left-hand side and the right-hand side of equations do not generally hold. This gives the non-strict equality the meaning of a "reducibility predicate" similar to [119].

While $\mathcal{I}_{ca}$ is the reference semantics from a programmer perspective, $\mathcal{I}_{\mathcal{F}}$ is suitable for the diagnosis [42], as we describe in the following.

Definition 22 (*Correctness and completeness w.r.t. reference semantics*)**.** Let $\mathcal{I}_{ca}$ be the intended success set semantics for $\mathcal{R}$.

(1) $\mathcal{R}$ is partially correct w.r.t. $\mathcal{I}_{ca}$, if $\mathcal{O}^{ca}_\varphi(\mathcal{R}) \subseteq \mathcal{I}_{ca}$.
(2) $\mathcal{R}$ is complete w.r.t. $\mathcal{I}_{ca}$, if $\mathcal{I}_{ca} \subseteq \mathcal{O}^{ca}_\varphi(\mathcal{R})$.
(3) $\mathcal{R}$ is totally correct w.r.t. $\mathcal{I}_{ca}$, if $\mathcal{O}^{ca}_\varphi(\mathcal{R}) = \mathcal{I}_{ca}$.

If a program contains errors, these are signalled by corresponding *symptoms*. The "intended success set semantics" allows us to establish the validity of an atomic equation by a simple "membership" test, in the style of the s-semantics [38,64].

Definition 23 (*Incorrectness and incompleteness symptoms*)**.** Let $\mathcal{I}_{ca}$ be 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" (partial) computations, and enjoys the semantic properties of the denotation formalized in Definition 10, that is, $\mathcal{I}_{\mathcal{F}}$ should correspond to the fixpoint semantics of the correct program and $\mathcal{I}_{ca} = \mathcal{I}_{\mathcal{F}} - \mathit{partial}(\mathcal{I}_{\mathcal{F}})$. Obviously, for a particular correct program $\mathcal{R}$, $\mathcal{F}^{ca}_\varphi(\mathcal{R}) \subseteq \mathcal{I}_{ca}$ and $\mathcal{F}_\varphi(\mathcal{R}) \subseteq \mathcal{I}_{\mathcal{F}}$. Nevertheless, in a practical system, these descriptions would not be provided by the user manually, but an approximation is automatically inferred from a finite set of input equations, e.g. a possibly inefficient (correct) version of the program, or a (executable) specification. The debugging of programs via specifications is an important topic in automated program development, where the specification is not only seen as the starting point for the subsequent program development, but also as the criterion for judging the correctness of the software system.

In case of errors, in order to determine the faulty rules, the following definitions are helpful.

Definition 24 (*Incorrect rule*)**.** Let $\mathcal{I}_{\mathcal{F}}$ be the intended fixpoint semantics for $\mathcal{R}$. If there exists an equation $e \in T^{\varphi}_{\{r\}}(\mathcal{I}_{\mathcal{F}})$ s.t. e is not closed by $\mathcal{I}_{\mathcal{F}}$, then the rule $r \in \mathcal{R}$ is incorrect on e.

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

Definition 25 (*Uncovered equation*)**.** Let $\mathcal{I}_{\mathcal{F}}$ be the intended fixpoint semantics for $\mathcal{R}$. An equation e is *uncovered* in $\mathcal{R}$ if $e \in \mathcal{I}_{\mathcal{F}}$ and e is not closed by $T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$.

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

Proposition 26. *If there are no incorrect rules in $\mathcal{R}$ w.r.t. the intended fixpoint semantics $\mathcal{I}_{\mathcal{F}}$, then $\mathcal{R}$ is partially correct w.r.t. the intended success set semantics $\mathcal{I}_{ca}$.*

Assume that $\mathcal{I}_{\mathcal{F}}$ is finite.[8] Proposition 26 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 27. Let $\varphi = out$. Consider program $\mathcal{R} = \{\mathtt{f(x)} \to \mathtt{a} \Leftarrow \mathtt{f(x)} \approx \mathtt{a}\}$ and $\mathcal{I}_{\mathcal{F}} = \{\mathtt{a} \approx \mathtt{a},\ \mathtt{f(x)} = \mathtt{f(x)},\ \mathtt{f(x)} = \bot,\ \mathtt{f(x)} = \mathtt{a}\}$. Then, $\mathcal{I}_{ca} = \{\mathtt{a} \approx \mathtt{a}, \mathtt{f(x)} = \mathtt{a}\}$ whereas $\mathcal{O}^{ca}_{out}(\mathcal{R}) = \{\mathtt{a} \approx \mathtt{a}\} \not\supseteq \mathcal{I}_{\mathtt{ca}}$; hence, $\mathcal{R}$ is not complete.

Now, let us show that there is no incompleteness symptom. First, let us compute $flat_{out}(f(x) \approx a) \equiv \{f(x) = y,\ y \approx a\}$.

Since $\mathcal{R}^{out}_{++} = \{\mathtt{f(x)} \to \mathtt{a} \Leftarrow \mathtt{f(x)} \approx \mathtt{a},\ \mathtt{f(x)} \to \bot,\ \mathtt{a} \approx \mathtt{a} \to \mathtt{true},\ \mathtt{x} \approx \mathtt{y} \to \bot\}$, then $T^{out}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}}) = \{\mathtt{a} \approx \mathtt{a},\ \mathtt{f(x)} = \mathtt{f(x)},\ (\mathtt{x} \approx \mathtt{y}) = (\mathtt{x} \approx \mathtt{y}),\ \mathtt{f(x)} = \bot,\ \mathtt{f(x)} = \mathtt{a},\ (\mathtt{x} \approx \mathtt{y}) = \bot\}$. Therefore, $\mathcal{I}_{\mathcal{F}} \subseteq T^{out}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$ and there are no uncovered equations.

The problem is related to the existence of several fixpoints for the $T_{\mathcal{R}}$ operator. See [55] for details.

It is worth noting that checking the conditions of Definitions 24 and 25 requires just one application of $T^{\varphi}_{\mathcal{R}}$ to $\mathcal{I}_F$, while the standard detection based on symptoms [117] would require either an external oracle or the construction of the semantics, and therefore a fixpoint computation.

5. Abstract semantics

The theory of abstract interpretation [56] provides a formal framework for developing advanced data-flow analysis tools. Abstract interpretation formalizes the idea of 'approximate computation' in which computation is performed with descriptions of data rather than with the data themselves. The semantics operators are then replaced by abstract operators that are shown to 'safely' approximate the standard ones. In this section, starting from the fixpoint semantics developed in Section 3, we formalize an abstract semantics that approximates the behavior of the program and is adequate for modular data-flow analysis, such as the analysis of unsatisfiability of equation sets or any analysis that is based on the program success set. We assume the framework of abstract interpretation for analysis of equational unsatisfiability as defined in [15]. In [15], we only dealt with the standard equality, whereas in this paper we consider two different equalities, and thus we slightly generalize the results in order to apply them in the case of the outer strategy, too.

We recall the basic definitions of the abstract domains and the associated abstract operators (see [8,15,18] for details). Then, we describe the 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. [15]). In the following, we denote the abstract analog of a concrete object O by $O^{\sharp}$.

The abstract methodology in this section, which was first presented in [9], generalizes the results in [8] by making them parametric w.r.t. $\varphi \in \{inn, out\}$.

5.1. Abstract programs and operators

Definition 28 (*Description*)**.** A *description* is the association of an *abstract domain* $(D, \leq)$ (a poset) with a *concrete domain* $(E, \leq)$ (a poset). When $E = Eqn$ or $E = Sub$, the description is called an *equation description* or a *substitution description*, respectively. 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 $e \in \gamma(d)$. The approximation relation can be lifted to relations and cross-products as usual [15].

[8] In our methodology, finiteness is achieved by considering a finite approximation of $\mathcal{I}_{\mathcal{F}}$ that is computed by abstract interpretation, as described in Section 5.

Abstract substitutions are introduced for the purpose of describing the computed answer substitutions for a given goal. Abstract equations and abstract substitutions correspond, in our approach, to abstract program denotations and abstract observable properties, respectively. The domains for equations and substitutions are based on a notion of the abstract Herbrand universe $\mathcal{H}_V^\sharp$, which introduces an irreducible symbol $\sharp$ (see [15,18]).

Definition 29 (*Abstract Herbrand universe*)**.** Let $\sharp$ be an irreducible fresh symbol, where $\sharp \notin \Sigma \cup \{\bot\}$. Let $\mathcal{H}_V^\sharp = (\tau(\Sigma \cup \{\bot\} \cup V \cup \{\sharp\}), \preceq)$ be the domain of terms over the signature augmented by $\sharp$, where the partial order $\preceq$ is defined as follows:

(a) $\forall t \in \mathcal{H}_V^\sharp, \sharp \preceq t$ and $t \preceq t$ and

(b) $\forall s_1, \ldots, s_n, s'_1, \ldots, s'_n \in \mathcal{H}_V^\sharp, \forall f/n \in \Sigma, s'_1 \preceq s_1 \wedge \cdots \wedge s'_n \preceq s_n \Rightarrow f(s'_1, \ldots, s'_n) \preceq f(s_1, \ldots, s_n)$.

This order can be extended to (unquantified) equations: $s' = t' \preceq s = t$ iff $s' \preceq s$ and $t' \preceq t$ and to (possibly infinite) sets of equations S, S':

(1) $S' \preceq S$ iff $\forall e' \in S', \exists e \in S$ such that $e' \preceq e$. Note that $S' \preceq \{true\} \Rightarrow S' \equiv \{true\}$.
(2) $S' \sqsubseteq S$ iff $(S' \preceq S)$ and $(S \preceq S'$ implies $S' \subseteq S)$.

Intuitively, $S' \sqsubseteq S$ means that either S' contains less information than S, or if they have the same information, then S' expresses it using fewer elements.

Roughly speaking, the special symbol $\sharp$ introduced in the abstract domains represents any concrete term. From the viewpoint of logic, $\sharp$ stands for an existentially quantified variable [15,97,99]. Thus, from a programming viewpoint, the behavior of the symbol $\sharp$ resembles that of an "anonymous" variable in Prolog. Define $[\![S]\!] = S'$, where the n-tuple of occurrences of $\sharp$ in S is replaced by an n-tuple of existentially quantified fresh variables in S'.

Definition 30 (*Abstract substitution*)**.** An abstract substitution is a set of the form $\{x_1/t_1, \ldots, x_n/t_n\}$ where, for each $i = 1, \ldots, n$, x_i is a distinct variable in V not occurring in any of the terms $t_1, \ldots, t_n$ and $t_i \in \tau(\Sigma \cup V \cup \{\sharp\})$. The ordering on abstract substitutions is given by logical implication: let $\theta, \kappa \in Sub^\sharp$, $\kappa \preceq \theta$ iff $[\![\widehat{\theta}]\!] \Rightarrow [\![\widehat{\kappa}]\!]$.

The descriptions for terms, substitutions and equations are as follows.

Definition 31. Let $\mathcal{H}_V = (\tau(\Sigma \cup \{\bot\} \cup V), \leq)$ and $\mathcal{H}_V^\sharp = (\tau(\Sigma \cup \{\bot\} \cup V \cup \{\sharp\}), \preceq)$. The *term description* is $\langle \mathcal{H}_V^\sharp, \gamma, \mathcal{H}_V \rangle$ where $\gamma : \mathcal{H}_V^\sharp \to 2^{\mathcal{H}_V}$ is defined by $\gamma(t') = \{t \in \mathcal{H}_V | t' \preceq t\}$.

Here, we would like to emphasize the differences between the two symbols $\sharp$ and $\bot$ which are related, in our framework, with the "lack of information". Let $\dot{\preceq}$ be the inverse of $\preceq$, i.e., $S \dot{\preceq} S'$ iff $S' \preceq S$. In terms of abstract interpretation, the symbol $\sharp$ corresponds to the abstract top element $\top^\sharp$ of the poset $(\mathcal{H}_V^\sharp, \dot{\preceq})$–i.e., the one with the biggest concretization—with $\gamma(\top^\sharp) = \tau(\Sigma \cup \{\bot\} \cup V)$. The abstract bottom element in our framework is $\bot^\sharp$, with $\gamma(\bot^\sharp) = \{\bot\}$, which will simply be denoted by $\bot$. The poset $(\mathcal{H}_V^\sharp, \dot{\preceq})$ can be extended to a complete lattice in the usual way, by considering the standard set union as least upper bound, and the intersection of the instances (over $\Sigma \cup \{\bot\}$) of the concretization as the greatest lower bound.

In the rest of the paper, *Eqn* denotes the set of (possibly infinite,[9] existentially quantified) equation sets (in the case when $\varphi = out$, we assume that the equations can contain the equality symbols $=$ and $\approx$) over $\tau(\Sigma \cup \{\bot\} \cup V)$ and $Eqn^\sharp$ is the corresponding set of finite sets of equations over $\tau(\Sigma \cup \{\bot\} \cup V \cup \{\sharp\})$.

Definition 32. The *equation description* is $\langle (Eqn^\sharp, \sqsubseteq), \gamma, (Eqn, \leq) \rangle$, where $\gamma : Eqn^\sharp \to 2^{Eqn}$ is defined by $\gamma(g') = \{g \in Eqn | g' \sqsubseteq g$ and g is unquantified$\}$.

Let *Sub* be the set of substitutions over $\tau(\Sigma \cup \{\bot\} \cup V)$ and $Sub^\sharp$ be the set of substitutions over $\tau(\Sigma \cup \{\bot\} \cup V \cup \{\sharp\})$. The *substitution description* $\langle (Sub^\sharp, \preceq), \gamma, (Sub, \leq) \rangle$, where $\gamma : Sub^\sharp \to 2^{Sub}$ is defined by $\gamma(\kappa) = \{\theta \in Sub | \kappa \preceq \theta\}$.

In order to perform computations over the abstract domains, we have to define the notion of *abstract unification*. The abstract most general unifier for our method is very simple and, roughly speaking, it boils down to computing a solved form of an equation set with (possibly) existentially quantified variables. We define the abstract most general unifier for an equation set $S' \in Eqn^\sharp$ as follows. First, replace all occurrences of $\sharp$ in S' by existentially quantified fresh variables. Then, take a solved form of the resulting quantified equation set and finally replace the existentially quantified variables again by $\sharp$.

Definition 33 (*Abstract most general unifier*)**.** Let $\exists y_1 \ldots y_n . S = solve([\![S']\!])$, where the equations in S are unquantified, and $\kappa = \{y_1/\sharp, \ldots, y_n/\sharp\}$. Then, $\widehat{mgu}^\sharp(S') = S\kappa$.

[9] We handle infinite equation sets only when we deal with abstract Herbrand interpretations.

The fact that $\forall\theta \in unif([\![S]\!]).\ mgu^{\sharp}(S) \preceq \theta$ justifies our use of 'most general'. The safety of the abstract unification algorithm has been proven in [15].

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, i.e. a graph of functional dependencies built from $\mathcal{R}$ which helps to recognize the narrowing derivations that definitely terminate.

Definition 34 (*Loop-check*)**.** Given a program $\mathcal{R}$, a loop-check for $\mathcal{R}$ is a pair $(\mathcal{G}_{\mathcal{R}}, \circ)$ where $\mathcal{G}_{\mathcal{R}}$ is a finite graph of terms and the set-valued function $\circ$: $\tau(\Sigma \cup V) \mapsto 2^{\tau(\Sigma\cup V)}$ assigns a set of nodes $\overset{\circ}{t}$ in $\mathcal{G}_{\mathcal{R}}$ to the term t such that, for any infinite sequence: $g_0 \overset{\theta_0}{\leadsto}_{\varphi} g_1 \overset{\theta_1}{\leadsto}_{\varphi} \ldots$ in $\mathcal{R}$, there exists $i \geq 0, u \in \overline{O}(g_i)$, and $t_i \in \tau(\Sigma \cup V)$, such that $t_i \in \overset{\circ}{g}_{i|u}$, and $\langle t_i, t_i\rangle \in \mathcal{G}^{+}_{\mathcal{R}}$, where $\mathcal{G}^{+}_{\mathcal{R}}$ is the transitive closure of $\mathcal{G}_{\mathcal{R}}$. We refer to $\langle t_i, t_i\rangle$ as a 'cycle' of $\mathcal{G}_{\mathcal{R}}$.

A loop-check can be thought of as a sort of 'oracle' whose usefulness in proving the termination of narrowing derivations is stated by the fact that if there is no cycle in $\mathcal{G}_{\mathcal{R}}$, then narrowing derivations for $\mathcal{R}$ terminate [15]. By choosing appropriate loop-checks, it is possible to tune the precision of the abstraction.

The following definition introduces a simple form of loop-check which can be seen as a sort of *estimated narrowing dependency graph* [11] $\mathcal{DG}_{\mathcal{R}}$ for the CTRS $\mathcal{R}$ which considers functional dependencies directly on $\mathcal{R}$ instead of first transforming it into an unconditional TRS, as is done in the analysis of conditional *rewriting* termination, e.g. [73,116]. A simpler version of this loop-check for the basic conditional narrowing strategy [90] was proposed in [14,15], and subsequently refined in [18].

We need some auxiliary definitions. We denote by $t \overset{?}{=} s$ the fact that t and (a fresh variant) of s are unifiable. Given a term t, $O_{\mathcal{F}}(t)$ denotes the set of positions of t that address a function-rooted subterm of t, and $\lfloor t \rfloor$ denotes the term which is obtained by inductively replacing by a fresh variable the subterms of t which are not constructor-rooted, i.e.

$$\lfloor t \rfloor = \begin{cases} c(\lfloor t_1\rfloor, \ldots, \lfloor t_k\rfloor) & \text{if } t = c(t_1, \ldots, t_k) \text{ and } c \in \mathcal{C} \\ y & \text{otherwise, where } y \text{ is a fresh variable.} \end{cases}$$

Roughly speaking, $\lfloor t \rfloor$ replaces every outermost, non-constructor-rooted subterm of t by a fresh variable, while keeping the constructor spine above those subterms. For instance, for $f \in \mathcal{F}$, and $s, c \in \mathcal{C}$, $\lfloor c(f(x), s(f(x)))\rfloor = c(z_1, s(z_2))$. This function, first defined in [14], was subsequently split into two functions respectively named CAP (removal of functional nestings) and REN (linearization by variable renaming) in the DP approach [26].

Definition 35 (*Graph of functional dependencies*)**.** Let $\mathcal{R}$ be a CTRS. The following transformation defines a directed graph $\mathcal{DG}_{\mathcal{R}}$ of functional dependencies induced by $\mathcal{R}$. We define $\bar{t} = f(\lfloor t_1\rfloor, \ldots, \lfloor t_n\rfloor)$ if $t = f(t_1, \ldots, t_n)$, and $f \in \mathcal{F}$. In order to build $\mathcal{DG}_{\mathcal{R}}$, the algorithm starts with $\langle \mathcal{R}, \emptyset\rangle$ and applies the inference rules (1) and (2) as long as they add new arrows. The symbol $\cup$ stands for set union (modulo renaming), i.e., graph arrows are considered equivalent up to renaming:

$$(1)\quad \frac{r = (\lambda \to \rho \Leftarrow C) \ll \mathcal{R}}{\langle \mathcal{R}, \mathcal{DG}_{\mathcal{R}}\rangle \longmapsto \langle \mathcal{R} - \{r\}, \mathcal{DG}_{\mathcal{R}} \cup \{\lambda \overset{\mathcal{R}}{\to} \bar{t} \mid (t = \rho_{|u}, u \in O_{\mathcal{F}}(\rho)) \text{ or } (t = C_{|u}, u \in O_{\mathcal{F}}(C)) \text{ or } (t = \lambda_{|u}, u \in (O_{\mathcal{F}}(\lambda) - \{\Lambda\}))\}\rangle}$$

$$(2)\quad \frac{(\lambda \overset{\mathcal{R}}{\to} r) \in \mathcal{DG}_{\mathcal{R}} \ \wedge\ (\lambda' \overset{\mathcal{R}}{\to} r') \in \mathcal{DG}_{\mathcal{R}} \ \wedge\ r \overset{?}{=} \lambda'}{\langle \mathcal{R}, \mathcal{DG}_{\mathcal{R}}\rangle \longmapsto \langle \mathcal{R}, \mathcal{DG}_{\mathcal{R}} \cup \{r \overset{u}{\to} \lambda'\}\rangle}.$$

Termination of this calculus is ensured since the number of terms occurring in the rules in $\mathcal{R}$ is finite. Roughly speaking, in Definition 35, for each rule $(\lambda \to \rho \Leftarrow C)$ in $\mathcal{R}$ and for each defined function call $f(t_1, \ldots, t_n)$ occurring in ρ, in C, or as a proper subterm[10] of λ, rule (1) adds an arrow $\lambda \overset{\mathcal{R}}{\to} f(\lfloor t_1\rfloor, \ldots, \lfloor t_n\rfloor)$ to $\mathcal{DG}_{\mathcal{R}}$. Rule (2) adds an arrow $r \overset{u}{\to} \lambda'$ between the right-hand side r of an arrow $\lambda \overset{\mathcal{R}}{\to} r$ in $\mathcal{DG}_{\mathcal{R}}$ and the left-hand side λ' of each arrow $\lambda' \overset{\mathcal{R}}{\to} r'$ with which r unifies (note that the arrows $\lambda \overset{\mathcal{R}}{\to} r$ and $\lambda' \overset{\mathcal{R}}{\to} r'$ can also be the same). A path in the graph contains arrows $\overset{\mathcal{R}}{\to}$ and arrows $\overset{u}{\to}$.

Now we define a particular instance of function $\circ$ of Definition 34 as follows: for any term t, define $\diamond t$ as the set of nodes in $\mathcal{DG}_{\mathcal{R}}$ such that for every function-rooted subterm t' of t, if $\bar{t'}$ unifies with some node λ, with $\lambda \overset{\mathcal{R}}{\to} r$ in $\mathcal{DG}_{\mathcal{R}}$, then $\lambda \in \diamond t$. By endowing $\mathcal{DG}_{\mathcal{R}}$ with $\diamond$, $(\mathcal{DG}_{\mathcal{R}}, \diamond)$ is a loop-check for $\mathcal{R}$. This follows from [18,11] because there are three basic patterns of non-terminating narrowing derivations [11]:

[10] Note that the dependencies between the left-hand side l of a rule and the non-constructor subterms of l itself are also considered. This is because in non-CB programs, these subterms can be brought into the narrowing derivation by instantiation, thus causing an *echoing* effect that may lead to non-termination [11]. For instance, in Example 36 below, an infinite narrowing derivation exists for the goal `c(fix(x),x)=0` due to the rule `fix(fix(x))` → `x`, namely: `c(fix(x),x)=0` ⇝ `c(x',fix(x'))=0` ⇝ `c(fix(x''),x'')=0` ⇝ ⋯.

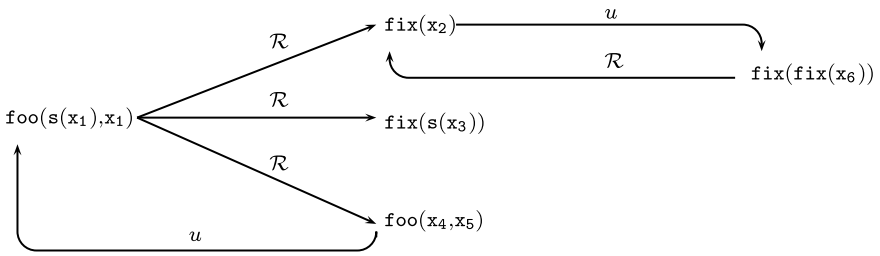

Fig. 1. Estimated graph of (narrowing) functional dependencies.

- the *top* derivations (e.g. a derivation $\underline{\texttt{last(ys)}} = 0 \leadsto (\underline{\texttt{append(zs, [y])}} = \texttt{xs}, \texttt{y} = 0) \leadsto (\texttt{[x'|}\underline{\texttt{append(xs', [y])}}\texttt{]} = \texttt{xs}, \texttt{y} = 0) \leadsto \cdots$ in the program of Example 3),
- the *echoing* derivations (e.g. the derivation $\texttt{c(}\underline{\texttt{fix(x)}}\texttt{,x)=0} \leadsto \texttt{c(x',}\underline{\texttt{fix(x')}}\texttt{)=0} \leadsto \texttt{c(}\underline{\texttt{fix(x'')}}\texttt{,x'')=0} \leadsto \cdots$. in the program of Example 36), and
- the *hybrid* derivations (e.g. the derivation $\texttt{c(}\underline{\texttt{f(x)}}\texttt{, x)} = 0 \leadsto \texttt{c(0,}\underline{\texttt{g(x')}}\texttt{)} = 0 \leadsto \texttt{c(0,}\underline{\texttt{g(x'')}}\texttt{)} = 0 \leadsto \cdots$ in the TRS $\{\texttt{f(g(x))} \to 0,\ \texttt{g(x)} \to \texttt{g(x)}\}$).

The top derivations are caught by $\mathcal{DG}_{\mathcal{R}}$ similarly to the non-terminating basic narrowing derivations of [15,18], while both echoing and hybrid narrowing derivations can be identified by using the non-vanilla ll-dependency pairs of [11], which correspond, in our narrowing dependency graph $\mathcal{DG}_{\mathcal{R}}$, to the arrows $\overset{\mathcal{R}}{\to}$ from a term on the left-hand side of a rule to its non-constructor subterms.

The key idea (originally from [14]) to extract in the form of (dependency) *pairs* the functional dependencies among the left-hand sides and the right-hand sides of the rules of $\mathcal{R}$, and then unifying the right-hand side of a pair with the left-hand side of another pair in order to catch infinite (rewriting) chains was later implemented, by means of the REN and CAP functions, for (unconditional) TRSs in the DP approach [26].

Example 36. Let us consider the following CTRS $\mathcal{R}$:

$$\begin{array}{lcl} \texttt{foo(0, x)} & \to & \texttt{c(x, x)} \\ \texttt{foo(s(x), x)} & \to & \texttt{fix(x)} \Leftarrow \texttt{fix(s(foo(x, x)))} = 0 \\ \texttt{fix(fix(x))} & \to & \texttt{x}. \end{array}$$

We depict the dependency graph induced by $\mathcal{R}$ in Fig. 1. There are two cycles in the graph: $\texttt{fix(x}_2\texttt{)} \to^* \texttt{fix(x}_2\texttt{)} \to^* \cdots$ and $\texttt{foo(x}_4\texttt{, x}_5\texttt{)} \to^* \texttt{foo(x}_4\texttt{, x}_5\texttt{)} \to^* \cdots$.

Let us now describe how we can abstract a program $\mathcal{R}$ in $\mathbb{R}_{\varphi}$. Roughly speaking, the program is abstracted by simplifying the right-hand side and the condition of each rule. 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[11] by replacing them by $\sharp$ (an optimization not considered in this paper could evaluate ordering constraints on the nodes of the graph in order to neglect some cycles, as is done in the analysis of rewriting termination, e.g. [73]). We use this definition in an 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 37 (*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 $sh(x, \mathcal{G}_{\mathcal{R}})$ of an expression x according to a loop-check $\mathcal{G}_{\mathcal{R}}$ is inductively defined as follows:

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

Example 38. Let us consider the following program, built up with pieces of code from previous examples:

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

11 Note that the fact that $\mathcal{R}$ is CB implies that no subterm on the left-hand side of a rule needs to be replaced by $\sharp$.

In order to abstract the recursive definitions of add and from, it suffices to consider the rough loop-check $(\mathcal{G}_{\mathcal{R}}, \circ)$ consisting of the following arrows $\{\mathtt{from}(\mathtt{x}_1) \to \mathtt{from}(\mathtt{x}_1), \mathtt{add}(\mathtt{x}_2, \mathtt{x}_3) \to \mathtt{add}(\mathtt{x}_2, \mathtt{x}_3)\}$, with function $\circ$ defined as function $\diamond$ of Definition 35 above. Then, the abstraction of the program $\mathcal{R}$ is $\mathcal{R}^\sharp$:

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

We can now formalize the abstract semantics.

5.2. Abstract fixpoint semantics

We define an abstract fixpoint semantics in terms of the least fixpoint of a continuous transformation $T_{\mathcal{R}}^{\sharp\varphi}$ 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 $T_{\mathcal{R}}^{\sharp\varphi}$. Although this abstract operator is not the best possible approximation, which could be achieved by $\alpha \circ T_{\mathcal{R}} \circ \gamma$ according to the theory of abstract interpretation, it is computed very efficiently and is the most appropriate in our abstract interpretation framework, where we do not formalize an explicit abstraction function α.

Definition 39 (*Abstract Herbrand base, abstract Herbrand interpretation*)**.** The abstract Herbrand base of equations $\mathcal{B}_V^\sharp$ is defined as the set of equations over the abstract Herbrand universe $\mathcal{H}_V^\sharp$. As in the concrete case, the equations in $\mathcal{B}_V^\sharp$ have the form $t = s$ when $\varphi = inn$, whereas equations have the form $t = s$ or $t \approx s$ whenever $\varphi = out$. An abstract Herbrand interpretation is any element of $2^{\mathcal{B}_V^\sharp}$.

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

Definition 40 (*Abstract immediate consequence operator*)**.** Let $\mathcal{R}$ be a program in $\mathbb{R}_\varphi$, $\mathcal{G}_{\mathcal{R}}$ 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 they exist. Let $\mathcal{I}$ be an abstract Herbrand interpretation. Then,

$$\begin{array}{rl} T_{\mathcal{R}}^{\sharp\varphi}(\mathcal{I}) = \Phi_{\mathcal{R}} \cup \Im_{\mathcal{R}}^{\varphi} \cup \{\, e \in \mathcal{B}_V^\sharp \mid & (\lambda \to \rho \Leftarrow C) \ll \mathcal{R}_{++}^{\sharp\varphi},\ \ l = r \in \mathcal{I},\ \ C' \subseteq \mathcal{I}^{\mathcal{C}}, \\ & mgu^\sharp(flat_\varphi(C), C') = \sigma,\ \ mgu^\sharp(\{\lambda = (\mathrm{r}_{|u})\}\sigma) = \theta, \\ & u \in \varphi(r),\ \ e = (l = r[\rho]_u)\sigma\theta\} \end{array}$$

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

Proposition 41. *The $T_{\mathcal{R}}^{\sharp\varphi}$ operator is continuous on the complete lattice of abstract Herbrand interpretations.*

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

Definition 42 (*Abstract least fixpoint Semantics*)**.** The abstract least fixpoint semantics of a program $\mathcal{R}$ is $\mathcal{F}_\varphi^\sharp(\mathcal{R}) = lfp(T_{\mathcal{R}}^{\sharp\varphi})$. Let $\mathcal{F}_\varphi^{ca\sharp}(\mathcal{R}) = \{l =_\varphi r \in \mathcal{F}_\varphi^\sharp(\mathcal{R}) \mid r \in \tau(\mathcal{C} \cup V)\}, \varphi \in \{inn, out\}$.

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

Theorem 43. *There exists a finite positive number k such that $\mathcal{F}_\varphi^\sharp(\mathcal{R}) = T_{\mathcal{R}}^{\sharp\varphi} \uparrow k, \varphi \in \{inn, out\}$.*

From a semantics viewpoint, given a program $\mathcal{R}$, the fixpoint semantics $\mathcal{F}_\varphi(\mathcal{R})$ (resp. $\mathcal{F}_\varphi^{ca}(\mathcal{R})$) is approximated by the corresponding abstract fixpoint semantics $\mathcal{F}_\varphi^\sharp(\mathcal{R})$ (resp. $\mathcal{F}_\varphi^{ca\sharp}(\mathcal{R})$). 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.

Theorem 44. *$\mathcal{F}_\varphi^\sharp(\mathcal{R}) \propto \mathcal{F}_\varphi(\mathcal{R})$, $\mathcal{F}_\varphi^{ca\sharp}(\mathcal{R}) \propto \mathcal{F}_\varphi^{ca}(\mathcal{R})$, and $\mathcal{O}_\varphi^{ca\sharp}(\mathcal{R}) \propto \mathcal{O}_\varphi^{ca}(\mathcal{R})$.*

The semantics $\mathcal{F}_\varphi^{ca\sharp}(\mathcal{R})$ 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 $\mathcal{F}_\varphi^{ca\sharp}(\mathcal{R})$.

Definition 45 (*Abstract closure of a goal*)**.** Let $\mathcal{R} \in \mathbb{R}_\varphi$ and g be a (non-trivial) goal for φ. Let S be a set of equations. We say that g is abstractly closed by S (with substitution θ) iff there exist $g' \equiv (e_1, \ldots, e_n) \subseteq S$ such that $\theta = mgu^\sharp(flat_\varphi(g), g')_{|Var(g)}$.

Theorem 46 (*Completeness*). *Let $\mathcal{R}$ be a program in $\mathbb{R}_\varphi$ and g be a (non-trivial) goal. If θ is a computed answer substitution for g in $\mathcal{R}$ w.r.t. φ, then g is abstractly closed by $\mathcal{F}_\varphi^{ca\sharp}(\mathcal{R})$ with substitution θ' and $(\theta' \preceq \theta)_{\lceil Var(g)}$.*

Example 47. Consider again the program of Example 38. Then, the corresponding abstract fixpoint semantics is the finite set:

$$\begin{array}{l}\mathcal{F}^{\sharp}_{out}(\mathcal{R}) = \\ \quad \{0 \approx 0,\ \mathtt{s(x)} \approx \mathtt{s(x)},\ [\,] \approx [\,],\ [\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}|\mathtt{y}],\ (\mathtt{x} \approx \mathtt{y}) = (\mathtt{x} \approx \mathtt{y}),\ \mathtt{double(x)} = \mathtt{double(x)},\\ \quad \mathtt{first(x)} = \mathtt{first(x)},\ \mathtt{add(x,y)} = \mathtt{add(x,y)},\ \mathtt{from(x)} = \mathtt{from(x)},\ \mathtt{and(x,y)} = \mathtt{and(x,y)}\} \ \cup \\ \\ \quad \{([\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}|\mathtt{y}]) = \bot,\ (\mathtt{x} \approx \mathtt{y}) = \bot,\ \mathtt{and(x,y)} = \bot,\ \mathtt{from(x)} = \bot,\ \mathtt{first(x)} = \bot,\ \mathtt{add(x,y)} = \bot,\\ \quad \mathtt{double(x)} = \bot,\} \ \cup \\ \\ \quad \{([\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}|\mathtt{y}]) = \sharp,\ (0 \approx 0) = \mathtt{true},\ (\mathtt{s(x)} \approx \mathtt{s(y)}) = \sharp,\ ([\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}|\mathtt{y}]) = \mathtt{and}(\sharp,\sharp),\ \mathtt{and(true,x)} = \mathtt{x},\\ \quad \mathtt{from(x)} = [\mathtt{x}|\sharp],\ \mathtt{first}([\mathtt{x}|\mathtt{y}]) = \mathtt{x},\ \mathtt{add}(0,\mathtt{x}) = \mathtt{x},\ \mathtt{double(x)} = \mathtt{add(x,x)},\ \mathtt{add(s(x),y)} = \mathtt{s}(\sharp),\\ \quad \mathtt{double}(0) = 0,\ \mathtt{double(s(x))} = \mathtt{s}(\sharp)\}\end{array}$$

which approximates the program success set:

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

Given a goal $g \equiv \mathtt{first}([\mathtt{double(x)}|\mathtt{from(x)}]) \approx \mathtt{y}$, outermost narrowing computes the infinite set of substitutions $\{\{\mathtt{x}/0,\ \mathtt{y}/0\}, \{\mathtt{x}/\mathtt{s}(0),\ \mathtt{y}/\mathtt{s}^2(0)\},\ldots,\{\mathtt{x}/\mathtt{s}^n(0),\ \mathtt{y}/\mathtt{s}^{2*n}(0)\},\ldots\}$. Now, the abstract substitutions computed by abstract unification in $\mathcal{F}^{ca\sharp}_{out}(\mathcal{R})$ of the equations of the 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 $\{\{\mathtt{x}/0,\ \mathtt{y}/0\}, \{\mathtt{x}/\mathtt{s}(\mathtt{x}_1), \mathtt{y}/\mathtt{s}(\sharp)\}, \{\mathtt{x}/\mathtt{x}', \mathtt{y}/\bot\}\}$, which approximate the computed answers of g.

Note that the two symbols $\sharp$ and $\bot$ may appear simultaneously in the semantics $\mathcal{F}^{\sharp}_{out}(\mathcal{R})$ but are handled differently, according to their interpretation. The "default value" $\bot$ is handled by the rule $f(x_1,\ldots,x_n) = \bot$ as a special data constructor (constant) symbol, as in [31]. On the other hand, the symbol $\sharp$ is handled as an anonymous variable that abstractly unifies with any term. If we handled $\bot$ as an existentially quantified variable and let it unify with every term, the resulting semantics would not correctly model computed answers due to the rules $f(x_1,\ldots,x_n) = \bot$.

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 finite sets to verify partial correctness: $\mathcal{I}^+$ which over-approximates the intended fixpoint semantics $\mathcal{I}_\mathcal{F}$ (that is, $\mathcal{I}_\mathcal{F} \in \gamma(\mathcal{I}^+)$) and $\mathcal{I}^-$ which under-approximates $\mathcal{I}_\mathcal{F}$. In particular, we restrict our interest to under-approximations which are a subset of $\mathcal{I}_\mathcal{F}$ (that is, $\mathcal{I}_\mathcal{F} \supseteq \mathcal{I}^-$). We then use these sets $\mathcal{I}^+$ and $\mathcal{I}^-$ as shown below, where the immediate consequence operator $T^{\varphi}_{\mathcal{R}}$ (w.r.t. the program $\mathcal{R}$) is applied once to $\mathcal{I}^-$ to check incorrectness w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$, and the abstract immediate consequence operator $T^{\sharp\varphi}_{\mathcal{R}}$ is applied to $\mathcal{I}^+$ to check incompleteness w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$.

Definition 48 (*Correct approximation*). We say that a pair $(\mathcal{I}^+, \mathcal{I}^-)$ of abstract Herbrand interpretations is a *correct approximation* of the interpretation $\mathcal{I} \subseteq \mathcal{B}_\mathcal{V}$, if $\mathcal{I}^-, \mathcal{I}^+$ respectively are an under-approximation and an over-approximation of $\mathcal{I}$.

Definition 49 (*Abstract correctness and completeness*). Let $(\mathcal{I}^+, \mathcal{I}^-)$ be a correct approximation of the intended semantics $\mathcal{I}_\mathcal{F}$. Then,

(1) the rule r is abstractly incorrect on e w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$ if $e \in T^{\varphi}_{\{r\}}(\mathcal{I}^-)$ and, for all $\mathcal{I} \in \gamma(\mathcal{I}^+)$, e is not closed by $\mathcal{I}$;

(2) $\mathcal{R}$ is abstractly incomplete on e w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$ if $e \in \mathcal{I}^-$ and, for all $\mathcal{I} \in \gamma(T^{\sharp\varphi}_{\mathcal{R}}(\mathcal{I}^+))$, e is not closed by $\mathcal{I}$.

Roughly speaking, Definition 49 states that, on the one hand, a rule r is abstractly incorrect w.r.t. the correct approximation $(\mathcal{I}^+, \mathcal{I}^-)$, whenever there exists an equation e obtained by applying rule r to some equation of the under-approximation $\mathcal{I}^-$ such that (the flattened version of) e does not unify with some equations in $\mathcal{I}$, for every concretization $\mathcal{I}$ of the over-approximation $\mathcal{I}^+$. On the other hand, abstract incompleteness of a program $\mathcal{R}$ is witnessed by an equation e, when e belongs to the under-approximation $\mathcal{I}^-$, and (the flattened version of) e does not unify with some equations in $\mathcal{I}$, for every concretization $\mathcal{I}$ of the abstract Herbrand interpretation obtained by applying abstract program rules to $\mathcal{I}^+$.

Following the abstract diagnosis approach for the debugging of computed answers [54], we do refer to the whole program $\mathcal{R}$ when we define our abstract incompleteness criterion of Definition 49. However, it is straightforward to particularize this criterion (together with all related notions defined in this paper) for identifying incomplete function definitions in the style of [95,44]. This can be done by just considering the root symbol f of the left-hand side of any equation e on which $\mathcal{R}$ is abstractly incomplete: if $\mathcal{R}$ is abstractly incomplete on e w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$, then f's definition (the set of rules $f(\bar{d}) \to t \Leftarrow C$ defining f in $\mathcal{R}$) can be considered to be abstractly incomplete w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$.

The following results hold. These propositions correct and also simplify the results in [8,9]. They establish the correctness of the abstract debugging.

Theorem 50. *Let $(\mathcal{I}^+, \mathcal{I}^-)$ be a correct approximation of the intended semantics $\mathcal{I}_{\mathcal{F}}$. If r is abstractly incorrect w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$ on e, then r is incorrect on e.*

Theorem 51. *Let $(\mathcal{I}^+, \mathcal{I}^-)$ be a correct approximation of the intended semantics $\mathcal{I}_{\mathcal{F}}$. If $\mathcal{R}$ is abstractly incomplete w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$ on e, then e is uncovered in $\mathcal{R}$.*

Hence, the use of γ in Definition 49 does not prevent us from having a finite debugging methodology. Below, we show how we can efficiently and safely implement the tests on a particular kind of *computed approximation* by simply checking whether the considered equation abstractly unifies with some element of $\mathcal{I}^+$ (Proposition 50) or $T^{\sharp\varphi}_{\mathcal{R}}(\mathcal{I}^+)$ (Proposition 56), and then performing an easy, finite test on the abstract *mgu*.

First, we need the following auxiliary result, which formalizes a useful relation between the elements of $\mathcal{I}$ and those in $\mathcal{I}^+$ and is the key for the implementation of our methodology.

Definition 52 (*Correct approximation*)**.** Let $(\mathcal{I}^+, \mathcal{I}^-)$ be a correct approximation of $\mathcal{I}_{\mathcal{F}}$ and let $e \in \mathcal{B}_{\mathcal{V}}$. We say that e is abstractly covered by $(\mathcal{I}^+, \mathcal{I}^-)$ if there exists $g' \subseteq \mathcal{I}^+$ s.t. $mgu^\sharp(flat_\varphi(e), g')_{|Var(e)} \neq fail$.

In our methodology, an executable specification $\mathcal{R}_{Spec}$ is given as a means to correctly provide the intended semantics. Then, given program $\mathcal{R}_{Spec}$, we formulate a method to compute suitable over- and under-approximations.

Definition 53 (*Computed approximation*)**.** Let $\mathcal{R}_{Spec}$ be a program. We define the computed approximation $(c\mathcal{I}^+, c\mathcal{I}^-)$ as follows: $c\mathcal{I}^+ = lfp(T^{\sharp\varphi}_{\mathcal{R}_{Spec}})$ and $c\mathcal{I}^- = T^{\varphi}_{\mathcal{R}_{Spec}} \uparrow i$, for some $i \geq 0$.

That is, we consider the abstract fixpoint semantics of $\mathcal{R}_{Spec}$ as over-approximation, whereas we take the set which results from a finite number of iterations of the $T^{\varphi}_{\mathcal{R}_{Spec}}$ function (the concrete operator) as under-approximation. This provides a simple, albeit useful, debugging scheme which is satisfactory in practice.

The following lemma is the key for our abstract diagnosis methodology.

Lemma 54. *Let $(c\mathcal{I}^+, c\mathcal{I}^-)$ be a computed approximation of the intended semantics $\mathcal{I}_{\mathcal{F}}$. Then, $(c\mathcal{I}^+, c\mathcal{I}^-)$ is a correct approximation of $\mathcal{I}_{\mathcal{F}}$.*

The following theorems formalize the abstract tests and are the main results of this section.

Theorem 55. *Let $(c\mathcal{I}^+, c\mathcal{I}^-)$ be a computed approximation of $\mathcal{I}_{\mathcal{F}}$. If there exists an equation e such that, $e \in T^{\varphi}_{\{r\}}(c\mathcal{I}^-)$ and e is not abstractly closed by $c\mathcal{I}^+$, then the rule $r \in \mathcal{R}$ is incorrect on e.*

Theorem 56. *Let $(c\mathcal{I}^+, c\mathcal{I}^-)$ be a computed approximation of $\mathcal{I}_{\mathcal{F}}$. If there exists an equation e such that $e \in c\mathcal{I}^-$ and e is not abstractly closed by $T^{\sharp\varphi}_{\mathcal{R}}(c\mathcal{I}^+)$, then e is uncovered in $\mathcal{R}$.*

The diagnosis w.r.t. approximate properties is always effective because the abstract specifications are finite. If no error is found, we say that $\mathcal{R}$ is *abstractly correct and complete w.r.t.* $(c\mathcal{I}^+, c\mathcal{I}^-)$. As one can expect, the results may be weaker than those that can be achieved on the concrete domain just because of the approximation: the fact that $\mathcal{R}$ is abstractly correct and complete w.r.t. $(c\mathcal{I}^+, c\mathcal{I}^-)$ does not generally imply the total correctness of $\mathcal{R}$ w.r.t. $\mathcal{I}$. The method is sound in the sense that each error which is found by using $\mathcal{I}^+, \mathcal{I}^-$ is really a bug w.r.t. $\mathcal{I}$. This is in contrast with the abstract diagnosis methodologies of [7,55,54], which work as follows: when the diagnoser finds that the program is correct, then it is certainly free of errors, whereas if an (abstract) error is reported, then it can be either a (concrete) error or not.

Let us illustrate this method by means of an example.

Example 57. Let us consider the following (wrong) Fibonacci program $\mathcal{R}$:

```
fib(0)           → 0                          add(0, x)    → x
fib(x)           → fibaux(0, 0, x)            add(s(x), y) → s(add(x, y))
fibaux(x, y, 0)  → x
fibaux(x, y, s(z)) → fibaux(y, add(x, y), z).
```

The specification is given by the following program $\mathcal{R}_{Spec}$:

```
fib(0)       → s(0)                       add(0, x)    → x
fib(s(0))    → s(0)                       add(s(x), y) → s(add(x, y))
fib(s(s(x))) → add(fib(s(x)), fib(x)).
```

Let $\varphi = inn$; then $\mathcal{R}^{\sharp}_{Spec}$ is

```
fib(0)       → s(0)                       add(0, x)    → x
fib(s(0))    → s(0)                       add(s(x), y) → s(♯)
fib(s(s(x))) → add(♯, ♯).
```

After two iterations of the $T^{inn}_{\mathcal{R}_{Spec}}$ operator, we get the following under-approximation:

$$\begin{aligned} c\mathcal{I}^- = \{&0 = 0, \mathtt{s(x)} = \mathtt{s(x)}, \mathtt{add(x,y)} = \mathtt{add(x,y)}, \mathtt{fib(x)} = \mathtt{fib(x)}, \mathtt{add(0,x)} = \mathtt{x},\\ &\mathtt{add(s(x),y)} = \mathtt{s(add(x,y))}, \mathtt{fib(0)} = \mathtt{s(0)}, \mathtt{fib(s(0))} = \mathtt{s(0)}, \mathtt{add(s(0),y)} = \mathtt{s(y)},\\ &\mathtt{fib(s^2(x))} = \mathtt{add(fib(s(x)),fib(x))}, \mathtt{add(s^2(x),y)} = \mathtt{s^2(add(x,y))},\\ &\mathtt{fib(s^2(0))} = \mathtt{add(s(0),fib(0))}, \mathtt{fib(s^2(0))} = \mathtt{add(fib(s(0)),s(0))},\\ &\mathtt{fib(s^3(x)))} = \mathtt{add(add(fib(s(x)),fib(x)),fib(s(x)))}\}. \end{aligned}$$

The over-approximation $c\mathcal{I}^+$ is given by the following set of equations (after three iterations of the $T^{\sharp inn}_{\mathcal{R}_{Spec}}$ operator, we get the fixpoint):

$$\begin{aligned} c\mathcal{I}^+ = \mathcal{F}^{\sharp}_{inn}(\mathcal{R}_{Spec}) = lfp(T^{\sharp inn}_{\mathcal{R}_{Spec}}) = \{&0 = 0,\ \mathtt{s(x)} = \mathtt{s(x)},\ \mathtt{add(x,y)} = \mathtt{add(x,y)},\ \mathtt{fib(x)} = \mathtt{fib(x)},\\ &\mathtt{add(0,x)} = \mathtt{x}, \mathtt{add(s(x),y)} = \mathtt{s(\sharp)}, \mathtt{fib(0)} = \mathtt{s(0)}, \mathtt{fib(s(0))} = \mathtt{s(0)},\\ &\mathtt{fib(s^2(x))} = \mathtt{add(\sharp,\sharp)}, \mathtt{fib(s^2(x))} = \sharp,\ \mathtt{fib(s^2(x))} = \mathtt{s(\sharp)}\} \end{aligned}$$

Now, consider the equation $\mathtt{fib(x)} = \mathtt{fib(x)}$ of $c\mathcal{I}^-$. By applying $T_{\{r\}}$ to this equation (with $r \equiv \mathtt{fib(0)} = \mathtt{0}$), we get the equation $e \equiv \mathtt{fib(0)} = \mathtt{0}$, which is not closed by $c\mathcal{I}^+$. This proves that r is incorrect on e.

We can also demonstrate the incompleteness of $\mathcal{R}$, by showing that the equation $\mathtt{fib(0)} = \mathtt{s(0)} \in \mathcal{I}^-$ is not closed by $T^{\sharp inn}_{\mathcal{R}}(\mathcal{I}^+)$.

The (wrong) abstract program $\mathcal{R}^{\sharp}$ is

$$\begin{aligned} &\mathtt{fib(0)} \to \mathtt{0} && \mathtt{add(0,x)} \to \mathtt{x}\\ &\mathtt{fib(x)} \to \mathtt{fibaux(0,0,x)} && \mathtt{add(s(x),y)} \to \mathtt{s(\sharp)}\\ &\mathtt{fibaux(x,y,0)} \to \mathtt{x}\\ &\mathtt{fibaux(x,y,s(z))} \to \sharp. \end{aligned}$$

Then, the equation $\mathtt{fib(0)} = \mathtt{s(0)}$ of $\mathcal{I}^-$ is not closed by $T^{\sharp inn}_{\mathcal{R}}(\mathcal{I}^+)$:

$$\begin{aligned} T^{\sharp inn}_{\mathcal{R}}(\mathcal{I}^+) = \{&0 = 0, \mathtt{s(x)} = \mathtt{s(x)}, \mathtt{fib(x)} = \mathtt{fib(x)},\ \mathtt{fibaux(x,y,z)} = \mathtt{fibaux(x,y,z)},\ \mathtt{add(0,x)} = \mathtt{x},\\ &\mathtt{add(x,y)} = \mathtt{add(x,y)},\ \mathtt{add(s(x),y)} = \mathtt{s(\sharp)},\ \mathtt{add(s(x),y)} = \sharp,\ \mathtt{fib(x)} = \mathtt{fibaux(0,0,x)},\\ &\mathtt{fib(s^2(x))} = \mathtt{s(\sharp)},\ \mathtt{fibaux(x,y,0)} = \mathtt{x},\ \mathtt{fibaux(x,y,s(z))} = \sharp\}. \end{aligned}$$

The following section presents a bug-correction technique that attempts to modify the erroneous components of the original code in order to correct the program. Then, we show how this mechanism can be combined within our diagnosis method in order to form a practical debugging system.

7. Program correction

Inductive logic programming (ILP) is the field of machine learning concerned with learning logic programs from positive and negative examples, generally in the form of ground literals [107]. A challenging subfield of ILP is known as *inductive theory revision,* which is close to program debugging under the *competent programmer* assumption of [117]. In other words, the initial program is assumed to be written with the intention of being correct and, if it is not, then a close variant of it is. The debugging technique we have developed attempts to find such a variant.

More 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 repaired, in order to produce $\mathcal{R}^c$. Therefore, 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 that is right. However, our approach has an important difference w.r.t. [117] and similar work in that we do not 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 one) or top-down (i.e. from an overly general rule to a more specific one). We mainly follow the top-down approach known as *example-guided unfolding* [40], which uses unfolding as a specialization operator, focusing on discriminating positive from negative examples. Unfortunately, it is known that the deduction process alone (i.e., unfolding) does not generally suffice for coming up with the corrected program, and inductive generalization techniques are necessary [59,111,112,70]. Therefore, we integrate our unfolding-based methodology with a bottom-up learner following a hybrid, top-down as well as bottom-up approach, which is able to infer program corrections that are hard, or even impossible, to obtain just by using deduction. The resulting blend of top-down and bottom-up synthesis is conceptually cleaner than more sophisticated, purely top-down or bottom-up ones and combines the advantages of both techniques.

7.1. Formalization of the program correction problem

We consider a program $\mathcal{R} \in \mathbb{R}^u$ along with an intended specification $\mathcal{I}$ such that $\mathcal{R}' \subseteq \mathcal{R}$ is a set of wrong rules w.r.t. $\mathcal{I}$, which have been detected by means of the abstract diagnosis of Section 6.

Moreover, let E^p and E^n be two disjoint sets of ground equations modeling the pursued as well as the unpursued computational behavior of $\mathcal{R}$ (see Definitions 59 and 60 below). Equations in E^p (respectively, E^n) are called *positive examples* (respectively, *negative examples*). Given an example set E, we say that $\mathcal{R}$ *entails* E using the strategy $\varphi \in \{inn, out\}$ (in symbols, $\mathcal{R} \vdash_\varphi E$) iff each $e \in E$ is proven in $\mathcal{R}$ using the strategy φ (i.e., e is reduced to *true* by using the rules of $\mathcal{R}$). Dually, $\mathcal{R}$ *disproves* E using $\varphi \in \{inn, out\}$ (in symbols, $\mathcal{R} \nvdash_\varphi E$) iff no $e \in E$ can be proven in $\mathcal{R}$ using φ.

The correction problem amounts to determining a set of rules $\mathcal{X}$ such that

$$\mathcal{R}^c = (\mathcal{R} \setminus \mathcal{R}') \cup \mathcal{X},\ \mathcal{R}^c \vdash_\varphi E^p \text{ and } \mathcal{R}^c \nvdash_\varphi E^n.$$

Program $\mathcal{R}^c$ will be called *corrected* program (w.r.t. E^p and E^n). We will call $\mathcal{R}^- = \mathcal{R} \setminus \mathcal{R}'$ the *diminished* program. Roughly speaking, a corrected program $\mathcal{R}^c$ is a program that entails the set of all the positive examples and disproves the set of all the negative examples.

7.2. Program correction by example-guided unfolding

An incorrect rule of a program $\mathcal{R}$ generally entails a subset of E^p (that is, equations that belong to the intended success set semantics $\mathcal{I}_{ca}$) and a subset of E^n (that is, equations in $\mathcal{O}^{ca}_\varphi(\mathcal{R})$ that do not belong to $\mathcal{I}_{ca}$). In other words, an incorrect rule is used to prove both positive and negative examples. In order to fix this erroneous behavior, we need to find a way of *guessing* a set of correct rules that entails all equations in E^p and no equation in E^n. For example, let E^p contain $\mathtt{even(0)} = \mathtt{true}$, $\mathtt{even(s^2(0))} = \mathtt{true}$, $\mathtt{even(s^4(0))} = \mathtt{true}$, ... and let E^n contain $\mathtt{even(s(0))} = \mathtt{true}$, $\mathtt{even(s^3(0))} = \mathtt{true}$, $\mathtt{even(s^5(0))} = \mathtt{true}$, ... Then, $\{\mathtt{even(0)} = \mathtt{true},\ \mathtt{even(s^2(x))} = \mathtt{even(x)}\}$ would be a corrected program.

Example-guided unfolding [3,40] is commonly applied to specialize the incorrect program $\mathcal{R}$ in order to exclude the negative examples without excluding the positive ones [40]. The basic idea of the method is as follows. We first specialize the program $\mathcal{R}$ by unfolding function calls on the right-hand sides of the rules yielding a close variant $\mathcal{R}'$ of $\mathcal{R}$. Then, we remove from $\mathcal{R}'$ those rules that allow us to derive negative examples. We will show that such rules can be safely deleted from the program without harming its behavior on the positive examples. The main insight for the method, formerly introduced in [39], is the following:

- unfolding tends to specialize (and shorten) the example rewrite sequences;
- if a negative example is proved by means of a rewrite sequence in which a rule r occurs and r is not used elsewhere for deriving a positive example, then the program can be "repaired" by deleting r.

Let us consider the following example.

Example 58. Let $\mathcal{R}$ be the program consisting of the following rules $\mathcal{R} = \{\mathtt{f(x)} \to \mathtt{g(x)},\ \mathtt{g(a)} \to \mathtt{a},\ \mathtt{g(b)} \to \mathtt{a}\}$. 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 described in Section 5, the rule $r : \mathtt{f(x)} \to \mathtt{g(x)}$ is incorrect. Now, let us choose the following example sets modeling the correct and the wrong program behavior: $E^p = \{\mathtt{f(a)} = \mathtt{a}\}$, and $E^n = \{\mathtt{f(b)} = \mathtt{a}\}$. Since r is used to prove both positive and negative examples, unfolding is applied in order to specialize the example rewrite sequences. Unfolding r upon $\mathtt{g(x)}$ w.r.t. the rules $\mathtt{g(a)} \to \mathtt{a}$ and $\mathtt{g(b)} \to \mathtt{a}$ replaces r with the following set of rules: $\mathcal{X} = \{\mathtt{f(a)} \to \mathtt{a},\ \mathtt{f(b)} \to \mathtt{a}\}$.

Observe that the second rule only occurs in the rewrite sequence of the negative example (either with $\varphi = inn$ or $\varphi = out$). Consequently, we can delete the rule $\mathtt{f(b)} \to \mathtt{a}$ from $(\mathcal{R} \setminus \{r\}) \cup \mathcal{X}$ since it does not affect the positive example reduction, and thus achieve a corrected program:

$$\mathcal{R}^c = \{\mathtt{f(a)} \to \mathtt{a},\ \mathtt{g(a)} \to \mathtt{a},\ \mathtt{g(b)} \to \mathtt{a}\}.$$

In the remainder of this section, we adapt this method to our debugging setting. First of all, we provide a simple technique, which exploits the outcomes of the abstract analysis performed during the diagnosis process, to automatically generate the example sets E^p and E^n. Then, we formalize the unfolding-based correction methodology we outlined above.

7.2.1. Generation of the example sets

When a negative example is entailed by the current version of the program, there is at least one rule which is responsible for the incorrect proof. In order to develop 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 $\mathcal{R}^-$ does not contain any incorrect rule. However, it might be incomplete w.r.t. the intended semantics as there can be ground equations which are proven using the specification[12] $\mathcal{I}$, but not using $\mathcal{R}^-$. Such equations are sensible positive examples since the computed corrected program has to entail them. Hence, we define the following set.

Definition 59 (*Positive example set*)**.** The set of *positive examples* E^p is defined as follows:

$$E^p = \{f(\bar{t}) = d \in c\mathcal{I}^- \mid \mathcal{R}^- \nvdash_\varphi f(\bar{t}) = d, f(\bar{t}) \in \tau(\Sigma), d \in \tau(\mathcal{C}), f \text{ is a defined symbol of } \mathcal{R}\}.$$

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

Similarly, we let E^n define the set of equations that allow the debugger to prove that some rule r of $\mathcal{R}$ is incorrect w.r.t. $\mathcal{I}$ using φ. Hence, we define the set E^n.

Definition 60 (*Negative example set*)**.** The set of *negative examples* E^n is defined as follows:

$$E^n = \{f(\bar{t}) = d \in T^{\varphi}_{\{r\}}(c\mathcal{I}^-) \mid r \text{ is incorrect on } f(\bar{t}) = d \text{ for some r} \in \mathcal{R}, f(\bar{t}) \in \tau(\Sigma), d \in \tau(\mathcal{C}), \\ f \text{ is a defined symbol of } \mathcal{R}\}.$$

Given a computable approximation $(c\mathcal{I}^+, c\mathcal{I}^-)$ of the intended fixpoint semantics $\mathcal{I}_{\mathcal{F}}$, the set E^p can be computed by exploiting Theorem 56, which provides a computable, abstract test for detecting uncovered equations w.r.t. $(c\mathcal{I}^+, c\mathcal{I}^-)$, while the set E^n is generated by using the computable, abstract diagnosis test for rule incorrectness w.r.t. $(c\mathcal{I}^+, c\mathcal{I}^-)$ of Theorem 55.

Note that since program $\mathcal{R}$ and specification $\mathcal{I}$ might use different auxiliary functions, we only consider ground examples of the form $f(\bar{t}) = d$ where $f(\bar{t})$ is a ground pattern calling a function defined in $\mathcal{R}$ and d is a ground constructor term.[13] In this way, the inductive process becomes independent of the extra functions contained in $\mathcal{I}$ since we start synthesizing directly from the data structures that occur in d and the functions defined in $\mathcal{R}$.

It is also worth noting that if we wanted to increase the number of examples, we could consider equations which may contain variables. In this case, we could instantiate such equations with ground constructors (typically provided by the user) in order to get a larger number of positive/negative ground examples. Actually, the prototypical implementation we present in Section 8 allows us to perform such an instantiation process.

Example 61. Consider the wrong program $\mathcal{R}$, and the corresponding specification $\mathcal{I}$:

$$\begin{array}{lll} \mathtt{odd(s(x))} & \to & \mathtt{odd(x)} \\ \mathtt{odd(0)} & \to & \mathtt{true} \end{array} \qquad \begin{array}{lll} \mathtt{odd(s(x))} & \to & \mathtt{true} \Leftarrow \mathtt{even(x)} = \mathtt{true} \\ \mathtt{even(s(s(x)))} & \to & \mathtt{even(x)} \\ \mathtt{even(0)} & \to & \mathtt{true}. \end{array}$$

Consider the following computed approximation $(c\mathcal{I}^+, c\mathcal{I}^-)$ of $\mathcal{I}$ (using the narrowing strategy *inn*), with $c\mathcal{I}^- = T^{inn}_{\mathcal{I}} \uparrow 2$.

$$c\mathcal{I}^+ = \{\mathtt{odd(s(0))} = \mathtt{true}, \mathtt{odd(s^3(\sharp))} = \mathtt{true}, \mathtt{odd(x)} = \mathtt{odd(x)}, \\ \mathtt{even(0)} = \mathtt{true}, \mathtt{even(s^2(\sharp))} = \sharp, \mathtt{even(x)} = \mathtt{even(x)}, \\ \mathtt{s(x)} = \mathtt{s(x)}, 0 = 0, \mathtt{true} = \mathtt{true}\}$$

$$c\mathcal{I}^- = \{\mathtt{odd(s(0))} = \mathtt{true}, \mathtt{odd(s^3(0))} = \mathtt{true}, \mathtt{odd(x)} = \mathtt{odd(x)}, \mathtt{even(0)} = \mathtt{true}, \\ \mathtt{even(s^2(0))} = \mathtt{true}, \mathtt{even(x)} = \mathtt{even(x)}, \mathtt{even(s^2(x))} = \mathtt{even(x)}, \\ \mathtt{even(s^4(x))} = \mathtt{even(x)}, \mathtt{s(x)} = \mathtt{s(x)}, 0 = 0, \mathtt{true} = \mathtt{true}\}.$$

By applying Theorem 55, we discover that all the rules in $\mathcal{R}$ are incorrect on some equations in $T^{inn}_{\mathcal{R}}(c\mathcal{I}^-)$. Specifically, $\mathtt{odd(0)} \to \mathtt{true}$ is incorrect on $\mathtt{odd(0)} = \mathtt{true}$ and $\mathtt{odd(s(x))} \to \mathtt{odd(x)}$ is incorrect on $\mathtt{odd(s(x))} = \mathtt{odd(x)}$. Therefore, the diminished program $\mathcal{R}^-$ becomes

$$\mathcal{R}^- = \mathcal{R} \setminus \{\mathtt{odd(0)} \to \mathtt{true}, \mathtt{odd(s(x))} \to \mathtt{odd(x)}\} = \emptyset.$$

Consequently, by Definitions 59 and 60, we obtain the following example sets: $E^p = \{\mathtt{odd(s(0))} = \mathtt{true}, \mathtt{odd(s^3(0))} = \mathtt{true}\}$ and $E^n = \{\mathtt{odd(0)} = \mathtt{true}\}$.

7.2.2. Unfolding operators

In the functional logic setting, 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 [19] for a complete description). A complete characterization of unfolding w.r.t. computed answers in functional logic languages with eager/lazy semantics can be found in [16]. Following [40], we use the unfolding operator for program correction.

Roughly speaking, *unfolding* a program $\mathcal{R}$ w.r.t. a rule r yields a new specialized version of $\mathcal{R}$ in which the rule r is replaced by new rules obtained from r by performing a narrowing step on the right-hand side of r. Typically, unfolding is non-deterministic since several subterms on the right-hand side of a rule may be narrowable. In our framework, we will take advantage of a deterministic version of unfolding, namely the *leftmost-innermost* unfolding, in which only the leftmost-innermost narrowing redex of the right-hand side is reduced according to the *inn* narrowing strategy. Thus, on the one hand, we are able to shrink the narrowing search space and consequently provide a faster correction algorithm; on the other hand, this will allow us to prove the soundness of our repair methodology for the class of uniform programs $\mathbb{R}^u$.

[13] For terminating CTRSs, we can consider examples whose right-hand sides are not constructor terms by normalizing the right-hand side of the positive examples w.r.t. $\mathcal{I}$ (resp. $\mathcal{R}$ for the negative examples) using φ.

Definition 62 (*Unfolding operators*)**.** Let $\mathcal{R}$ be a program.

(i) Let $r_1, r_2 \ll \mathcal{R}$ such that $r_1 \equiv (\lambda_1 \to \rho_1 \Leftarrow C_1)$ and $r_2 \equiv (\lambda_2 \to \rho_2 \Leftarrow C_2)$. The *rule unfolding* of r_1 w.r.t. r_2 is defined as follows:

$$\mathsf{U}_{r_2}(r_1) = \{\lambda_1\sigma \to \rho' \Leftarrow C' \mid (\rho_1 = y, C_1) \overset{\sigma, r_2, u}{\rightsquigarrow}_{inn} (\rho' = y, C'), u \in \overline{O}(\rho_1)\},$$

where y is a fresh variable.

(ii) Let $r \ll \mathcal{R}$. The *program unfolding* of r w.r.t. $\mathcal{R}$ is as follows:

$$\mathsf{U}_{\mathcal{R}}(r) = \left(\mathcal{R} \cup \bigcup_{r' \in \mathcal{R}} \mathsf{U}_{r'}(r)\right) \setminus \{r\}.$$

Note that the absence of narrowable positions in the rule r to be unfolded yields no specialization of r. We just get the removal of r from $\mathcal{R}$. In the sequel, we use the following notion of "unfoldable rule".

Definition 63 (*Unfoldable rule*)**.** Let $\mathcal{R}$ be a program and r be a rule in $\mathcal{R}$. The rule r is *unfoldable* w.r.t. $\mathcal{R}$ if $\mathsf{U}_{\mathcal{R}}(\mathsf{r}) \neq \mathcal{R} \setminus \{\mathsf{r}\}$.

Definition 64. Let $\mathcal{R} \in \mathbb{R}_{inn}$ be a program. The *unfolding succession* $\mathcal{S}(\mathcal{R}) \equiv \mathcal{R}_0, \mathcal{R}_1, \ldots$ of program $\mathcal{R}$ is defined as follows:

$$\begin{aligned} \mathcal{R}_0 &= \mathcal{R} \\ \mathcal{R}_{i+1} &= \mathsf{U}_{\mathcal{R}_i}(\mathsf{r}), \quad \text{where } \mathsf{r} \in \mathcal{R}_\mathsf{i} \text{ is unfoldable.} \end{aligned}$$

As proved in [16], each program $\mathcal{R}_i$, $i = 0, 1, \ldots$ has the same ground as well as nonground semantics provided that $\mathcal{R}_i \in \mathbb{R}_{inn}$. This guarantees that if a given example e is proven in $\mathcal{R} \equiv \mathcal{R}_0$, then e is also proven in $\mathcal{R}_i$, for any $i = 1, 2, \ldots$. Moreover, we can show that the successful rewrite sequence used to prove a given example e in $\mathcal{R}$ is longer than the one used to prove e in $\mathcal{R}'$, where $\mathcal{R}'$ is obtained by applying the unfolding operator to $\mathcal{R}$. This result is formally stated by Proposition 66. The following definition is auxiliary.

Definition 65. Let e be an equation. A rewrite sequence for e w.r.t. $\mathcal{R}$, $\mathcal{D}_{\mathcal{R}}(e) : e \equiv e_1 \overset{r_1,p_1}{\to} e_2 \overset{r_2,p_2}{\to} \cdots e_{n-1} \overset{r_{n-1},p_{n-1}}{\to} e_n$, is *leftmost innermost*, whenever each $e_{i|p_i}, i = 1, \ldots, n-1$, is the leftmost innermost redex in e_i. Besides, the set $\mathsf{OR}(\mathcal{D}_{\mathcal{R}}(e)) = \{r_1, r_2, \ldots, r_n\}$ is called the set of *occurring rules* of $\mathcal{D}_{\mathcal{R}}(e)$. A leftmost innermost rewrite sequence for an equation e w.r.t. $\mathcal{R}$ is called *successful*, if it is of the form $e \equiv e_1 \overset{r_1,p_1}{\to} e_2 \overset{r_2,p_2}{\to} \ldots e_{n-1} \overset{r_{n-1},p_{n-1}}{\to} e_n \equiv \mathit{true}$, and its corresponding *rule application sequence* is $\langle r_1, \ldots, r_{n-1}\rangle$.

Successful leftmost innermost rewrite sequences are also called *proofs*. In the following, we denote by $|\mathcal{D}_{\mathcal{R}}(e)|$ the *length* of a leftmost innermost rewrite sequence $\mathcal{D}_{\mathcal{R}}(e)$.

In order to select the rules for specialization, we need the following auxiliary definition. Given the program $\mathcal{R}$, a *discriminable rule* of $\mathcal{R}$ is a rule that is unfoldable w.r.t. $\mathcal{R}$ and that occurs in the proof of, at least, one positive example.

The following result essentially states that the length of the proofs for the considered examples is shortened by innermost unfolding.

Proposition 66. *Let $\mathcal{R} \in \mathbb{R}_{inn}$, $\mathcal{R}' = \mathsf{U}_r(\mathcal{R})$, $r \in \mathcal{R}$ be a discriminable rule, and e be an equation. Then, we have*

(1) *if $e \to^* \mathit{true}$ in $\mathcal{R}$, then also $e \to^* \mathit{true}$ in $\mathcal{R}'$;*
(2) *if $r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}}(\mathsf{e}))$, then $|\mathcal{D}_{\mathcal{R}'}(e)| < |\mathcal{D}_{\mathcal{R}}(e)|$;*
(3) *if $e \to^* \mathit{true}$ in $\mathcal{R}'$, then also $e \to^* \mathit{true}$ in $\mathcal{R}$.*

7.2.3. The top-down correction algorithm

We formulate a basic algorithm that specializes programs w.r.t. positive and negative examples by applying the rule unfolding transformation together with rule removal. The (backbone of the) procedure for program correction is described in Algorithm 1. The procedure is inspired by [3], which is known to produce a correct specialization when the program is *overly general* (with some extra outfit which is needed to specialize recursive definitions [40]); that is, it allows us to prove all positive examples and some incorrect ones.

Definition 67 (*Overly general program*)**.** Let $\mathcal{R}$ be a program and E^p be a set of positive examples. Then, $\mathcal{R}$ is *overly general* w.r.t. E^p iff $\mathcal{R} \vdash_{\varphi} E^p$.

Note that the definition above allows the case where $\mathcal{R}$ contains no incorrect rules as it is only aimed at ensuring soundness. However, in order to achieve effectiveness of the transformation, at least one negative example is required to drive the example-guided correction procedure, which essentially generates a sequence of program transformations ending up in a corrected program.

Using the unfolding operator, it is possible to generate a succession of "specialized" programs in the following way. Algorithm 1 works in two phases: the *unfolding phase* and the *deletion phase*. Roughly speaking, we first perform unfolding upon (arbitrarily selected) discriminable rules until we get a specialized version $\mathcal{R}_i$ of the program $\mathcal{R}$ where each negative example can be proven by applying at least one rule not occurring in the proof of any positive examples (*unfolding phase*).

Then, those rules that only contribute to the proofs of the negative examples can be safely removed without compromising the proofs of the positive examples (*deletion phase*). The program, which we obtain after the deletion phase, is a corrected program.

More formally, the key idea of the algorithm is to apply unfolding until we get a specialized program $\mathcal{R}_i$ such that the following property holds:

$$\forall\, e^n \in E^n, \exists\, r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^n)) \text{ such that } \forall\, e^p \in E^p,\ r \notin \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^p)).$$

Then, the deletion phase removes all the rules that are only used to prove the negative examples.

Let us illustrate the algorithm with the following example.

Example 68. Consider the wrong program $\mathcal{R}$, and the specification $\mathcal{I}$ of Example 61:

```
odd(s(x)) → odd(x)          odd(s(x))      → true ⇐ even(x) = true
odd(0)    → true            even(s(s(x))) → even(x)
                            even(0)       → true
```

As shown in Example 61, the following sets of examples can be computed, for $\varphi = inn$:

$E^p = \{\mathtt{odd}(\mathtt{s}^3(\mathtt{0})) = \mathtt{true}, \mathtt{odd}(\mathtt{s}(\mathtt{0})) = \mathtt{true}\}$
$E^n = \{\mathtt{odd}(\mathtt{0}) = \mathtt{true}\}$.

Since the rule $\mathtt{odd}(\mathtt{0}) \to \mathtt{true}$ is used to prove positive as well as negative examples, we enter the main loop of the top-down correction algorithm. Then, by unfolding the discriminable rule $\mathtt{odd}(\mathtt{s}(\mathtt{x})) \to \mathtt{odd}(\mathtt{x})$, we get the following unfolded program $\mathcal{R}_1$

```
    odd(0) → true
  odd(s(0)) → true
odd(s(s(x))) → odd(x).
```

Now, in every negative example proof w.r.t. $\mathcal{R}_1$, there appears at least one rule that does not occur in the proofs of any positive example; thus, the unfolding phase ends and we enter the deletion phase, which "purifies" $\mathcal{R}_1$ by removing the rule $\mathtt{odd}(\mathtt{0}) \to \mathtt{true}$ that only occurs in the proof of a negative example. Therefore, as outcome, we obtain the program

```
  odd(s(0)) → true
odd(s(s(x))) → odd(x)
```

which is a corrected program w.r.t. the considered E^p and E^n.

Example 68 not only shows us how the algorithm works but also allows us to clarify the differences between the preliminary correction algorithm in [9] and the one presented in this paper. The algorithm in [9] only unfolds incorrect rules, whereas the new correction procedure does consider any discriminable rule for unfolding, which is generally needed in order to achieve the correction. In fact, the previous approach would not work in Example 68, as it would try to unfold the incorrect rule $\mathtt{odd}(\mathtt{0}) \to \mathtt{true}$, which is not discriminable.

Algorithm 1 The top-down correction algorithm.

1: **function** TD-CORRECTOR($\mathcal{R}$, $c\mathcal{I}^+$, $c\mathcal{I}^-$)
2: $(E^p, E^n) \leftarrow$GENERATEEXAMPLESETS($\mathcal{R}$, $c\mathcal{I}^+$, $c\mathcal{I}^-$)
3: **if** $\mathcal{R} \not\models_\varphi E^p$ **then** HALT
4: **end if**
5: $i \leftarrow 0$ ▷ Unfolding phase
6: $\mathcal{R}_0 \leftarrow \mathcal{R}$
7: **while** $\exists\, e^n \in E^n, e^p \in E^p, r \in \mathcal{R}\ (r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^n)) \wedge r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^p)))$ **do**
8: SELECT a discriminable rule $r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^p))$ of $\mathcal{R}_i$, for some $e^p \in E^p$
9: $\mathcal{R}_{i+1} \leftarrow \mathsf{U}_{\mathcal{R}_i}(\mathsf{r})$
10: $i \leftarrow i + 1$
11: **end while**
12: **for all** $e^n \in E^n$ **do** ▷ Deletion phase
13: $\mathcal{R}_{i+1} \leftarrow \mathcal{R}_i \setminus \{r\}$, where $r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^n)) \wedge \forall e^p \in E^p\ (r \notin \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^p)))$
14: $i \leftarrow i + 1$
15: **end for**
16: $\mathcal{R}^c \leftarrow \mathcal{R}_i$
17: **return** $\mathcal{R}^c$
18: **end function**

7.2.4. Correctness of the algorithm

We prove the soundness of our repair methodology for the class of uniform programs $\mathbb{R}^u$ [60]. As we recalled in Section 2.1, completeness of the *inn* as well as the *out* narrowing strategy is guaranteed for this class of programs. Additionally, the semantics of uniform programs is preserved by leftmost-innermost unfolding transformations in the sense that any equation e is provable in the original program iff e is provable in the unfolded program (see Corollary 96 in Appendix B). These two facts are crucial in our proof scheme and allow us to prove that the repaired programs we obtain can be safely used with both the *inn* and the *out* narrowing strategy without losing completeness of the chosen evaluation strategy.

We proceed as follows.

(i) First, we show that, if $\mathcal{R}$ is overly general w.r.t. E^p, the unfolding phase produces a specialized version $\mathcal{R}'$ of $\mathcal{R}$ (still overly general w.r.t. E^p) such that, for each negative example, there is a rule occurring in the corresponding proof that is not used in the proof of any positive example.
(ii) Next, we show that the deletion phase yields a corrected version of $\mathcal{R}$ such that $\mathcal{R} \vdash_\varphi E^p$ and $\mathcal{R} \not\vdash_\varphi E^n$.

The following proposition proves our claim (i): by a suitable finite number of applications of the unfolding operator to a given program, we get a specialized program such that, the proofs of negative examples contain at least one rule that is never applied for proving any positive example. A condition is necessary for proving this result: no negative/positive couple of the considered examples can have the same application rule sequence, as shown in the following counterexample.

Example 69. Consider the program $\mathcal{R}$

$$r_1 : \mathtt{f(x)} \to \mathtt{g(x)}$$
$$r_2 : \mathtt{g(x)} \to \mathtt{0}$$

with example sets $E^p = \{\mathtt{f(a)} = \mathtt{0}\}$, $E^n = \{\mathtt{f(b)} = \mathtt{0}\}$. Then, $\mathtt{f(a)} = \mathtt{0}$ and $\mathtt{f(b)} = \mathtt{0}$ are proven by using the same rule application sequence (i.e., $\langle r_1, r_2 \rangle$). By applying the top-down algorithm, we unfold rule $\mathtt{f(x)} \to \mathtt{g(x)}$, which produces the outcome $\mathcal{R}_1 = \{\mathtt{f(x)} \to \mathtt{0}, \mathtt{g(x)} \to \mathtt{0}\}$.

Note that $\mathcal{R}_1$ cannot be repaired by deleting rules since removing the wrong rule $\mathtt{f(x)} \to \mathtt{0}$ would cause the loss of E^p.

Proposition 70. *Let $\mathcal{R} \in \mathbb{R}_{inn}$. Let E^p (resp. E^n) be a set of positive (resp. negative) examples. If there are no $e^p \in E^p$ and $e^n \in E^n$ that can be proven in $\mathcal{R}$ by using the same rule application sequence, then, for each unfolding succession $\mathcal{S}(\mathcal{R})$, there exists k such that $\forall e^n \in E^n,\ \exists r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_k}(\mathtt{e^n}))$ s.t. r is not discriminable.*

We note that Proposition 70 holds for every unfolding succession of the original program; this implies that the rule to be unfolded at each unfolding step can be arbitrarily selected, provided that it is discriminable. Moreover, the termination of the unfolding phase is granted by the finite number k of applications of the unfolding operator needed to obtain the program $\mathcal{R}_k$.

After the unfolding phase, the proof of every negative example contains a rule of $\mathcal{R}_k$ not occurring in the proof of any positive example, thus we can safely remove this rule without jeopardizing completeness (claim (ii)). In other words, the deletion phase purges $\mathcal{R}_k$ of those rules that are only needed to reduce negative examples and yields a program which is correct w.r.t. both positive and negative examples.

Theorem 71 (*Correctness*)**.** *Let $\mathcal{R} \in \mathbb{R}^u$. Let $\mathcal{I}_\mathcal{F}$ be the intended fixpoint semantics of $\mathcal{R}$, and $(c\mathcal{I}^+, c\mathcal{I}^-)$ be a computed approximation of $\mathcal{I}_\mathcal{F}$. Then, if E^p and E^n are two sets of examples generated w.r.t. $(c\mathcal{I}^+, c\mathcal{I}^-)$ such that $\mathcal{R} \vdash_\varphi E^p$, $\varphi \in \{inn, out\}$, and there are no $e^p \in E^p$ and $e^n \in E^n$ which can be proven in $\mathcal{R}$ by using the same rule application sequence, then the execution of* TD-Corrector$(\mathcal{R}, c\mathcal{I}^+, c\mathcal{I}^-)$ *yields a corrected program $\mathcal{R}^c$ w.r.t. E^p and E^n.*

As in other approaches for example-guided program correction, derived programs might need to be newly diagnosed for correctness at the end of the correction process. This is because correctness of $\mathcal{R}$ w.r.t. E^p and E^n does not generally imply that the program is correct w.r.t. the intended semantics unless the considered sets E^p and E^n cover all counterexamples that might be derived from any pair $(c\mathcal{I}^+, c\mathcal{I}^-)$.

7.3. Improving the correction algorithm: a hybrid approach

In the following, we propose a bottom-up correction methodology that we smoothly combine with the top-down approach of Section 7.2 in order to correct programs that do not fulfil the applicability condition (over-generality). The methodology consists in applying a bottom-up pre-processing to "generalize" the initial wrong program, before proceeding to the usual top-down correction. In other words, we extend the original program with new synthesized rules so that the entire example set E^p succeeds w.r.t. the generalized program, and hence the top-down corrector can be effectively applied.

7.3.1. Bottom-up generation of overly general (wrong) programs

Our generalization method directly employs the bottom-up technique for the inductive learning of functional logic programs developed by Ferri, Hernández and Ramírez [87,70] which is able to produce an intensional description (expressed

by a functional logic program) of a set of ground examples. The technique is also able to introduce functions, defined as a background theory, in the inferred intensional description (see [87,70] for details). In the following, we recall the definitions of *restricted generalization* and *inverse narrowing*, which are the heart of the bottom-up procedure of [87,70]. The former allows one to generalize program rules, the latter is needed to introduce defined symbols on the right-hand sides of the synthesized rules.

Definition 72 (*Generalization operator*)**.** The rule $r' \equiv (s' \to t' \Leftarrow C')$ is a *restricted generalization* of $r \equiv (s \to t \Leftarrow C)$ if there exists a substitution θ such that (i) $\theta(r') \equiv r$; (ii) $Var(t') \subseteq Var(s')$. The *generalization operator* $\mathsf{RG}(r)$ is defined as follows: $\mathsf{RG}(r) = \{r' | r' \text{ is a restricted generalization of } r\}$.

Roughly speaking, the notion of inverse narrowing is inspired in Muggleton's inverse resolution operator [106], which essentially reverses the classical deductive inference process in order to generate valid premises (typically, in the form of logic programs) from known consequences (i.e. examples). The inverse narrowing operator of [87,70] is defined as follows.

Definition 73 (*Inverse narrowing operator*)**.** The rule $r \equiv s \to t \Leftarrow C$ *inversely narrows* into $r' \equiv s\theta \to t' \Leftarrow C'$ (in symbols $r \overset{u,r'',\theta}{\hookleftarrow} r'$) iff there exist a position $u \in O(t)$ and a rule $r'' \equiv \lambda \to \rho \Leftarrow C''$ such that (i) $\theta = mgu(t_{|u}, \rho)$; (ii) $t' = (t[\lambda]_u)\theta$; (iii) $C' = (C'', C)\theta$.
The *inverse narrowing* operator $\mathsf{INV}(r, r'')$ is given by

$$\mathsf{INV}(r, r'') = \{\bar{r} \mid r \overset{u,r'',\theta}{\hookleftarrow} r' \text{ and } \bar{r} \text{ is computed by instantiating the extra-variables on the right-hand side of } r' \text{ with variables on the left-hand side of } r\}.$$

Following the terminology of [87,70], inverse narrowing takes as an input a pair of rules: the sender (or applied) rule r'' and the receiver (or transformed) rule r. After computing r' by an inverse narrowing step over r by using r'', a new rule $\bar{r}$ is obtained by instantiating the extra-variables on the right-hand side of r' with variables on the left-hand side of r. This instantiation is done in such a way that every possible combination is performed, while avoiding to instantiate two different extra variables to the same variable name. If the new rule has more extra variables than the number of variables on the left-hand side, then the rule is rejected. We note that FLIP is based on an incremental algorithm [70] which provides a number of optimality criteria and good heuristics that allow one to select the best programs w.r.t. the considered examples without the need to compute all possible instantiations in advance.

Example 74. By applying the inverse narrowing operator between rules

$$r_1 \equiv \mathtt{add(x, y)} \to \mathtt{s(x)} \Leftarrow \mathtt{y = s(0)} \quad r_2 \equiv \mathtt{add(x', 0)} \to \mathtt{x'},$$

we get

$$\mathsf{INV}(r_1, r_2) = \{\mathtt{add(x, y)} \to \mathtt{add(s(x), 0)} \Leftarrow \mathtt{y = s(0)}, \mathtt{add(x, y)} \to \mathtt{s(add(x, 0))} \Leftarrow \mathtt{y = s(0)}\}.$$

The extra instantiation of the variables on the right-hand sides of the synthesized rules is necessary since inverse narrowing may introduce extra-variables, which are not allowed.

Example 75. Let us consider the rule $r_1 \equiv \mathtt{f(x, y)} \to \mathtt{c}$ and rule $r_2 \equiv \mathtt{g(x')} \to \mathtt{c}$. The result of an inverse narrowing step on r_1w.r.t. r_2 is the rule $r_3 \equiv \mathtt{f(x, y)} = \mathtt{g(x')}$ which contains the extra-variable $\mathtt{x'}$. The operator INV then instantiates the extra-variable $\mathtt{x'}$ to variables appearing on the left-hand side of r_3. Hence, $\mathsf{INV}(r_1, r_2) = \{\mathtt{f(x, y)} = \mathtt{g(x)}, \mathtt{f(x, y)} = \mathtt{g(y)}\}$.

The following definitions are helpful for discerning the overspecialized program rules. $Def_{\mathcal{R}}(f)$ is the set of rules in $\mathcal{R}$ that defines the function f. This might be computed by constructing a functional dependency graph of the program $\mathcal{R}$ and by statically analyzing it. Given a set E of positive examples, $Res_f(E)$ denotes the restriction of E to the set of f-rooted examples (that is, examples of the form $f(\bar{t}) = d$). We say that a function definition $Def_{\mathcal{R}}(f)$ is *overspecialized* w.r.t. the set of positive examples E^p, if there exists $e \in Res_f(E^p)$ which is not entailed by $Def_{\mathcal{R}}(f)$. An incorrect rule belonging to an overspecialized function definition is called an *overspecialized* rule.

The generalization algorithm works as follows. In its initial phase, it discovers all the function definitions that are overspecialized w.r.t. the set of positive examples E^p, by computing the subset of f-rooted examples not provable in $\mathcal{R}$ (and, hence, not provable by the corresponding function definition). Then, overspecialized rules are deleted from $\mathcal{R}$. Now, by applying generalization and inverse narrowing operators, we try to reconstruct the missing part of the code. More formally, we synthesize a functional logic program $\mathcal{A}$ such that $\mathcal{R} \cup \mathcal{A} \setminus \{r \in \mathcal{R} \mid r \text{ is overspecialized}\}$ allows us to prove the entire E^p. At the end, we get an overly general program to which the top-down corrector can be applied for repairing (incorrectness) bugs on the derived overly general faulty rules.

Following [70], the bottom-up synthesis algorithm first generates a set P_H (Program Hypothesis set), which consists of programs that contain exactly one rewrite rule and that are associated with the restricted generalizations of E^p, that is, $P_H = \{\{r\} \mid r \in \mathsf{RG}(s \to t), s = t \in E^p\}$. Then, it enters a loop in which, by means of INV and RG operators, new programs in P_H are produced. The algorithm leaves the loop when an "optimal" solution, which entails E^p entirely, has been found in P_H, or a maximal number of iterations is reached. In the latter case no solution might be found.

Algorithm 2 Bottom-up Synthesis Algorithm.

function BU-GENERALIZE($\mathcal{R}$, E^p)
  $S \leftarrow \{\mathcal{R}' \mid \mathcal{R}' = Def_{\mathcal{R}}(f), f \in \mathcal{F}\}$
  $\mathcal{R}_{aux} \leftarrow \mathcal{R}$
  for all overspecialized $\mathcal{R}'$ in S **do**
    if $r \in \mathcal{R}' \wedge r$ is incorrect **then** $\mathcal{R}_{aux} \leftarrow \mathcal{R}_{aux} \setminus \{r\}$
    end if
  end for
  $P_H \leftarrow \{\{r\} \mid r \in \mathsf{RG}(s \to t), s = t \in E^p\}$
  $it \leftarrow 0$
  while $\neg(SelectBest(P_H) \vdash E^p \wedge SelectBest(P_H) > Opt) \wedge it < it_{\max}$ **do**
    select the best $\mathcal{R}' \in P_H$ and $\mathcal{R}'' \in P_H \cup \{\mathcal{R}^-\}$ w.r.t. the optimality criterion
    select the best $r' \in \mathcal{R}'$ and $r'' \in \mathcal{R}''$ w.r.t. the optimality criterion
    for all $r \in \mathsf{INV}(r', r'')$ **do**
      for all $rg \in \mathsf{RG}(r)$ **do**
        $P_H \leftarrow P_H \cup \{\mathcal{R}' \cup \mathcal{R}'' \setminus \{r'\} \cup \{rg\}\}$
      end for
    end for
    $it \leftarrow it + 1$
  end while
  $\mathcal{A} = SelectBest(P_H)$
  if $\mathcal{A} \vdash E^p \wedge \mathcal{A} > Opt$ **then**
    $\mathcal{R}_{gen} \leftarrow \mathcal{R}_{aux} \cup \mathcal{A}$
  else output('No solution found')
  end if
end function

Due to the huge search space that this method involves, some heuristics must be implemented to guide the search. *Minimum description length* (MDL)[14] and *covering factor*[15] criteria could be taken into consideration so that inverse narrowing steps are only performed among the best programs and rules w.r.t. these criteria. Moreover, by means of MDL and covering factor, only the most concise programs are selected during the induction process. The notion of *optimality* w.r.t. programs and equations could be defined as a linear combination of these two criteria [87]. Algorithm 2 sketches a high-level procedure for the bottom-up synthesis which is based on [70,87]. We refer to [70,87] for an in-depth formalization of the bottom-up algorithm.

It is worth noting that the induction process is guided by search heuristics which generally (but not always) allow one to come up with the desired solution. This implies that the bottom-up generalization algorithm might not generate a solution, that is, a suitable generalization of the wrong program that can be fixed using our top-down methodology. In this case, no correction can be inferred and thus the debugging process will simply remove the incorrect rules from the wrong program under examination.

In our last example, we only pinpoint the relevant outcomes for the sake of clarity.

Example 76. Consider the following (wrong) program and the specification:

$$\mathcal{R} = \{\ \mathtt{playdice(x)} \to \mathtt{dd(winface(x))},\ \mathtt{dd(0)} \to \mathtt{0},\ \underline{\mathtt{dd(s(x))} \to \mathtt{dd(x)}},\ \underline{\mathtt{winface(s(x))} \to \mathtt{s(winface(x))}},\ \underline{\mathtt{winface(0)} \to \mathtt{0}}\ \}$$

$$\mathcal{I} = \{\ \mathtt{playdice(x)} \to \mathtt{dd(winface(x))},\ \mathtt{dd(x)} \to \mathtt{sum(x,x)},\ \mathtt{sum(x,0)} \to \mathtt{x},\ \mathtt{sum(x,s(y))} \to \mathtt{s(sum(x,y))},\ \mathtt{winface(s(0))} \to \mathtt{s(0)},\ \mathtt{winface(s(s(0)))} \to \mathtt{s(s(0))}\ \}.$$

Program rules that are signalled as incorrect by the diagnosis system are underlined. The example generation procedure described in Section 7.2.1 can produce the following example sets:

$$E^p = \{\ \mathtt{playdice}(\mathtt{s}^2(\mathtt{0})) = \mathtt{s}^4(\mathtt{0}),\ \mathtt{playdice}(\mathtt{s(0)}) = \mathtt{s}^2(\mathtt{0}),\ \mathtt{dd}(\mathtt{s}^4(\mathtt{0})) = \mathtt{s}^8(\mathtt{0}),\ \mathtt{dd}(\mathtt{s}^3(\mathtt{0})) = \mathtt{s}^6(\mathtt{0}),\ \mathtt{dd}(\mathtt{s}^2(\mathtt{0})) = \mathtt{s}^4(\mathtt{0}),\ \mathtt{dd(s(0))} = \mathtt{s}^2(\mathtt{0})\ \mathtt{dd(0)} = \mathtt{0},\ \mathtt{winface}(\mathtt{s}^2(\mathtt{0})) = \mathtt{s}^2(\mathtt{0}),\ \mathtt{winface(s(0))} = \mathtt{s(0)}\ \}.$$

[14] For MDL we use the definition of [86] which works well in practice: $length(e) = 1 + n_v/2 + n_f$, where n_v and n_f are the number of variables and function symbols on the right-hand side of e; here constants are regarded as 0-ary functions.

[15] $CovF(E) = card(\{e \in E \mid \mathcal{R} \vdash e\})/card(E)$.

The analysis for dd and winface determines that dd is overspecialized. The generalization algorithm removes the rule $\mathtt{dd(s(x))} \to \mathtt{dd(x)}$. Note that rule $\mathtt{dd(s(0))} \to \mathtt{s}^2\mathtt{(0)}$ inversely narrows to rule $r_{dd} \equiv \mathtt{dd(s(0))} \to \mathtt{s}^2\mathtt{(dd(0))}$ by using rule $\mathtt{dd(0)} \to \mathtt{0}$. The following restricted generalizations of rule r_{dd} are computed: $\mathtt{dd(s(0))} \to \mathtt{s}^2\mathtt{(dd(0))}$, $\mathtt{dd(s(x))} \to \mathtt{s}^2\mathtt{(dd(0))}$, $\mathtt{dd(s(x))} \to \mathtt{s}^2\mathtt{(dd(x))}$. Now, when the third rule is added to the program, all the examples in E^p are covered. In other words, the program

$$\mathcal{R}' \equiv \mathcal{R} \cup \{\mathtt{dd(s(x))} \to \mathtt{s}^2\mathtt{(dd(x))}\} \setminus \{\underline{\mathtt{dd(s(x))} \to \mathtt{dd(x)}}\}$$

is overly general w.r.t. E^p. Thus, the top-down corrector can be applied to $\mathcal{R}'$ to repair the remaining wrong rules:

$$\{\underline{\mathtt{winface(s(x))} \to \mathtt{s(winface(x))}}, \underline{\mathtt{winface(0)} \to \mathtt{0}}\}.$$

8. Implementation

The basic methodology presented so far has been implemented in the prototypical system Buggy [8,10], which is written in SICStus Prolog and available at

http://users.dsic.upv.es/grupos/elp/buggy/buggy.html.

The complete implementation consists of about 600 clauses (2500 lines of code). The system description can be found in [10]. Here, we just point out the main issues involved in such an endeavor.

Buggy includes a (conditional) functional logic language which supports leftmost innermost narrowing [71], (innermost) basic narrowing [89] (a better strategy which does not require functions to be completely defined [37,89]), leftmost outermost narrowing [60] and needed narrowing [22]. The module that computes the abstract program is an improvement of the implementation reported in [8], which iteratively recomputes 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. In order to debug programs under the needed narrowing strategy, we follow the transformational approach based on [84]. That is, functional nestings on the left-hand sides of the (inductively sequential) rules are removed by replacing them by a "case distinction" on the right-hand side of the rules (see Appendix A). Then, the translated program is executed by using the leftmost outermost strategy, which is strongly equivalent (i.e. equivalent w.r.t. computed answers) to needed narrowing on the translated programs, as mentioned in Section 2.

Once bugs have been detected, the user can choose either to indicate the corrections to be made on the wrong rules manually or to automatically repair the program by using the correction methodology of Section 7. In the latter case, the positive and negative example sets are produced according to a parameter which is provided by the user to improve example generation. More specifically, our debugging system requires the user to enter a list of ground constructors which are used to instantiate the non-ground equations belonging to the under- and over-approximation in order to increase the number of ground examples. In general, the larger the list of ground constructors, the bigger the cardinality of the example sets.

Then, an automatic repair is obtained by running an implementation of the top-down correction method based on example-guided unfolding (see Section 7.2). In the case when wrong programs are not overly general, a pure top-down technique is not applicable. Therefore, our correction methodology initially generates a set of positive examples which are not covered by the wrong program. Then, such examples are passed to the inductive functional logic system FLIP [69] in order to synthesize an overly general program to which our top-down methodology can be directly applied, as explained in Section 7.3. Because our methodology is based on abstract interpretation, it may happen that we end up with the original specification as corrected program. Nevertheless, our experimental evaluation demonstrates that this pathological behavior never shows up in practical examples. The alternative, if no other correction is possible, may be simply to remove the incorrect rules from the original wrong program and deliver the diminished program $\mathcal{R}^-$, which is partially correct—albeit not complete—w.r.t. the intended specification.

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 be considered to be a drawback of the theoretical framework. Of course, the user can reuse pieces of code that have already been checked, and this code is simply trusted. Trusting can be done at the level of functions or modules, either statically (by means of annotations) or dynamically (by means of flags). Assisting the user in the task of (manually) entering the (under- and over)-approximations $\mathcal{I}^-$ and $\mathcal{I}^+$ is a possible extension that has not been implemented yet. We are also trying to define an incremental version of our debugger in which coarse, easy-to-compute approximations of the intended semantics can be made more precise at runtime according to the level of detail we want to obtain. Other improvements that we have already implemented are used to filter out unintended (completeness resp. correctness) errors derived from the fact that one of the programs ($\mathcal{I}$ resp. $\mathcal{R}$) lacks some of the rules defining the auxiliary functions of the other program.

Our prototype is equipped with a graphical front-end (see Fig. 2), which has been designed to simplify the user interaction. To this respect, all the operations supported by our system (program and specification loading, narrowing strategy selection, diagnosis as well as correction options) can be easily accessed via intuitive widgets (panels, menus, buttons, *etc.*). Moreover, the graphical interface provides a smooth integration of the FLIP system with our debugger (e.g. FLIP outcomes can be

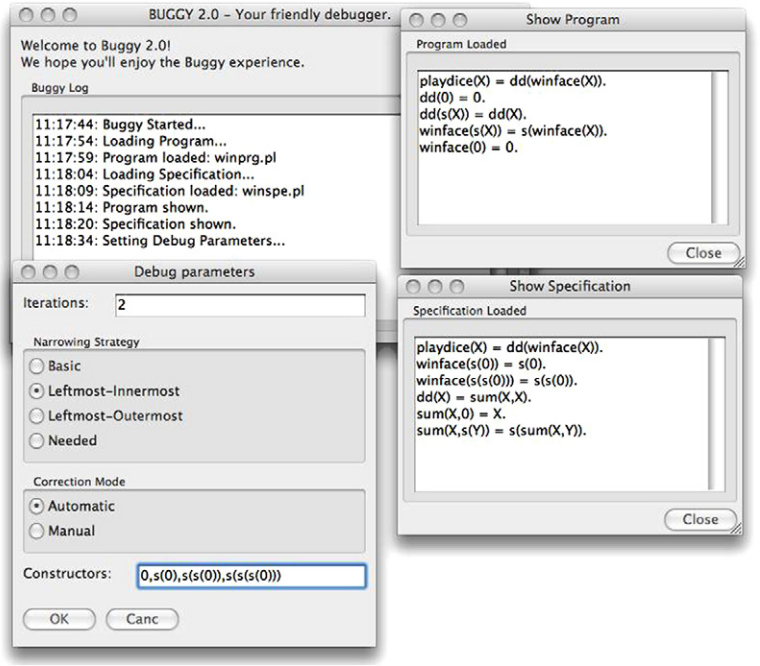

Fig. 2. Program/specification loading and parameter setting.

produced and directly loaded into the debugger). To better understand how our system works and collaborates with FLIP, Section 8.1 presents a complete debugging session of the wrong program of Example 76.

We have tested our debugging and correction methodologies over several benchmarks which are available at http://users.dsic.upv.es/grupos/elp/buggy/buggy.html. In order to systematize the generation of the benchmarks, we have slightly modified correct programs with the aim of obtaining wrong program mutations. Specifically, we have introduced bugs in program rules that affect recursive as well as non-recursive definitions. We were able to successfully diagnose and repair the faulty mutations, achieving, in many cases, a correction both w.r.t. the example sets and the intended program semantics. We have noticed that small example sets generally suffice to get a satisfactory correction. In particular, all experiments required less than 20 positive examples and less than 10 negative examples. Our benchmarks include programs that work with several domains such as natural numbers, lists and finite domains. For instance, we have tested mutations of `append` for the concatenation of two input lists; `last`, which returns the last element of a list; `knapsack`, which returns a set of elements of the input list whose weight sum is equal to an input integer value; `fibonacci`, which computes the Fibonacci numbers; `fact`, which computes the factorial of a positive number; and `sort`, which uses the insertion sort for ordering an input list of integers. By using the intended semantics, we were able to find the errors that were inserted in the program, for all these programs. The final programs have passed the tests of correctness and completeness.

8.1. A debugging session

We show how BUGGY and FLIP can be coupled to diagnose and correct the errors of the program

$$\mathcal{R} = \{\texttt{playdice(x)} \to \texttt{dd(winface(x))}, \texttt{dd(0)} \to \texttt{0}, \texttt{dd(s(x))} \to \texttt{dd(x)},\\ \texttt{winface(s(x))} \to \texttt{s(winface(x))}, \texttt{winface(0)} \to \texttt{0}\}$$

w.r.t. the specification

$$\mathcal{I} = \{\texttt{playdice(x)} \to \texttt{dd(winface(x))}, \texttt{dd(x)} \to \texttt{sum(x, x)},\\ \texttt{sum(x, 0)} \to \texttt{x}, \texttt{sum(x, s(y))} \to \texttt{s(sum(x, y))},\\ \texttt{winface(s(0))} \to \texttt{s(0)}, \texttt{winface(s(s(0)))} \to \texttt{s(s(0))}\}.$$

Overall, the two systems interact in this example as follows. First, the BUGGY system discovers that the program $\mathcal{R}$ to be debugged is not overly general and hence it produces a set of examples which are used by FLIP to generate an overly general version $\mathcal{R}'$ of the wrong program $\mathcal{R}$. Then, BUGGY analyzes $\mathcal{R}'$ and yields the final corrected program w.r.t. the specification $\mathcal{I}$ by applying the example-guided unfolding technique of Section 7.2.

The debugging session starts by invoking the BUGGY system on the specification $\mathcal{I}$ and the program $\mathcal{R}$ (see Fig. 2). The user is required to initialize the following parameters before running the debugger.

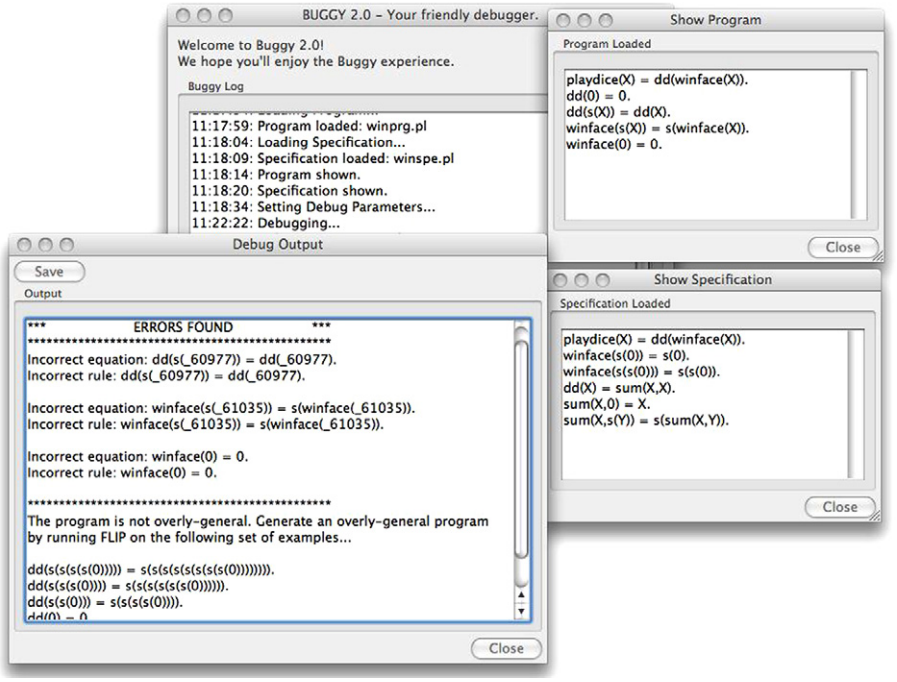

Fig. 3. Debug output for the nonoverly general program $\mathcal{R}$.

Narrowing strategy: it identifies the operational mechanism with which both the program and the specification are evaluated.

Number of iterations: it establishes the number of iterations of the immediate consequence operator which are used to compute the under-approximation of the intended semantics.

List of constructors: it is used to build *ground* (positive as well as negative) examples from non-ground equations generated by the debugger.

Correction flag: it is used to switch from the manual correction to the automatic, unfolding-based correction.

In the considered debugging example, such parameters are set to the values shown in Fig. 2. Fig. 3 illustrates the outcome of the execution of the debugger on the input presented in Fig. 2. In this case, the debugger detects that

- some program rules of $\mathcal{R}$ are wrong w.r.t. the specification $\mathcal{I}$.
- $\mathcal{R}$ is not overly general (specifically, the definition of function dd of program $\mathcal{R}$ is too specific), and hence it generates the input needed by the FLIP system to generalize the program.

Overly specific definitions can be generalized by directly invoking FLIP from our graphical environment on the input data produced by Buggy. In this particular case, we were able to generalize the definition of function dd (see Fig. 4) and produce the following overly general version of the original program $\mathcal{R}$:

$$\begin{aligned}
\texttt{playdice(x)} &\to \texttt{dd(winface(x))}\\
\texttt{dd(s(x))} &\to \texttt{s(s(dd(x)))}\\
\texttt{dd(0)} &\to \texttt{0}\\
\texttt{winface(s(x))} &\to \texttt{s(winface(x))}\\
\texttt{winface(0)} &\to \texttt{0}
\end{aligned}$$

Now, the overly general program can be automatically repaired by using the unfolding-based procedure encoded into Buggy. The final outcome is illustrated in Fig. 5.

It is worth noting that our graphical environment hides several technical details as well as intermediate results which are typically not needed by the final user. However, for a more in-depth analysis of the debugging process, it is possible to inspect the log file produced by our tool. For instance, the log file records the computed under- and over-approximations, and the sets of positive and negative examples generated during the correction phase. Part of the log file referring to the debugging of the overly general version of $\mathcal{R}$ is given in Appendix C.

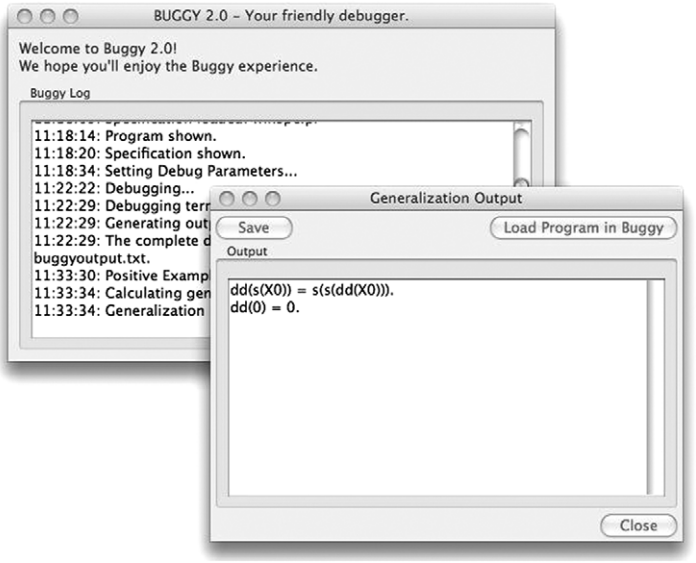

Fig. 4. Generalization of function dd produced by the FLIP system.

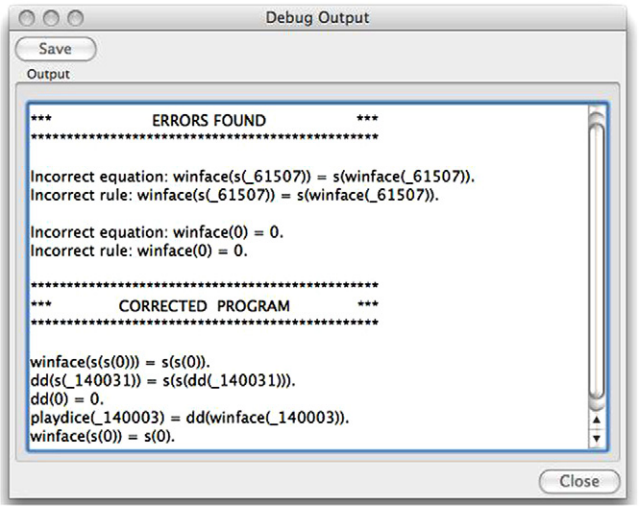

Fig. 5. Correction of the overly general version of program $\mathcal{R}$.

9. Conclusions and future work

The main purpose of this work is to provide a methodology for developing effective debugging and correction tools for functional logic programs or, more generally, for rule-based programs that are executed by narrowing. We have presented a generic debugging scheme that supports program diagnosis w.r.t. the set of computed answers. Our method is based on a fixpoint semantics for functional logic programs that models the set of computed answers in a bottom-up manner and is parametric w.r.t. the narrowing strategy. The proposed 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. As mentioned in Section 4, it is also true that the program specification which is required by our method could be similarly used as an oracle in algorithmic debugging which would become more automatic. Even though, an important advantage of our method is the fact that we develop a finite methodology which is also goal independent, in contrast to algorithmic debugging.

Our approach and the algorithmic debugging approach can be considered complementary. The main advantage of our approach is that once the specification of the correct behavior has been given, the detection of bugs becomes almost fully automatic and can be easily integrated with the correction methodology. The algorithmic debugging approach relies on the existence of an oracle (typically the user) that must interact with the system and drives the debugging diagnosis by

indicating the error symptoms. An advantage is that this does not require a full specification of the program, not even for a specific module. On the other hand, it is less automatic and strongly relies on the users' programming skills. This approach is top-down goal dependent, so it heavily relies on the user inputs and follows the operational semantics. Our approach does not consider any specific goal and provides a finitely terminating dataflow analysis for detection of program bugs. The information collected during the debugging process can then be used as an input to the program corrector. In summary, since the two approaches have different advantages and drawbacks, we could think of a useful integration of the two in declarative programming environments in the future.

9.1. Some final remarks on the debugging framework

The debugging methodology that we consider in this paper can be very useful for a programmer who wants to debug a program w.r.t. a preliminary version which was written with no concern for efficiency. Our technique can also be appreciated by a programmer who wants to better understand the behavior of her program under different semantics. Actually, call-by-value functional programming and lazy functional programming are often considered separate paradigms as it is often not possible to carry programs that are designed in one paradigm over the other (e.g. the evaluation order has critical consequences for the termination behavior of programs) [114]. This is why we consider the support of a form of parametricity w.r.t. the evaluation strategy to be very interesting since it allows us to provide a common framework in which both eager and lazy programs can be developed and program correctness can be checked without burdening the programmer with operational details concerning the evaluation strategy, which is common to traditional program tracing. Moreover, both innermost and outermost narrowing have recently regained much attention as they have proven to be very useful for analysing security properties in rewriting-based languages such as Elan [91] and Maude [103]. These strategies are of great interest in the field of security analysis, because policy application can not only be specified in a precise, clean and expressive way but also dynamically enforced efficiently and effectively, by using strategies that control rule application [91].

However, note that we cannot prove our results in a parametric way for a generic rule-based language with fixpoint semantics and computed answer semantics based on narrowing since most of the results may depend on the considered narrowing strategy and advanced language features. For instance, we have not considered in our basic diagnosis methodology sophisticated language features such as Maude's sorts and algebraic laws. The extension to deal with non-deterministic, non-strict, higher order functional logic languages is also an issue in its own right, which should be dealt with separately, and is beyond the scope of this paper (an extension of the abstract diagnosis methodology to the case of the first order, non-confluent functional logic programs with call-time choice behavior has been recently investigated in [28]).

9.2. Ongoing and future work

We have discussed some successful experiments that have been performed with a prototypical implementation of our debugging system which is publicly available. This is the first step in creating an integrated development environment in which it is possible to diagnose errors in a program, and in the case the program is wrong, to automatically generate the correct program. To the best of our knowledge, this is the first attempt to generally integrate semantics-based program debugging and synthesis for rule-based languages fitted with a narrowing mechanism. However, our method should be viewed as a basis for the development of more sophisticated tools for the debugging of narrowing-based languages, for which there is still much work to be done.

Actually, we do believe that it is possible to derive several optimizations to improve precision as well as scalability of the considered debugging method. As for precision, an example of our ongoing work towards such an endeavour is presented in [12], where we developed an extension of the known untyped generalization algorithm modulo equational axioms. Moreover, we are currently studying new heuristics to drive the generalization process in order to improve the accuracy of the bottom-up synthesis. To keep the performance of the debugging process acceptable even when we consider large programs, we are exploring static analysis [58] as well as program slicing [115] techniques with the aim of isolating the minimal fragment of the wrong code which has to be fixed.

As future work, we are also considering to formalize specifications by means of assertions, that is, logical constraints defining the correct program behavior, following the style of [53]. This approach has two main advantages: on the one hand, it allows (even partial) specifications to be modeled in a very concise way; on the other hand, it might be coupled with an implementation of the inductive framework for the program synthesis of [59], which allows one to infer programs (and hence corrections) from equational axiomatizations. Indeed, we believe that handling assertions simply requires the choice of a more concrete semantics, modeling call patterns in addition to computed answers.

Acknowledgements

We thank thank Michele Baggi, Stefano Marson, and Sergio Lara-Pons for their help in implementing the declarative debugger Buggy. Santiago Escobar and José Iborra provided helpful comments on this paper. We also thank the anonymous referees for many useful suggestions.

Appendix A. The flattening transformations

A.1. Flattening of equations

In this section, we recall the flattening transformations for equational goals of [37,75], which we unify as two cases of a single, generic flattening transformation.

Following [75], 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 an operation-headed term, then it can be obtained as the application of $[\ ]$ (the empty context) to t itself. The following definition is auxiliary.

Definition 77 (*Pre-flattening*)**.** The function $flat^{\varphi}_{-}(s)$ for an expression s is defined inductively as follows:

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

Example 79 below illustrates the different outputs of the flattening transformation depending on φ, which (recursively) appear 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 that result from the transformation above are trivial and do not contribute to the semantics of our framework. Hence, they can simply be removed after applying the most general unifier to the remaining equations. We formalize this idea as follows.

An equation $x = y$, with $x, y \in V$ is called a *trivial equation*. Note that equations $x \approx y$ are not trivial. Given a set of equations g, we define $split(g) = (g_1, g_2)$ as the function that 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 78 (*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 that 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 79. Consider the function `from` of Example 16. Let $g \equiv$ (`first([first([0])]) = first([0, 1])`) and $\varphi = inn$. Then, the pre-flattening of g is $g_0 \equiv flat^{inn}_{-}(g) =$ (`first([0]) = y, first([y]) = w, first([0, 1]) = z, w = z`), and $split(g_0) \equiv$ ({`first([0]) = y, first([y]) = w, first([0, 1]) = z`}, {`w = z`}). Hence,

$$flat_{inn}(g) \equiv \texttt{(first([0]) = y, first([y]) = w, first([0, 1]) = w)}.$$

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

Note that the equality symbols $=$ and $\approx$ may appear simultaneously, but are handled differently since equations $x \approx y$ are not trivial.

A.2. A transformation preserving needed narrowing computations

Needed narrowing [22] is a complete lazy narrowing strategy that is optimal w.r.t. the length of the derivations and the number of computed solutions in inductively sequential (IS) programs, that is, programs in which all their 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. For a precise definition of this class of programs and the needed narrowing strategy based on the notion of a definitional tree, see [20].

Needed narrowing can be easily and efficiently implemented by translating definitional trees into "case expressions" as proposed in [84], which also proves that there is a strong equivalence (i.e. equivalence w.r.t. computed answers) of needed narrowing derivations in the original program and leftmost-outermost narrowing derivations in the transformed program. A similar transformation is presented in [122], where inductively sequential programs are translated to the *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. Uniform programs have been further studied in [13].

The following example illustrates the transformation into case expressions of [84]. Roughly speaking, each inductively sequential function f is transformed into a new function (which is also called f) 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 80. Consider the following inductively sequential program for addition and doubling of natural numbers in Peano's notation:

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

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

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

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

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

Note that different *case* functions (which we distinguish by using different subindices) are needed for case expressions with different patterns. Also, note that the transformation produces rewrite rules with extra variables on their right-hand sides. 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 on the right-hand sides of the rules). Hence, via this transformation, we do not 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. There is only a slight detail here: Hanus & Prehofer do not consider conditional rules explicitly but this is not a substantial characterization for our discussion (it can be amended through the usual encoding of conditions by means of the predefined function "if" [21]).

Appendix B. Proofs

B.1. Proofs of Section 3

Let us first recall some basic definitions which are instrumental for the proofs.

A *pre-fixpoint* of $T^{\varphi}_{\mathcal{R}}$ is any $\mathcal{I}$ such that $T^{\varphi}_{\mathcal{R}}(\mathcal{I}) \subseteq \mathcal{I}$. If $T^{\varphi}_{\mathcal{R}}(\mathcal{I}) = \mathcal{I}$, then $\mathcal{I}$ is called a *fixpoint* of $T^{\varphi}_{\mathcal{R}}$.

Definition 81. We now define the *powers* of operator $T^{\varphi}_{\mathcal{R}}$ as follows:

$$\begin{array}{lll} T^{\varphi}_{\mathcal{R}} \uparrow 0(\mathcal{I}) & = \mathcal{I}, \\ T^{\varphi}_{\mathcal{R}} \uparrow (n+1)(\mathcal{I}) & = T^{\varphi}_{\mathcal{R}}(T^{\varphi}_{\mathcal{R}} \uparrow n(\mathcal{I})), \\ T^{\varphi}_{\mathcal{R}} \uparrow \omega(\mathcal{I}) & = \bigcup_{n=0}^{\infty} T^{\varphi}_{\mathcal{R}} \uparrow n(\mathcal{I}). \end{array}$$

Informally, $T^{\varphi}_{\mathcal{R}} \uparrow n(\mathcal{I})$ is the result of the n-fold iteration of $T^{\varphi}_{\mathcal{R}}$ starting at $\mathcal{I}$. Note that we abbreviate $T^{\varphi}_{\mathcal{R}} \uparrow n = T^{\varphi}_{\mathcal{R}} \uparrow n(\emptyset)$.

The following result is necessary to demonstrate the continuity of $T^{\varphi}_{\mathcal{R}}$.

Lemma 82. *Let $\mathcal{I}_1 \subseteq \mathcal{I}_2 \subseteq \cdots$ be any infinite sequence of equation sets and let $\mathcal{C}$ be a finite set of equations. Then,*

$$C \subseteq \left(\bigcup_{n=1}^{\infty} \mathcal{I}_n\right)^{\mathcal{C}} \quad \textit{iff} \quad (\exists i.i \geq 1)(C \subseteq (\mathcal{I}_i)^{\mathcal{C}}).$$

Proof. Let $\mathcal{I}_1 \subseteq \mathcal{I}_2 \subseteq \ldots$ be an infinite sequence of equation sets and let $C \subseteq (\cup_{n=1}^{\infty} \mathcal{I}_n)^{\mathcal{C}}$. Since C is a finite set of equations, then $C = \{e_1, e_2, \ldots, e_n\}$, where the right-hand side of each e_i is a constructor term. For all i, $1 \leq i \leq n$, $e_i \in (\cup_{n=1}^{\infty} \mathcal{I}_n)$ since $(\cup_{n=1}^{\infty} \mathcal{I}_n)^{\mathcal{C}} \subseteq (\cup_{n=1}^{\infty} \mathcal{I}_n)$. Hence, there exits $j_i \geq 1$ such that $e_i \in \mathcal{I}_{j_i}$. Let $k = max\{j_1, j_2, \ldots, j_n\}$ and since $\mathcal{I}_1 \subseteq \mathcal{I}_2 \subseteq \ldots$, then $\mathcal{I}_{j_i} \subseteq \mathcal{I}_k$ for $1 \leq i \leq n$; therefore, $\{e_1, e_2, \ldots, e_n\} \subseteq \mathcal{I}_k$, i.e., $C \subseteq \mathcal{I}_k$. Finally, by the condition that the right-hand side of each equation of C is a constructor term, we conclude that $C \subseteq (\mathcal{I}_k)^{\mathcal{C}}$.

Conversely, assume there exists $k \geq 1$ such that $C \subseteq (\mathcal{I}_k)^{\mathcal{C}}$; hence, the right-hand sides of the equations of C are constructor terms. Since $(\mathcal{I}_k)^{\mathcal{C}} \subseteq \mathcal{I}_k$, then $C \subseteq \mathcal{I}_k$. Therefore, $C \subseteq \cup_{n=1}^{\infty} \mathcal{I}_n$, and the desired result follows. □

Proposition 9. *The operator $T_{\mathcal{R}}^{\varphi}$ is continuous on the lattice of Herbrand interpretations, $\varphi \in \{inn, out\}$. The least fixpoint of $T_{\mathcal{R}}^{\varphi}$ is $lfp(T_{\mathcal{R}}^{\varphi}) = T_{\mathcal{R}}^{\varphi} \uparrow \omega$.*

Proof. Let us prove that, for any infinite sequence $I_1 \subseteq I_2 \subseteq \ldots$

$$T_{\mathcal{R}}^{\varphi}(\cup_{n=1}^{\infty} \mathcal{I}_n) = \cup_{n=1}^{\infty} T_{\mathcal{R}}^{\varphi}(\mathcal{I}_n).$$

If $e \in T_{\mathcal{R}}^{\varphi}(\cup_{n=1}^{\infty} \mathcal{I}_n)$ by definition of $T_{\mathcal{R}}^{\varphi}$ is equivalent to

$$e \in \Phi_{\mathcal{R}} \cup \Im_{\mathcal{R}}^{\varphi} \cup \{e' \in \mathcal{B}_V \mid (\lambda = \rho \Leftarrow C) \ll \mathcal{R}_{++}^{\varphi},\ \{l = r\} \subseteq \cup_{n=1}^{\infty} \mathcal{I}_n,\ C' \subseteq (\cup_{n=1}^{\infty} \mathcal{I}_n)^{\mathcal{C}},\ mgu(flat_{\varphi}(C), C') = \sigma,\ mgu(\{\lambda = r_{|u}\}\sigma) = \theta,\ u \in \varphi(r),\ e' = (l = r[\rho]_u)\sigma\theta\}.$$

iff there exists i, $1 \leq i \leq n$ such that, by definition of $\cup_{n=1}^{\infty} \mathcal{I}_n$, and by Lemma 82, there exists j, $1 \leq j \leq n$ such that

$$e \in \Phi_{\mathcal{R}} \cup \Im_{\mathcal{R}}^{\varphi} \cup \{e' \in \mathcal{B}_V \mid (\lambda = \rho \Leftarrow C) \ll \mathcal{R}_{++}^{\varphi},\ \{l = r\} \subseteq \mathcal{I}_i,\ C' \subseteq (\mathcal{I}_j)^{\mathcal{C}},\ mgu(flat_{\varphi}(C), C') = \sigma,\ mgu(\{\lambda = r_{|u}\}\sigma) = \theta,\ u \in \varphi(r),\ e' = (l = r[\rho]_u)\sigma\theta\}.$$

iff for all k, $k \geq i$, $k \geq j$

$$e \in \Phi_{\mathcal{R}} \cup \Im_{\mathcal{R}}^{\varphi} \cup \{e' \in \mathcal{B}_V \mid (\lambda = \rho \Leftarrow C) \ll \mathcal{R}_{++}^{\varphi},\ \{l = r\} \subseteq \mathcal{I}_k,\ C' \subseteq (\mathcal{I}_k)^{\mathcal{C}},\ mgu(flat_{\varphi}(C), C') = \sigma,\ mgu(\{\lambda = r_{|u}\}\sigma) = \theta,\ u \in \varphi(r),\ e' = (l = r[\rho]_u)\sigma\theta\}.$$

iff there exists k, $1 \leq k \leq n$, such that $e \in T_{\mathcal{R}}^{\varphi}(\mathcal{I}_k)$.
iff $e \in \cup_{n=1}^{\infty} T_{\mathcal{R}}^{\varphi}(\mathcal{I}_n)$.

Thus, since $T_{\mathcal{R}}^{\varphi}$ is continuous, by Kleene's theorem there exists the least fixpoint $lfp(T_{\mathcal{R}}^{\varphi}) = T_{\mathcal{R}}^{\varphi} \uparrow \omega$. □

Lemma 83. *Let $\mathcal{R} \in \mathbb{R}_{\varphi}$. If $l =_{\varphi} t \in T_{\mathcal{R}}^{\varphi} \uparrow k$, then l is a pattern or l is a constructor term.*

Proof. The proof is by induction on the number k of iterations.

If $k = 1$ and $l =_{\varphi} r \in T_{\mathcal{R}}^{\varphi} \uparrow 1$, then the equation $l =_{\varphi} r$ is a reflexive axiom and the proof is done.

Let us consider the inductive case. Let $l =_{\varphi} r \in T_{\mathcal{R}}^{\varphi} \uparrow k$ which generates the new equation $l' =_{\varphi} r' \in T_{\mathcal{R}}^{\varphi} \uparrow (k+1)$ and $l =_{\varphi} r$ fulfills the inductive hypothesis (hence l is a pattern). Then, by definition of $T_{\mathcal{R}}^{\varphi} \uparrow (k+1)$, there exists a set of equations $l = r \in T_{\mathcal{R}}^{\varphi} \uparrow k$, $C' \subseteq (T_{\mathcal{R}}^{\varphi} \uparrow k)^{\mathcal{C}}$ and a rule $(\lambda \to \rho \Leftarrow C) \ll \mathcal{R}_{++}^{\varphi}$ such that there exist $\sigma = mgu(flat_{\varphi}(C), C')$, $\beta = mgu(\{\lambda = r_{|u}\}\sigma)$ and $u \in \varphi(r)$ which satisfy $(l' = r') \equiv ((l = r[\rho]_u)\sigma\beta) \in T_{\mathcal{R}}^{\varphi} \uparrow (k+1)$.

By definition of $\mathcal{I}^{\mathcal{C}}$ and the inductive hypothesis, the equations of C' have the form $d'_l =_{\varphi} d'_p$ or $f'(d'_1, \ldots, d'_n) = d'_{n+1}$ where $d'_1, \ldots, d'_n, d'_{n+1}, d'_l, d'_p$ are constructor terms. Similarly, by Definition 78 (flattening), the equations of $flat_{\varphi}(C)$ have the form $f(d_1, \ldots, d_n) = x$ or $d_l =_{\varphi} y$, where $d_1, \ldots, d_n, d_l$ are constructor terms and x, y are variables. If $\varphi = inn$, then the equation $d_l = y$ is non-trivial, since d_l is not variable. If $\varphi = out$, then the equation $d_l \approx y$ only unifies with an equation of the form $s \approx t$ where s, t are constructor terms. Now, if $d'_l =_{\varphi} d'_p \in (T_{\mathcal{R}}^{\varphi} \uparrow k)^{\mathcal{C}}$ then, by definition of $T_{\mathcal{R}}^{\varphi} \uparrow i$, d'_l is not variable. Therefore, we conclude that σ is a constructor substitution. Now, we consider the cases for $\varphi = inn$, out separately.

(i) $\varphi = inn$. If $l = r \in T_{\mathcal{R}}^{inn} \uparrow k$ and $u \in inn(r)$, then $r|_u$ is the innermost redex of r, which unifies with the left-hand side of a program rule (which is also a pattern because $\mathcal{R}$ is CB). Therefore, β is a constructor substitution and hence $l' = l\sigma\beta$ is a pattern.

(ii) $\varphi = out$. If $l = r \in T_{\mathcal{R}}^{out} \uparrow k$ and $u \in out(r)$, then $r|_u$ is the outermost redex of r, which unifies with the left-hand side of a program rule (which is a linear pattern because $\mathcal{R}$ is CB and left linear). Therefore, $\beta_{|Var(r)}$ is a constructor substitution and hence $l' = l\sigma\beta$ is a pattern as well. □

Lemma 84 (*[18]*). *Let $\mathcal{R} \in \mathbb{R}_{\varphi}$ and g_1, g_2 be a goal.*

$(g_1, g_2) \overset{\theta}{\leadsto}{}^*_{\varphi} \top$ *iff* $g_1 \overset{\sigma_1}{\leadsto}{}^*_{\varphi} \top$, $g_2 \overset{\sigma_2}{\leadsto}{}^*_{\varphi} \top$ *and* $\theta = \sigma_1 \Uparrow \sigma_2$, *for* $\varphi \in \{inn, out\}$.

Lemma 85. *Let g be a set of equations. $g \overset{\theta}{\leadsto}{}^*_{\varphi} \top$ iff $flat_{\varphi}(g) \overset{\theta'}{\leadsto}{}^*_{\varphi} \top$ and $\theta = \theta'_{|Var(g)}$, for $\varphi = \{inn, out\}$.*

Proof (*Sketch*). The proof for the case $\varphi = inn$ can be found in [19]. The key idea for the proof is the following property of *non-productive substitutions*: for all $(g_1, g_2) \overset{\theta}{\leadsto}{}^*_{\varphi}(g'_1, g_2\theta)$ such that no position of g_2 has been reduced by narrowing, $\varphi(g_2\theta) \subseteq \varphi(g_2)$. This property trivially holds for $\varphi = inn, out$ in programs of $\mathcal{R} \in \mathbb{R}^{\varphi}$, since in this class both strategies compute only constructor substitutions. Thus, for the case when $\varphi = out$, the proof is perfectly analogous. □

In the following, the notation $g \leadsto_\varphi^n g'$ is used to represent a narrowing derivation from g to g' which respects φ and whose length is n.

Lemma 86. *Let $\mathcal{R} \in \mathbb{R}_\varphi$, $\varphi \in \{inn, out\}$, f be a defined function of $\mathcal{R}$ and $\bar{x}$ a tuple of pairwise distinct variables. Then, for each $f(\bar{x})\theta = t \notin T_\mathcal{R}^\varphi \uparrow (k-1)$, we have*

$$(f(\bar{x})\theta = t) \in T_\mathcal{R}^\varphi \uparrow k \quad \textit{iff} \quad f(\bar{x}) = y \overset{\theta,m}{\leadsto_\varphi} t = y$$

where $\perp$ does not occur in t and $m \geq k$.

Proof. ($\Longrightarrow$): For all $(f(\bar{x})\theta = t) \in T_\mathcal{R}^\varphi \uparrow k$, such that $\perp$ does not occur in t, we show that there exists a narrowing derivation of length $m \geq k$ in $\mathcal{R}$ such that $f(\bar{x}) = y \overset{\theta,m}{\leadsto_\varphi} t = y$. The proof is by induction on the number k of iterations.

If $k = 1$, then $(f(\bar{x}) = f(\bar{x})) \in T_\mathcal{R}^\varphi \uparrow 1$. Thus, for the reflexive closure of the narrowing relation, we get $f(\bar{x}) = y \overset{\epsilon,1}{\leadsto_\varphi} f(\bar{x}) = y$.

Let us consider the inductive case. We assume that the lemma holds for k and prove that it holds for $k+1$. Consider the equations in $T_\mathcal{R}^\varphi \uparrow k$ which generate new equations in $T_\mathcal{R}^\varphi \uparrow (k+1)$.

Let $(f(\bar{x})\theta = t_k) \in T_\mathcal{R}^\varphi \uparrow k$ be one of such equations, which generates an equation such that $(f(\bar{x})\theta' = t_{(k+1)}) \in T_\mathcal{R}^\varphi \uparrow (k+1)$ and $(f(\bar{x})\theta' = t_{(k+1)}) \notin T_\mathcal{R}^\varphi \uparrow k$. Then, by definition of $T_\mathcal{R}^\varphi \uparrow (k+1)$, there exists a set of equations $\{f(\bar{x})\theta = t_k\} \cup C'$ with $\{f(\bar{x})\theta = t_k\} \subseteq T_\mathcal{R}^\varphi \uparrow k$, $C' \subseteq (T_\mathcal{R}^\varphi \uparrow k)^c$ and a rule $(\lambda \to \rho \Leftarrow C) \ll \mathcal{R}_{++}^\varphi$ such that there exist $\sigma = mgu(flat_\varphi(C), C')$, $\beta = mgu(\{\lambda = t_k|_u\}\sigma)$ and $u \in \varphi(t_k)$ which satisfy $(f(\bar{x})\theta' = t_{k+1}) \equiv ((f(\bar{x})\theta = t_k[\rho]_u)\sigma\beta)$ where $\theta' = \theta\sigma\beta$.

Since rule $(\lambda \to \rho \Leftarrow C)$ is a variant of a rule in $\mathcal{R}_{++}^\varphi$ (so it contains only new variable symbols) and σ depends on the variables in $flat_\varphi(C)$ and C', it follows that $Var(\sigma) \cap Var(t_k) = \emptyset$, then $t_k = t_k\sigma$ from which it derives that $\beta = mgu(\{\lambda = t_k|_u\}\sigma) = mgu(\lambda\sigma = t_k|_u)$ and $t_{k+1} = (t_k[\rho]_u)\sigma\beta = (t_k[\rho\sigma]_u)\beta$.

Since $(f(\bar{x})\theta = t_k) \in T_\mathcal{R}^\varphi \uparrow k$, by the inductive hypothesis, there exists a narrowing derivation following strategy φ such that $f(\bar{x}) = y \overset{\theta,m}{\leadsto_\varphi} t_k = y$ with $m \geq k$. Let us consider the subsequent narrowing step. Then, there exist $u \in \varphi(t_k)$ i.e., $u \in \varphi(t_k\sigma)$ and a rule $r \equiv (\lambda\sigma \to \rho\sigma \Leftarrow C\sigma) \ll \mathcal{R}_{++}^\varphi$ s.t. $mgu(\lambda\sigma = t_k|_u\sigma) = \beta$. By hypothesis, $\perp$ does not occur in t_k, t_{k+1}, hence $r \ll \mathcal{R}_+^\varphi$ and thus a narrowing step which uses strategy φ can be proven and derives the goal $C\sigma\beta$, $(t_{k+1} = y)$, with $t_{k+1} = (t_k[\rho]_u)\sigma\beta$.

$$f(\bar{x}) = y \overset{\theta,m}{\leadsto_\varphi} t_k = y \overset{\sigma\beta,m+1}{\leadsto_\varphi} C\sigma\beta, (t_{k+1} = y).$$

Since $C' \subseteq (T_\mathcal{R}^\varphi \uparrow k)^c \subseteq (T_\mathcal{R}^\varphi \uparrow k)$ and the left-hand sides of equations in $T_\mathcal{R}^\varphi \uparrow k$ are patterns or constructor terms (Lemma 83), then, by the inductive hypothesis, we have that, for all $f_s(\bar{d}) = d_s$ (resp. $d_m =_\varphi d'_m$) of C', there exists a derivation of length n such that $f_s(\bar{x}) = y \overset{\theta,n}{\leadsto_\varphi} d_s = y \overset{\{y/d_s\}}{\leadsto_\varphi} \top$ (resp. $d_m =_\varphi d'_m \overset{\epsilon,*}{\leadsto_\varphi} \top$). In general, for every equation $e \in C'\sigma$, we derive that $e \overset{\epsilon,*}{\leadsto_\varphi} \top$ (since σ is constructor), that is, $C'\sigma \overset{\epsilon,*}{\leadsto_\varphi} \top$, and then $flat_\varphi(C)\sigma \overset{\epsilon,*}{\leadsto_\varphi} \top$. Since $flat_\varphi(C)$ preserves the computed answers under the narrowing strategy φ, by Lemma 85, then $C\sigma \overset{\epsilon,*}{\leadsto_\varphi} \top$. Since narrowing strategy φ is correct and complete w.r.t. the computed answers and $\beta_{|Var(t_k)}$ is a constructor substitution, then $C\sigma\beta \overset{\epsilon,*}{\leadsto_\varphi} \top$. In conclusion,

$$f(\bar{x}) = y \overset{\theta,m}{\leadsto_\varphi} t_k = y \overset{\sigma\beta,m+1}{\leadsto_\varphi} C\sigma\beta, (t_{k+1} = y) \overset{\epsilon,*}{\leadsto_\varphi} . t_{k+1} = y.$$

We conclude that there exists a derivation of length $m' \geq k$ for narrowing strategy φ w.r.t. $\mathcal{R}$ such that $f(\bar{x}) = y \overset{\theta',m'}{\leadsto_\varphi} t = y$ where $\theta' = \theta\sigma\beta$.

($\Longleftarrow$): For the opposite direction, we prove the more general result that, for all derivations of the form $\mathcal{D} : f(\bar{x}) = y \overset{\theta,n}{\leadsto_\varphi} C,\ t = y$ s.t. $C \overset{\beta,m}{\leadsto_\varphi} \top$, there exists $1 \leq k \leq n+m$ s.t. $(f(\bar{x})\theta = t)\beta \in T_\mathcal{R}^\varphi \uparrow k$.

The proof is by induction on the length n of $\mathcal{D}$.

Let $n = 1$, then by the reflexive closure of the narrowing relation $f(\bar{x}) = y \overset{\epsilon}{\leadsto_\varphi} f(\bar{x}) = y$ and by Definition 8 $(f(\bar{x}) = f(\bar{x})) \in T_\mathcal{R}^\varphi \uparrow 1$.

Let us consider that the statement of the lemma holds for all derivations of length $m < n$ and then prove that it holds for n.

We consider a derivation of length n, $\mathcal{D} : f(\bar{x}) = y \overset{\theta,n}{\leadsto_\varphi} C,\ t = y$ such that $C \overset{\beta,m}{\leadsto_\varphi} \top$.

For $\varphi \in \{inn, out\}$, it is immediate to see that $\mathcal{D}$ has one of the following two forms.

The first form is

$$f(\bar{x}) = y \overset{\theta',n-1}{\leadsto_\varphi} t' = y \overset{\beta}{\leadsto_\varphi} t = y,$$

where every rule condition eventually introduced within the derivation is solved before the final narrowing step $t' = y \overset{\beta}{\leadsto_\varphi} t = y$. In this case, by the inductive hypothesis there exists $1 \leq j \leq n-1$ s.t. $f(\bar{x})\theta' = t' \in T_\mathcal{R}^\varphi \uparrow j$, and hence it follows trivially from the definition of strategy φ of narrowing and $T_\mathcal{R}^\varphi$ that $f(\bar{x})\theta'\beta = t \in T_\mathcal{R}^\varphi \uparrow j+1$, and the conclusion follows.

The second possible form is

$$f(\bar{x}) = y \overset{\theta',n-1}{\leadsto_\varphi} C', t' = y \overset{\alpha}{\leadsto_\varphi} C, t = y \overset{\beta,m}{\leadsto_\varphi} (t\beta = y),$$

where the final subderivation $C, t = y \overset{\beta,m}{\leadsto}_\varphi (t\beta = y)$ reduces to $\top$ the condition C without narrowing the term t, and the step $C', t' = y \overset{\alpha}{\leadsto}_\varphi C, t = y$ narrows t' to the term t by applying a rule $(\lambda \to \rho \Leftarrow B) \ll \mathcal{R}^\varphi_+$ to the selected position $u \in \varphi(t')$ s.t. $\alpha = mgu(\{\lambda = t'|_u\})$ and $t = (t'[\rho]_u)\alpha$, with $C = (C', B)\alpha$.

In the following, we will apply structural analysis over $\mathcal{D}$ for concluding that this derivation produces an equation in $T^\varphi_\mathcal{R} \uparrow k$ for any k.

Since $C, t = y \overset{\beta,m}{\leadsto}_\varphi (t\beta = y)$, we have that $C \overset{\beta,m}{\leadsto}_\varphi \top$ and, by Lemma 85 $flat_\varphi(C) \overset{\beta' *}{\leadsto}_\varphi \top$, with $\beta = \beta'_{|Var(C)}$, hence $flat_\varphi(C)\beta' \overset{\epsilon *}{\leadsto}_\varphi \top$. The flat equations in $flat_\varphi(C)$ have the form $f'(x_1, \ldots, x_n)\delta = x$ or $d_l =_\varphi y$, where $\delta = \{x_1/d_1, \ldots, x_n/d_n\}$ and $d_l, d_i, i = 1 \ldots n$, are constructor terms and x, y are variables.

For each equation $e \equiv (d_l =_\varphi y)\beta'$ in $flat_\varphi(C)\beta'$, by the definition of $T^\varphi_\mathcal{R}$, we will prove that there exist $m_e \geq 1$ and an equation $e'' \in T^\varphi_\mathcal{R} \uparrow m_i$ such that $e''\gamma \equiv e\beta'$, where γ is a constructor substitution. In fact, in the case when $\varphi = inn$, by definition of $T^{inn}_\mathcal{R}$, for $c/n \in \mathcal{C}$ we know that the equations $c(x_1, \ldots, x_n) = c(x_1, \ldots, x_n) \in T^{inn}_\mathcal{R} \uparrow 1$, and the claim follows. In the case when $\varphi = out$, if $(d_l \approx y)\beta' \overset{\epsilon *}{\leadsto}_{out} \top$, then $(d_l \approx y)\beta'$ is ground since the rules which define the strict equality $\approx$ are applied. Now, by definition of $T^{out}_\mathcal{R}$, there exist $m_e \geq 0$ such that $(d_l \approx y)\beta' \in T^{out}_\mathcal{R} \uparrow m_e$.

Now, we consider $e' \equiv (f'(x_1, \ldots, x_n)\delta = x) \in flat_\varphi(C)$ and $f'(x_1, \ldots, x_n)\delta = x \overset{\beta'}{\leadsto}_\varphi \top$. Then, $f'(x_1, \ldots, x_n) = y \overset{\delta\beta'}{\leadsto}_\varphi x\beta = y$. By hypothesis, we conclude that there exists $m_{e'}$ such that $(f'(x_1, \ldots, x_n)\delta = x)\beta' \in T^\varphi_\mathcal{R} \uparrow m_{e'}$.

In general, let us construct the set $C'' = \{e_1, \ldots, e_p\}$ such that $C'' \subseteq T^\varphi_\mathcal{R} \uparrow m$, where m is the maximum of the m_e's, for all equations e in $flat_\varphi(C)$ such that $C''\vartheta = flat_\varphi(C)\beta'$. Note that, since β', ϑ are constructor substitutions without common variables, then C'' is a flat set of equations. Then, $C'' \subseteq (T^\varphi_\mathcal{R} \uparrow m)^c$. and $mgu(flat(C), C'') = \vartheta\beta'$. Moreover, $\beta = \beta'_{|Var(C)} = (\vartheta\beta')_{|Var(C)}$.

Finally, consider the prefix $f(\bar{x}) = y \overset{\theta', n-1}{\leadsto}_\varphi C', t' = y$ of $\mathcal{D}$. By the inductive hypothesis, there exists $q \geq 0$ s.t. $\{f(\bar{x})\theta'\alpha\beta = t'\alpha\beta\} \subseteq T^\varphi_\mathcal{R} \uparrow q$.

In conclusion, we have proven that there exists $p, 1 \leq p \leq q+m$, such that $\{f(\bar{x})\theta'\alpha\beta = t'\alpha\beta\} \subseteq T^\varphi_\mathcal{R} \uparrow p, C'' \subseteq (T^\varphi_\mathcal{R} \uparrow p)^c$, and $\beta = mgu(flat_\varphi(C), C'')_{|Var(C)}$. Therefore, since there exist $u \in \varphi(t')$ and a rule $(\lambda \to \rho \Leftarrow B) \ll \mathcal{R}^\varphi_+$ (hence $(\lambda \to \rho \Leftarrow B) \ll \mathcal{R}^\varphi_{++}$) with $mgu(flat_\varphi(B), C'') = \beta_{|Var(B)}$, $\alpha = mgu(\{\lambda = t'|_u\})$ and $t = (t'[\rho]_u)\alpha$, we conclude that $(f(\bar{x})\theta = t\beta) \in T^\varphi_\mathcal{R} \uparrow k$, with $k = p + 1$ and $\theta = \theta'\alpha\beta$. □

Theorem 15 (*Strong soundness and completeness*). *Let $\mathcal{R} \in \mathbb{R}_\varphi$ 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 g is closed by $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ with substitution θ.*

Proof. ($\Rightarrow$): Let g be a goal such that $flat_\varphi(g) = (e_1, \ldots, e_n)$ and $g \overset{\theta *}{\leadsto}_\varphi \top$. Using Lemma 85, it follows that $(e_1, \ldots, e_n) \overset{\theta' *}{\leadsto}_\varphi \top$ and $\theta_{|g} = \theta'_{|g}$. By Lemma 84 $e_1 \overset{\sigma_1 *}{\leadsto}_\varphi \top, \ldots, e_n \overset{\sigma_n *}{\leadsto}_\varphi \top$ such that $\theta' = \sigma_1 \Uparrow \ldots \Uparrow \sigma_n$.

Since $e_i, i = 1, \ldots, n$ are flat equations, then they have the following form $f_s(d^i_1, \ldots, d^i_n) = x_i$ or $d^i_l =_\varphi y_i$. We consider the two cases separately:

If $e_i \equiv f_s(d^i_1, \ldots, d^i_n) = x_i$, then we can write it as $f_s(\bar{x})\beta_i = x_i$ where $\beta_i = \{x_1/d^i_1, \ldots, x_n/d^i_n\}$. Now, for every i, since $f_s(\bar{x})\beta_i = x_i \overset{\sigma_i *}{\leadsto}_\varphi \top$, then $f_s(\bar{x})\beta_i\sigma_i = x_i\sigma_i \overset{\epsilon *}{\leadsto}_\varphi \top$ i.e. $f_s(\bar{x}) = y \overset{\beta_i\sigma_i *}{\leadsto}_\varphi x_i\sigma_i = y$ where $x_i\sigma_i$ is a constructor term not including $\bot$. By Lemma 86 and the definition of $\mathcal{F}^{ca}_\varphi(\mathcal{R})$, we have $f_s(\bar{x})\beta_i\sigma_i = x\sigma_i \in \mathcal{F}^{ca}_\varphi(\mathcal{R})$. Let $e'_i \equiv f_s(\bar{x})\beta_i\sigma_i = x\sigma_i$, then we conclude that $e'_i = e_i\sigma_i$ and therefore e_i is closed by $\mathcal{F}^{ca}_\varphi(\mathcal{R})$.

If $e_i \equiv d^i_l =_\varphi y$. If $\varphi = inn$, then $d^i_l = y \overset{\sigma_i}{\leadsto}_\varphi \top$ by using the rule $x = x \to true$. By Definition 8, every equation $c(\bar{x}) = c(\bar{x})$, with constructor symbol c/n, is in $T_\mathcal{R} \uparrow k$ for all k. Therefore, there exists a substitution α_i such that $(d^i_l = y)\sigma_i = (c(\bar{x}) = c(\bar{x}))\alpha_i$. Hence, there exits a substitution θ'_i such that $\theta'_i = mgu((d^i_l = y), (c(\bar{x}) = c(\bar{x})))$. Note that, $\theta'_{i\,|Var(d^i_l = y)} = \sigma_i$. If $\varphi = out$, then $d^i_l \approx y \overset{\sigma_i *}{\leadsto}_\varphi \top$ by using the rules defining the strict equality $\approx$ which are added to the program. Let $e'_i \equiv c_i(x) = c_i(x)$, then in both strategies we conclude that $e'_i = e_i\sigma_i$ and therefore e_i is closed by $\mathcal{F}^{ca}_\varphi(\mathcal{R})$.

Thus, we have proved that for all $1 \leq i \leq n, e'_i = e_i\sigma_i$ with $e'_i \in \mathcal{F}^{ca}_\varphi(\mathcal{R})$; hence,

$$\begin{aligned} mgu((e_1, \ldots, e_n), (e'_1, \ldots, e'_n)) &= mgu((e_1, \ldots, e_n), (e_1\sigma_1, \ldots, e_n\sigma_n)) \\ &= mgu(\hat{\sigma}_1, \ldots, \hat{\sigma}_n) \\ &= \theta'. \end{aligned}$$

Then, $\theta' = mgu(flat_\varphi(g), g')$ with $g' = e'_1, \ldots, e'_n$ and $\theta_{|g} = \theta'_{|g}$. Therefore, g is closed by $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ with substitution θ.

($\Leftarrow$): Let g be a goal closed by $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ such that $flat_\varphi(g) = (e_1, \ldots, e_n)$, then there exist $(e'_1, \ldots, e'_n) \in \mathcal{F}^{ca}_\varphi(\mathcal{R})$ s.t. $\theta' = mgu((e_1, \ldots, e_n), (e'_1, \ldots, e'_n))$. Therefore, for every $i = 1, \ldots, n$, there is σ_i s.t. $e_i\sigma_i = e'_i\sigma_i$. By Lemma 83, the left-hand sides of equations in $T^\varphi_\mathcal{R} \uparrow k$ are patterns or constructor terms; then by definition of $\mathcal{F}^{ca}_\varphi(\mathcal{R})$, each $e'_i \in \mathcal{F}^{ca}_\varphi(\mathcal{R})$ has the form $f_s(\bar{x})\beta_i = d^i_s$ or $d^i_l =_\varphi d^i_r$. Now, for $\varphi \in \{inn, out\}$ we know that θ' is a constructor substitution, hence so is σ_i. By Lemma 86, there exists a derivation $f_s(\bar{x}) = y \overset{\beta_i *}{\leadsto}_\varphi d^i_s = y$; hence $f_s(\bar{x})\beta_i = d^i_s \overset{\epsilon *}{\leadsto}_\varphi \top$, then $f_s(\bar{x})\beta_i\sigma_i = d^i_s\sigma_i \overset{\epsilon *}{\leadsto}_\varphi \top$.

Similarly, $d_l^i\sigma_i =_\varphi d_r^i\sigma_i \overset{\epsilon}{\leadsto}{}^*{}_\varphi \top$. Since $e_i'\sigma_i \overset{\epsilon}{\leadsto}{}^*{}_\varphi \top$, then $e_i\sigma_i \overset{\epsilon}{\leadsto}{}^*{}_\varphi \top$; hence $e_i \overset{\sigma_i}{\leadsto}{}^*{}_\varphi \top$. In general, $e_1 \overset{\sigma_1}{\leadsto}{}^*{}_\varphi \top, \ldots, e_n \overset{\sigma_n}{\leadsto}{}^*{}_\varphi \top$ and by Lemma 84 $(e_1, \ldots, e_n) \overset{\theta'}{\leadsto}{}^*{}_\varphi \top$ where $\theta' = \sigma_1 \Uparrow \ldots \Uparrow \sigma_n$. By Lemma 85, $g \overset{\theta}{\leadsto}{}^*{}_\varphi \top$ and $\theta = \theta'_{|Var(g)}$. □

The equivalence between the operational and the (computed answers) fixpoint semantics is established in the following lemma.

Lemma 87. *If $\mathcal{R} \in \mathbb{R}_{out}$, then $\mathcal{O}^{ca}_{out}(\mathcal{R}) = \{l = r \in \mathcal{F}^{ca}_{out}(\mathcal{R}) \mid \bot$ does not occur in $r\}$. If $\mathcal{R} \in \mathbb{R}_{inn}$, then $\mathcal{O}^{ca}_{inn}(\mathcal{R}) = \mathcal{F}^{ca}_{inn}(\mathcal{R})$.*

Proof. ($\supseteq$): Let $(f(\bar{x})\theta_i = t_i) \in \mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ where t_i is a constructor term and $\bot$ does not occur in t_i. By Lemma 86, there exists a narrowing derivation with strategy φ such that $f(\bar{x}) = y \overset{\theta_i}{\leadsto}{}^*_{\varphi}\ t_i = y$ and then $t_i = y \overset{\{y/t_i\}}{\leadsto}_{\varphi}\ \top$. Hence, by definition of $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$, we have

$$(f(\bar{x}) = y)\theta_i\{y/t_i\} \in \mathcal{O}^{ca}_{\varphi}(\mathcal{R}).$$

Since $y \notin Var(\theta_i)$ then

$$(f(\bar{x})\theta_i = t_n) \in \mathcal{O}^{ca}_{\varphi}(\mathcal{R}).$$

In the case of constructor equations, we distinguish between two cases: $\varphi = inn$ or $\varphi = out$.

When $\varphi = inn$ and $d_l^i = d_r^i \in \mathcal{F}^{ca}_{inn}(\mathcal{R})$, then $d_l^i = d_r^i \in \Im^{inn}_{\mathcal{R}}$ which is a subset of $\mathcal{O}^{ca}_{inn}(\mathcal{R})$. Therefore, by Definition 17, $d_l^i = d_r^i \in \mathcal{O}^{ca}_{inn}(\mathcal{R})$.

The case when $\varphi = out$ and $d_l^i \approx d_r^i \in \mathcal{F}^{ca}_{out}(\mathcal{R})$ is perfectly analogous, by considering the rules defining $\approx$.

Note that, we disregard all equations in $\mathcal{F}^{ca}_{out}(\mathcal{R})$ and $\mathcal{O}^{ca}_{out}(\mathcal{R})$ such that they contain $\bot$ symbol on their right-hand side.

($\subseteq$): Let $(s = t) \in \mathcal{O}^{ca}_{\varphi}(\mathcal{R})$, by definition of $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$; hence $s = t$ has the form $(f(\bar{x}) = y)\theta'$. Since θ' is a constructor substitution, then the equation can be written as $(f(\bar{x})\theta = t_n)$, where $\theta = \theta'_{|f(\bar{x})}$ and t_n is a constructor term. There are two cases:

If $f/m \in \mathcal{C}$, then trivially $(s = t) \in \mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ (see the first part of this proof).

If $f/m \in \mathcal{F}$, by definition of $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$, there exists a narrowing derivation with strategy φ

$$f(\bar{x}) = y \overset{\theta}{\leadsto}{}^*{}_{\varphi}\ t_n = y \overset{\{y/t_n\}}{\leadsto}{}_{\varphi}\ \top.$$

By Lemma 86, $(f(\bar{x})\theta = t_n) \in \mathcal{F}^{ca}_{\varphi}(\mathcal{R})$; therefore, $(s = t) \in \mathcal{F}^{ca}_{\varphi}(\mathcal{R})$. □

Theorem 19. *The following relation holds:*

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

Proof. It follows straightforwardly from Lemma 87. □

B.2. Proofs of Section 4

Proposition 26. *If there are no incorrect rules in $\mathcal{R}$ w.r.t. the intended fixpoint semantics $\mathbb{1}_{\mathcal{F}}$, then $\mathcal{R}$ is partially correct w.r.t. the intended success set semantics $\mathbb{1}_{ca}$.*

Proof. Consider $e \in \mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ and assume that $e \notin \mathbb{1}^{ca}$. Since $\mathbb{1}^{ca} \subseteq \mathbb{1}_{\mathcal{F}}$, this implies that e is an incorrectness symbol, which contradicts the hypothesis that $\mathcal{R}$ is correct. □

B.3. Proofs of Section 5

Proposition 41. *The operator $T^{\sharp\varphi}_{\mathcal{R}}$ is continuous in the lattice of abstract Herbrand interpretations, $\varphi \in \{inn, out\}$.*

Proof. It is analogous to Proposition 9. □

We now define an abstract narrowing relation by means of an abstract transition system. An abstract computation is performed w.r.t. an abstract goal and an abstract program $\mathcal{R}^\sharp$. The calculus corresponds to the concrete conditional one, where the concrete domains and operators are replaced by their corresponding abstract versions. The abstract *narrowing* relation is generic w.r.t. a strategy φ. Let $\mathcal{R}^\sharp_+$ be the extension of the abstract program $\mathcal{R}^\sharp$ by the rules that deal with syntactic equality, as in the concrete case.

Definition 88 (*Abstract narrowing $\leadsto_{\sharp\varphi}$*)**.** Let $\mathcal{R} \in \mathbb{R}_\varphi$ and $\mathcal{R}^\sharp$ be the corresponding abstraction of $\mathcal{R}$. We define abstract conditional narrowing with strategy φ, $\varphi \in \{inn, out\}$, as the smallest relation $\leadsto_{\sharp\varphi}$ satisfying:

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

Lemma 89 (*Termination of the abstract narrowing*). *Let $\mathcal{R} \in \mathbb{R}_\varphi$ and $\mathcal{R}^\sharp$ be the corresponding abstraction of $\mathcal{R}$ and g be an abstract goal. The abstract transition system for $\leadsto_{\sharp\varphi}$ and g has a finite number of nodes [15].*

Lemma 90. *Let $\mathcal{R} \in \mathbb{R}_\varphi$ and $\mathcal{R}^\sharp$ be the corresponding abstraction of $\mathcal{R}$. For all $k \geq 0$, we have that*

$$(f(\bar{x})\theta = t) \in T_{\mathcal{R}}^{\sharp\varphi} \uparrow k \quad \textit{iff} \quad f(\bar{x}) = y \overset{\mathcal{R}^\sharp, \theta *}{\leadsto}_{\sharp\varphi} t = y$$

where $\perp$ does not occur in t.

Proof (*Outline*). Analogous to Lemma 86. By simply replacing unification *mgu*, TRS $\mathcal{R}$, narrowing $\overset{\sigma}{\leadsto}_\varphi$ and the immediate consequences operator $T_{\mathcal{R}}^{\varphi}$ with abstract unification *mgu*$^\sharp$, abstract TRS $\mathcal{R}^\sharp$, abstract narrowing $\overset{\sigma}{\leadsto}_{\sharp\varphi}$ and the abstract immediate consequences operator $T_{\mathcal{R}}^{\sharp\varphi}$. Routine. □

Theorem 43. *There exists a finite positive number k such that $\mathcal{F}_\varphi^\sharp(\mathcal{R}) = T_{\mathcal{R}}^{\sharp\varphi} \uparrow k$, $\varphi \in \{inn, out\}$.*

Proof. Let us show that there exists a finite positive number k such that $\mathcal{F}_\varphi^\sharp(\mathcal{R}) = T_{\mathcal{R}}^{\sharp\varphi} \uparrow k$ is the fixpoint of the abstract immediate consequence operator. First, we show that $\mathcal{F}_\varphi^\sharp(\mathcal{R})$ is finite.

Let us assume, by contradiction, that $\mathcal{F}_\varphi^\sharp(\mathcal{R})$ is an infinite set. This means that the set generated by $T_{\mathcal{R}}^{\sharp\varphi} \uparrow k$ for each $k > 0$ is different from the preceding one; i.e., for all k there exists at least one equation $e \notin T_{\mathcal{R}}^{\sharp\varphi} \uparrow k$ such that $e \in T_{\mathcal{R}}^{\sharp\varphi} \uparrow (k+1)$. This implies that there is an infinite sequence:

$$\begin{array}{ll} f(\bar{x}) = f(\bar{x}) & \in T_{\mathcal{R}}^{\sharp\varphi} \uparrow 1 \\ f(\bar{x})\theta_1 = t_1 & \in T_{\mathcal{R}}^{\sharp\varphi} \uparrow 2 \\ f(\bar{x})\theta_1\theta_2 = t_2 & \in T_{\mathcal{R}}^{\sharp\varphi} \uparrow 3 \\ \vdots & \\ f(\bar{x})\theta_1\theta_2 \ldots \theta_{k+1} = t_{k+1} & \in T_{\mathcal{R}}^{\sharp\varphi} \uparrow (k+2) \\ \vdots & \end{array}$$

Moreover, for each equation of the form $f(\bar{x})\phi_i = t_i$ in this series, by Lemma 90 there exists an abstract narrowing derivation:

$$f(\bar{x}) = y \leadsto_{\sharp\varphi}^{\phi_i *} t_i = y.$$

Therefore, there exists an infinite abstract narrowing derivation $\leadsto_{\sharp\varphi}$ stemming from the goal $f(\bar{x}) = y$:

$$f(\bar{x}) = y \leadsto_{\sharp\varphi}^{*} t_1 = y \leadsto_{\sharp\varphi}^{*} t_2 = y \leadsto_{\sharp\varphi}^{*} \cdots \leadsto_{\sharp\varphi}^{*} t_k = y \leadsto_{\sharp\varphi}^{*} \ldots$$

which contradicts the termination of abstract narrowing proven in Lemma 89.

Thus, there exists a finite positive number k such that $\mathcal{F}_\varphi^\sharp(\mathcal{R}) = T_{\mathcal{R}}^{\sharp\varphi} \uparrow k$. □

Lemma 91. $T_{\mathcal{R}}^{\sharp\varphi} \propto T_{\mathcal{R}}^{\varphi}$.

Proof (*Sketch*). We must prove that, for all $\mathcal{I}, \mathcal{I}'$ such that $\mathcal{I}' \propto \mathcal{I}$, $T_{\mathcal{R}}^{\sharp\varphi}(\mathcal{I}') \propto T_{\mathcal{R}}^{\varphi}(\mathcal{I})$, that is $T_{\mathcal{R}}^{\varphi}(\mathcal{I}) \in \gamma(T_{\mathcal{R}}^{\sharp\varphi}(\mathcal{I}'))$. The proof follows immediately from Lemmas 86, 90 and the fact that, for any strategy φ with the property of *non-productive substitutions*, $\overset{\mathcal{R}'}{\leadsto}_{\sharp\varphi} \propto \overset{\mathcal{R}}{\leadsto}_\varphi$ whenever $\mathcal{R}' \propto \mathcal{R}$. □

Theorem 44. $\mathcal{F}_\varphi^\sharp(\mathcal{R}) \propto \mathcal{F}_\varphi(\mathcal{R})$ *and* $\mathcal{F}_\varphi^{ca\sharp}(\mathcal{R}) \propto \mathcal{F}_\varphi^{ca}(\mathcal{R})$.

Proof. The result is an immediate corollary of Lemma 91. □

In [8], this result was only established for $\varphi = inn$, as we heavily relied on the abstract narrowing methodology of [15] which was developed for an eager strategy, namely (innermost) basic conditional narrowing. It was only afterwards that we realized that the key property for the 'inherited' abstract narrowing results to hold is the property of non-productive substitutions, which is trivially fulfilled by innermost as well as outermost narrowing because both of them only compute constructor substitutions.

Theorem 46 (*Completeness*). *Let $\mathcal{R}$ be a program in $\mathbb{R}_\varphi$ and g be a (non-trivial) goal. If θ is a computed answer substitution for g in $\mathcal{R}$ w.r.t. φ, then g is abstractly closed by $\mathcal{F}_\varphi^{ca\sharp}(\mathcal{R})$ with substitution θ' and $(\theta' \preceq \theta)_{|Var(g)}$.*

Proof (*Sketch*). It is analogous to Theorem 15 by using Lemma 90 in the place of Lemma 86, and Theorem 44. □

B.4. Proofs of Section 6

Theorem 50. *Let $(\mathcal{I}^+, \mathcal{I}^-)$ be a correct approximation of the intended semantics $\mathcal{I}_\mathcal{F}$. If r is abstractly incorrect w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$ on e, then r is incorrect on e.*

Proof. Let r be a rule such that it is abstractly incorrect on e w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$. Then, by Definition 49, $e \in T^{\varphi}_{\{r\}}(\mathcal{I}^-)$ and for all $\mathcal{I} \in \gamma(\mathcal{I}^+))$, e is not closed by $\mathcal{I}$. Note that e is not an abstract equation. We will prove that the rule $r \in \mathcal{R}$ is incorrect on e according to Definition 24.

Since $(\mathcal{I}^+, \mathcal{I}^-)$ is a correct approximation of the intended semantics $\mathcal{I}_{\mathcal{F}}$, then $\mathcal{I}^- \subseteq \mathcal{I}_{\mathcal{F}}$. By the continuity of $T^{\varphi}_{\mathcal{R}}$, $T^{\varphi}_{\{r\}}(\mathcal{I}^-) \subseteq T^{\varphi}_{\{r\}}(\mathcal{I}_{\mathcal{F}})$. Hence $e \in T^{\varphi}_{\{r\}}(\mathcal{I}_{\mathcal{F}})$.

On the other hand, if $(\mathcal{I}^+, \mathcal{I}^-)$ is a correct approximation of the intended semantics $\mathcal{I}_{\mathcal{F}}$, then we have $\mathcal{I}_{\mathcal{F}} \in \gamma(\mathcal{I}^+)$, and since e is not closed by any $\mathcal{I} \in \gamma(\mathcal{I}^+)$, then in particular e is not covered by $\mathcal{I}_{\mathcal{F}}$.

Therefore, we have proved that $e \in T^{\varphi}_{\{r\}}(\mathcal{I}_{\mathcal{F}})$ whereas e is not closed by $\mathcal{I}_{\mathcal{F}}$, and the result follows. □

Theorem 51. *Let $(\mathcal{I}^+, \mathcal{I}^-)$ be a correct approximation of the intended semantics $\mathcal{I}_{\mathcal{F}}$. If $\mathcal{R}$ is abstractly incomplete on e w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$, then e is uncovered in $\mathcal{R}$.*

Proof. Since $\mathcal{R}$ is abstractly incomplete on e w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$, then by Definition 49, $e \in \mathcal{I}^-$ and e is not closed by any $\mathcal{I} \in \gamma(T^{\sharp\varphi}_{\mathcal{R}}(\mathcal{I}^+))$. We will prove that e is uncovered by $\mathcal{R}$, i.e., $e \in \mathcal{I}_{\mathcal{F}}$ and e is not covered by $T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$.

Consider $e \in \mathcal{I}^-$. Since $\mathcal{I}^- \subseteq \mathcal{I}_{\mathcal{F}}$, then $e \in \mathcal{I}_{\mathcal{F}}$. On the other hand, since $T^{\sharp\varphi}_{\mathcal{R}} \propto T^{\varphi}_{\mathcal{R}}$ (Lemma 91) and $\mathcal{I}^+ \propto \mathcal{I}_{\mathcal{F}}$ (hypothesis), then $T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}}) \in \gamma(T^{\sharp\varphi}_{\mathcal{R}}(\mathcal{I}^+))$. Now, since e is not closed by any $\mathcal{I} \in \gamma(T^{\sharp\varphi}_{\mathcal{R}}(\mathcal{I}^+))$, then e is not closed by $T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$, which proves the claim. □

Lemma 54. *Let $(c\mathcal{I}^+, c\mathcal{I}^-)$ be a computed approximation of the intended semantics $\mathcal{I}_{\mathcal{F}}$. Then $(c\mathcal{I}^+, c\mathcal{I}^-)$ is a correct approximation of $\mathcal{I}_{\mathcal{F}}$.*

Proof. First, let $\mathcal{R}_{Spec}$ be a program, $\mathcal{I}_{\mathcal{F}} = lfp(T^{\varphi}_{\mathcal{R}_{Spec}})$ and $c\mathcal{I}^- = T^{\varphi}_{\mathcal{R}_{Spec}} \uparrow i$ for $i \geq 0$. Since $T^{\varphi}_{\mathcal{R}_{Spec}} \uparrow i \subseteq T^{\varphi}_{\mathcal{R}_{Spec}} \uparrow \omega$ for each $i \geq 0$ then $c\mathcal{I}^- \subseteq \mathcal{I}_{\mathcal{F}}$.

Second, let $\mathcal{R}^{\sharp}_{Spec}$ be the corresponding abstract program of $\mathcal{R}_{Spec}$ and $c\mathcal{I}^+ = lfp(T^{\sharp\varphi}_{\mathcal{R}_{Spec}})$. Then $c\mathcal{I}^+ = \mathcal{F}^{\sharp}_{\varphi}(\mathcal{R})$, by Theorem 44, $c\mathcal{I}^+ \propto \mathcal{I}_{\mathcal{F}}$ then $\mathcal{I}_{\mathcal{F}} \in \gamma(c\mathcal{I}^+)$. □

Theorem 55. *Let $(c\mathcal{I}^+, c\mathcal{I}^-)$ be a computed approximation of $\mathcal{I}_{\mathcal{F}}$. If there exists an equation e such that, $e \in T^{\varphi}_{\{r\}}(c\mathcal{I}^-)$ and e is not abstractly closed by $c\mathcal{I}^+$, then the rule $r \in \mathcal{R}$ is incorrect on e.*

Proof. Since $(c\mathcal{I}^+, c\mathcal{I}^-)$ is a computed approximation of $\mathcal{I}_{\mathcal{F}}$, by Theorem 54, $(c\mathcal{I}^+, c\mathcal{I}^-)$ is a correct approximation of the intended semantics $\mathcal{I}_{\mathcal{F}}$, hence $c\mathcal{I}^- \subseteq \mathcal{I}_{\mathcal{F}}$ and $\mathcal{I}_{\mathcal{F}} \in \gamma(c\mathcal{I}^+)$. By hypothesis, e is not abstractly closed by $c\mathcal{I}^+$; hence, we have that for all $g' \in c\mathcal{I}^+$, $mgu^{\sharp}(flat_{\varphi}(e), g') = fail$. Therefore, e is not abstractly closed by $\mathcal{I}_{\mathcal{F}}$.

On the other hand, if $e \in T^{\varphi}_{\{r\}}(c\mathcal{I}^-)$, by the monotonicity of $T^{\varphi}_{\mathcal{R}}$, we obtain that $e \in T^{\varphi}_{\{r\}}(\mathcal{I}_{\mathcal{F}})$.

Therefore, we have that $e \in T^{\varphi}_{\{r\}}(\mathcal{I}_{\mathcal{F}})$ whereas e is not closed by $\mathcal{I}_{\mathcal{F}}$. Thus, the rule $r \in \mathcal{R}$ is incorrect on e. □

Theorem 56. *Let $(c\mathcal{I}^+, c\mathcal{I}^-)$ be a computed approximation of $\mathcal{I}_{\mathcal{F}}$. If there exists an equation e such that $e \in c\mathcal{I}^-$ and e is not abstractly closed by $T^{\sharp\varphi}_{\mathcal{R}}(c\mathcal{I}^+)$ then e is uncovered in $\mathcal{R}$.*

Proof. Since $(c\mathcal{I}^+, c\mathcal{I}^-)$ is a computed approximation of $\mathcal{I}_{\mathcal{F}}$, by Theorem 54, $(c\mathcal{I}^+, c\mathcal{I}^-)$ is a correct approximation of the intended semantics $\mathcal{I}_{\mathcal{F}}$; hence, $c\mathcal{I}^- \subseteq \mathcal{I}_{\mathcal{F}}$ and $\mathcal{I}_{\mathcal{F}} \in \gamma(c\mathcal{I}^+)$.

Since e is not abstractly closed by $T^{\sharp\varphi}_{\mathcal{R}}(c\mathcal{I}^+)$, then for all $g' \in T^{\sharp\varphi}_{\mathcal{R}}(c\mathcal{I}^+)$, $mgu^{\sharp}(flat_{\varphi}(e), g') = fail$. Hence, since $T^{\sharp\varphi}_{\mathcal{R}} \propto T^{\varphi}_{\mathcal{R}}$ (Lemma 91) and $c\mathcal{I}^+ \propto \mathcal{I}_{\mathcal{F}}$, we derive that $T^{\sharp\varphi}_{\mathcal{R}}(c\mathcal{I}^+) \propto T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$. This implies that e is not closed by $T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$, and the result follows. □

B.5. Proofs of Section 7

In order to prove Proposition 66, we need the following auxiliary lemmata.

Lemma 92. *Let $\mathcal{R}$ be a program and $r \equiv (\lambda \to \rho \Leftarrow C) \in \mathcal{R}$ be a rule. Let σ be a constructor substitution and $s \in \tau(\Sigma \cup V)$. If $s\sigma \xrightarrow{r,p} t$, then*

(1) $p \in \overline{O}(s\sigma) \cap \overline{O}(s)$;
(2) $mgu(\{s_{|p} = \lambda\}) = \theta$ *and* $\theta_{|Var(s)} \leq \sigma_{|Var(s)}$.

Proof. To prove claim 1, simply note that $s\sigma_{|p}$ is a redex of $s\sigma$; thus, $p \in \overline{O}(s\sigma)$. Moreover, no redexes of $s\sigma$ can occur in subterms introduced in s by σ, as σ is a constructor substitution. Therefore, $p \in \overline{O}(s)$. And finally, $p \in \overline{O}(s\sigma) \cap \overline{O}(s)$.

Let us demonstrate claim 2. First of all, we assume that s and λ are variable disjoint; otherwise, we choose a suitable variant of r which meets this assumption. Since $s\sigma_{|p}$ is a redex of $s\sigma$, there exists a substitution ϕ such that $s\sigma_{|p} = \lambda\phi$. This implies that $s_{|p}$ and λ are unifiable (e.g., $\phi\sigma$ is a unifier for $s_{|p}$ and λ). Therefore, the most general unifier $mgu(\{s_{|p} = \lambda\}) = \theta$ exists and $\theta_{|Var(s)} \leq (\phi\sigma)_{|Var(s)}$. Since $s_{|p}$ and λ are variable disjoint, $(\phi\sigma)_{|Var(s)} = \sigma_{|Var(s)}$. Hence, $\theta_{|Var(s)} \leq \sigma_{|Var(s)}$. □

Let p_1 and p_2 be two positions. We say that p_1 and p_2 are *not comparable* (in symbols, $p_1 \,||\, p_2$), if $p_1 \not\leq p_2$ and $p_2 \not\leq p_1$. A position p_1 is said to be *on the left* (resp., *on the right*) of a position p_2, if $p_1 \,||\, p_2$, $p_1 = w.i.w'$, $p_2 = w.j.w''$ and $i < j$ (resp., $i > j$) for some sequences of natural numbers w, w', w'' and natural numbers i, j. Analogously, given a term t and subterms $t_{|p}$ and $t_{|q}$, we say that $t_{|p}$ is *on the left* (resp., *the right*) of $t_{|q}$, if p is on the left (resp., the right) of q.

Lemma 93. *Let $\mathcal{R}$ be a program and $r \equiv (\lambda \to \rho \Leftarrow C) \in \mathcal{R}$ be a rule. Let σ be a constructor substitution and $s, t \in \tau(\Sigma \cup V)$.*

(1) *If $s\sigma \overset{r,p}{\to} t$, with $p \in \overline{O}(s\sigma)$, then $s \overset{r,p,\theta}{\rightsquigarrow} (t', C\theta)$ and there exists σ' such that $\sigma_{|Var(s)} = (\sigma'\theta)_{|Var(s)}$ with $t'\sigma' \equiv t$.*
(2) *If $\mathcal{R}$ is completely defined and p is the position of the leftmost, innermost redex of $s\sigma$, then p is the leftmost, innermost position narrowable in s.*

Proof. Let r be $\lambda \to \rho \Leftarrow C$. Let us first demonstrate point 1. By Lemma 92, there exist $p \in \overline{O}(s\sigma) \cap \overline{O}(s)$, $mgu(\{s_{|p} = \lambda\}) = \theta$ and $\theta_{|Var(s)} \leq \sigma_{|Var(s)}$. Therefore, the following narrowing step is enabled:

$$s \overset{r,p,\theta}{\rightsquigarrow} (t', C\theta) \quad \text{with } (s_{|p})\theta = \lambda\theta.$$

Besides, since $\theta_{|Var(s)} \leq \sigma_{|Var(s)}$, there exists a substitution σ' such that $(\sigma'\theta)_{|Var(s)} = \sigma_{|Var(s)}$.

Now, we have that

$$(s_{|p})\sigma = (s\sigma'\theta)_{|p} = \sigma'(\theta(s_{|p})) = \lambda\sigma'\theta$$

which implies that $t \equiv s\sigma[\rho\sigma'\theta]_p$. Finally,

$$t'\sigma' = (s[\rho]_p)\sigma'\theta = s\sigma'\theta[\rho\sigma'\theta]_p = s\sigma[\rho\sigma'\theta]_p \equiv t.$$

To prove point 2, we suppose by contradiction that p is not the leftmost, innermost narrowable position of s. Thus, there exists q, which is on the left of p, such that s is narrowable at position q. Now, for each constructor substitution σ, $s\sigma_{|q}$ is a redex of $s\sigma$, as $\mathcal{R}$ is completely defined. Hence, there is a redex of $s\sigma$ which is on the left of $s\sigma_{|p}$, which contradicts that $s\sigma_{|p}$ is the leftmost, innermost redex of $s\sigma$. □

Lemma 94. *Let $\mathcal{R}$ be a completely defined program and $r_1, r_2 \in \mathcal{R}$ be two rules with r_1 discriminable. Let e_0, e_1 and e_2 be equations. If $e_0 \overset{r_1,p_1}{\to} e_1 \overset{r_2,p_2}{\to} e_2$, $p_i \in \overline{O}(e_{i-1})$, is a leftmost, innermost rewrite sequence, then $e_0 \overset{r^*,p_1}{\to} e_2$ is a leftmost, innermost rewrite sequence where $r^* \in \mathsf{U}_{r_1}(\mathcal{R})$.*

Proof. Let r_1 be the rule $\lambda_1 \to \rho_1 \Leftarrow C_1$. Let us consider the leftmost, innermost rewrite sequence $\mathcal{D} \equiv e_0 \overset{r_1,p_1}{\to} e_1 \overset{r_2,p_2}{\to} e_2$. Then $e_{0|p_1} \equiv \lambda_1\sigma_1$ is the leftmost, innermost redex of e_0. σ_1 must be a constructor substitution, since $\mathcal{R}$ is completely defined (otherwise, the leftmost, innermost redex would not be rooted at p_1).

Moreover, $e_1 \equiv e_0[\rho_1\sigma_1]_{p_1}$. As rule r_1 is discriminable (in particular, unfoldable), some defined function occurs in $\rho_1\sigma_1$. Again, by the fact that $\mathcal{R}$ is completely defined, each subterm in $\rho_1\sigma_1$ of the form $f(t_1, \ldots, t_n)$, where f is a defined function symbol, is a redex of $\rho_1\sigma_1$. Let $p' \in \overline{O}(\rho_1\sigma_1)$ be the position of the leftmost, innermost redex of $\rho_1\sigma_1$.

Now, we show that $p_2 = p_1.p'$. Suppose by contradiction that $p_2 \neq p_1.p'$; hence, $p_1 \not\leq p_2$. We distinguish three exhaustive cases.

$p_2 < p_1$. In this case, the contradiction is immediate: $e_{1|p_1}$ cannot be the leftmost, innermost redex of e_1, since there is an inner redex of e_1 rooted at position $p_1.p'$.

p_2 *is on the left of* p_1. This implies that $e_{1|p_2} \equiv e_{0|p_2}$. Hence, there is a redex of e_0 (namely, $e_{0|p_2}$) occurring on the left of the redex $e_{0|p_1}$, which contradicts the hypothesis that $e_{0|p_1}$ is the leftmost, innermost redex of e_0.

p_2 *is on the right of* p_1. p_2 is also on the right of $p_1.p'$. And $e_{1|p_1.p'}$ is a redex of e_1, which contradicts the hypothesis that $e_{1|p_2}$ is the leftmost, innermost redex of e_1.

Therefore, $p_2 = p_1.p'$.

Now, consider the leftmost, innermost rewrite step $\rho_1\sigma_1 \overset{r_2,p'}{\to} t$. Since $p_2 = p_1.p'$, the rewrite sequence $\mathcal{D}$ can be seen as follows:

$$e_0 \overset{r_1,p_1}{\to} e_0[\rho_1\sigma_1]_{p_1} \overset{r_2,p_1.p'}{\to} e_0[t]_{p_1} \equiv e_2.$$

By applying point 1 of Lemma 93 to the rewrite step $\rho_1\sigma_1 \overset{r_2,p'}{\to} t$, we have that (i) $\rho_1 \overset{r_2,p',\theta}{\rightsquigarrow} (t', C_2\theta)$, where C_2 is the condition of rule r_2; (ii) there exists a substitution σ' such that $t \equiv t'\sigma'$ and $\sigma_{1|Var(\rho_1)} = (\sigma'\theta)_{|Var(\rho_1)}$. Besides, the rule $r^* \equiv \lambda_1\theta \to t'$ belongs to $\mathsf{U}_{r_1}(\mathcal{R})$ by point 2 of Lemma 93 and Definition 62.

Finally, we complete the proof by simply noting that

$$e_0 \equiv e_0[\lambda_1\sigma_1]_{p_1} \equiv e_0[\lambda_1\sigma'\theta]_{p_1} \overset{r^*,p_1}{\to} e_0[t'\sigma']_{p_1} \equiv e_0[t]_{p_1} \equiv e_2. \quad \square$$

Lemma 95. *Let $\mathcal{R} \in \mathbb{R}_{inn}$ and $r_1, r_2 \in \mathcal{R}$, with $r_1 \equiv (\lambda_1 \to \rho_1 \Leftarrow C_1)$ and $r_2 \equiv (\lambda_2 \to \rho_2 \Leftarrow C_2)$. Let $\rho_1 \overset{\theta,r_2,q}{\rightsquigarrow}_{inn} (\rho^*, C^*)$ be a leftmost-innermost narrowing step such that $r^* \equiv (\lambda_1\theta \to \rho^* \Leftarrow C^*)$ and $r^* \in \mathsf{U}_{r_2}(r_1)$. If $e \overset{r^*,p}{\to} e''$ for some $p \in \overline{O}(e)$, then $e \overset{r_1,p}{\to} e' \overset{r_2,p.q}{\to} e''$.*

Proof. Given the leftmost-innermost narrowing step $\rho_1 \overset{\theta,r_2,q}{\leadsto}_{inn} (\rho^*, C^*)$, by the soundness of leftmost-innermost narrowing w.r.t. $\mathbb{R}_{inn}$, we get $\rho_1\theta \overset{r_2,q}{\to} \rho^*$. Since $e \overset{r^*,p}{\to} e''$, there exists a substitution σ such that $\lambda_1\theta\sigma = e|_p$ and $e'' = e[\rho^*\sigma]_p$. Since $\rho_1\theta \overset{r_2,q}{\to} \rho^*$, by the stability of the rewrite relation, we get $\rho_1\theta\sigma \overset{r_2,q}{\to} \rho^*\sigma$. Therefore, $e \equiv e[\lambda_1\theta\sigma]_p \overset{r_1,p}{\to} e[\rho_1\theta\sigma]_p \overset{r_2,p.q}{\to} e[\rho^*\sigma]_p \equiv e''$, where $e' \equiv e[\rho_1\theta\sigma]_p$. □

Now, we are able to proceed with the proof of Proposition 66.

Proposition 66. *Let $\mathcal{R} \in \mathbb{R}_{inn}$, $\mathcal{R}' = \mathsf{U}_r(\mathcal{R})$, $r \in \mathcal{R}$ be a discriminable rule, and e be an equation. Then, we have*

(1) *if $e \to^* true$ in $\mathcal{R}$, then also $e \to^* true$ in $\mathcal{R}'$*
(2) *if $r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}}(\mathsf{e}))$, then $|\mathcal{D}_{\mathcal{R}'}(e)| < |\mathcal{D}_{\mathcal{R}}(e)|$*
(3) *if $e \to^* true$ in $\mathcal{R}'$, then also $e \to^* true$ in $\mathcal{R}$.*

Proof. Claims (1) and (2). Since the leftmost, innermost rewrite strategy is complete w.r.t. the class of programs $\mathbb{R}_{inn}$ and $e \to^* true$ in $\mathcal{R}$, then there exists a leftmost, innermost rewrite sequence from e to *true* in $\mathcal{R}$. Let us call this rewrite sequence $\mathcal{D}_{\mathcal{R}}(e)$. We prove (1) and (2) by induction on the length n of $\mathcal{D}_{\mathcal{R}}(e)$.

n = 0. Since $e \equiv true$, claims (1) and (2) hold trivially.

n > 0. Equation e contains at least one redex. Let $e_{|q}$ be the leftmost, innermost redex of e which is reduced via $r_1 \equiv \lambda_1 \to \rho_1 \Leftarrow C_1 \in \mathcal{R}$. So, we have

$$\mathcal{D}_{\mathcal{R}}(e) \equiv e \overset{q,r_1}{\to} e[\rho_1\sigma_1]_q \to^* true.$$

Here, we consider two cases.

Case $\mathbf{r_1} \in \mathcal{R}'$. Since r_1 belongs to both programs $\mathcal{R}$ and $\mathcal{R}'$, we have that the first reduction step of $\mathcal{D}_{\mathcal{R}}(e)$ is also a reduction step w.r.t. $\mathcal{R}'$. Moreover, by the induction hypothesis, $e[\rho_1\sigma_1]_q \to^* true$ is a successful rewrite sequence in $\mathcal{R}'$. Thus, (1) holds.

To prove (2), we first observe that $r_1 \not\equiv r$ by Definition 62 (unfolding). By induction hypothesis, if $r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}}(\mathsf{e}[\rho_1\sigma_1]_q))$, then $|\mathcal{D}_{\mathcal{R}'}(e[\rho_1\sigma_1]_q)| < |\mathcal{D}_{\mathcal{R}}(e[\rho_1\sigma_1]_q)|$, and claim (2) follows.

Case $\mathbf{r_1} \notin \mathcal{R}'$. By Definition 62, we have that $r_1 \equiv r$. Since $r \equiv r_1$ is discriminable (in particular, unfoldable), there is at least one defined function on the right-hand side of r_1. Therefore, the leftmost, innermost rewrite sequence $\mathcal{D}_{\mathcal{R}}(e)$ has the form

$$\mathcal{D}_{\mathcal{R}}(e) \equiv e \overset{q,r}{\to} e[\rho_1\sigma_1]_q \overset{p,r_2}{\to} e'' \to^* true$$

with $r_2 \in \mathcal{R}$.

By applying Lemma 94 to the following leftmost, innermost rewrite sequence

$$e \overset{q,r}{\to} e[\rho_1\sigma_1]_q \overset{p,r_2}{\to} e''$$

we know that

$$e \overset{q,r^*}{\to} e''$$

with $r^* \in \mathcal{R}' = \mathsf{U}_r(\mathcal{R})$. Now, by the induction hypothesis, a leftmost, innermost rewrite sequence for e'' to *true* in $\mathcal{R}'$ does exist. Hence, the sequence $\mathcal{D}_{\mathcal{R}'}(e) \equiv e \to^* true$ must exist, which demonstrates (1).

Let us give the proof for (2). Clearly,

$$|e \overset{q,r^*}{\to} e''| < |e \overset{q,r}{\to} e[\rho_1\sigma_1]_q \overset{p,r_2}{\to} e''|.$$

Now, if r occurs in $\mathsf{OR}(\mathcal{D}_{\mathcal{R}'}(\mathsf{e}''))$, by induction hypothesis,

$$|\mathcal{D}_{\mathcal{R}'}(e'')| < |\mathcal{D}_{\mathcal{R}}(e'')|,$$

thus claim 2 follows directly. Otherwise, If r does not occur in $\mathsf{OR}(\mathcal{D}_{\mathcal{R}'}(\mathsf{e}''))$, then by Definition 62, each rule in $\mathsf{OR}(\mathcal{D}_{\mathcal{R}'}(\mathsf{e}'')) \subset \mathcal{R}$. Then,

$$|\mathcal{D}_{\mathcal{R}'}(e'')| = |\mathcal{D}_{\mathcal{R}}(e'')|,$$

hence $|\mathcal{D}_{\mathcal{R}'}(e)| < |\mathcal{D}_{\mathcal{R}}(e)|$.

Claim (3). Again, since the leftmost, innermost rewrite strategy is complete w.r.t. the class of programs $\mathbb{R}_{inn}$ and $e \to^* true$ in $\mathcal{R}'$, then there exists a leftmost, innermost rewrite sequence from e to *true* in $\mathcal{R}'$. Let us call this rewrite sequence $\mathcal{D}_{\mathcal{R}'}(e)$. Now, we prove (3) by induction on the length n of $\mathcal{D}_{\mathcal{R}'}(e)$.

n = 0. Since $e \equiv true$, claim (3) is trivial.

n > 0. Let $\mathcal{D}_{\mathcal{R}'}(e)$ be the following leftmost-innermost rewrite sequence:

$$e \equiv e_0 \overset{p_0,r_0}{\to} e_1 \overset{p_1,r_1}{\to} \cdots e_{n-1} \overset{p_{n-1},r_{n-1}}{\to} e_n \equiv true.$$

First, we observe that—by the inductive hypothesis and the completeness of the leftmost-innermost rewrite strategy—there exists a leftmost-innermost rewrite sequence $e \equiv e_0 \to^* e_{n-1}$ in $\mathcal{R}$.

Then, in order to prove $e \to^* e_n \equiv \mathit{true}$ in $\mathcal{R}$, it suffices to show that it is always possible to rewrite e_{n-1} to e_n in $\mathcal{R}$ and then combining the rewrite sequence $e \equiv e_0 \to^* e_{n-1}$ given by the induction hypothesis with the rewrite step $e_{n-1} \to e_n$. To this purpose, we consider the following two cases.

Case $\mathbf{r_{n-1}} \in \mathcal{R}$. In this case, the rewrite step $e_{n-1} \overset{p_{n-1},r_{n-1}}{\to} e_n \equiv \mathit{true}$ is enabled both in $\mathcal{R}'$ and $\mathcal{R}$. Thus, $e \equiv e_0 \to^* e_{n-1} \overset{p_{n-1},r_{n-1}}{\to} e_n \equiv \mathit{true}$ in $\mathcal{R}$.

Case $\mathbf{r_{n-1}} \notin \mathcal{R}$. Since $r_{n-1} \notin \mathcal{R}$ and $r_{n-1} \in \mathcal{R}'$, r_{n-1} is a rule of $\mathcal{R}'$ which has been obtained by unfolding a rule $r_1 \in \mathcal{R}$ w.r.t. a rule $r_2 \in \mathcal{R}$ by means of a leftmost-innermost narrowing step. Hence, $r_{n-1} \in \mathsf{U}_{r_2}(r_1)$. Now, since $e_{n-1} \overset{p_{n-1},r_{n-1}}{\to} e_n \equiv \mathit{true}$ in $\mathcal{R}'$ and $r_{n-1} \in \mathsf{U}_{r_2}(r_1)$, we can obtain the rewrite sequence

$$e_{n-1} \overset{p_{n-1},r_1}{\to} e' \overset{p',r_2}{\to} e_n \equiv \mathit{true} \quad \text{in } \mathcal{R}$$

by applying Lemma 95. Thus, $e \equiv e_0 \to^* e_{n-1} \overset{p_{n-1},r_1}{\to} e' \overset{p',r_2}{\to} e_n \equiv \mathit{true}$ in $\mathcal{R}$. □

Corollary 96. *Let $\mathcal{R} \in \mathbb{R}^u$, $\mathcal{R}' = \mathsf{U}_r(\mathcal{R})$, $r \in \mathcal{R}$ be a discriminable rule, and e be an equation. Then, $e \to^* \mathit{true}$ in $\mathcal{R}$ iff $e \to^* \mathit{true}$ in $\mathcal{R}'$.*

Proof. It directly follows by Proposition 66 and by the fact that $\mathbb{R}^u \subseteq \mathbb{R}_{inn}$. □

Lemma 97. *Let $\mathcal{R} \in \mathbb{R}_{inn}$, and $l = c$ be an example such that $c \in \tau(\mathcal{C})$. If $(l = c) \to^* \mathit{true}$ in $\mathcal{R}$, then there exists a leftmost, innermost rewrite sequence*

$$l = c \overset{q_1,r_1}{\to} l_1 = c \overset{q_2,r_2}{\to} l_2 = c \cdots \overset{q_n,r_n}{\to} l_n = c \quad \text{in } \mathcal{R}$$

where $l_n \equiv c$, $n \geq 0$, $r_i \in \mathcal{R}$, $q_i \in \overline{O}(l_i)$, $i = 1, \ldots, n$.

Proof. Straightforward. □

Proposition 70. *Let $\mathcal{R} \in \mathbb{R}_{inn}$. Let E^p (resp. E^n) be a set of positive (resp. negative) examples. If there are no $e^p \in E^p$ and $e^n \in E^n$ that can be proven in $\mathcal{R}$ by using the same rule application sequence, then, for each unfolding succession $\mathcal{S}(\mathcal{R})$, there exists k such that $\forall e^n \in E^n,\ \exists r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_k}(e^n))$ s.t. r is not discriminable.*

Proof. The key idea for the proof is in the following fact (which holds by Proposition 66).

> At each unfolding step involving a discriminable rule, the length of the proof of, at least, one positive example decreases.

Therefore, by a finite number k of unfolding steps, we get the program $\mathcal{R}_k$ where each $e^p \equiv (l^p = c^p) \in E^p$, $l^p \in \tau(\mathcal{F} \cup \mathcal{C})$ and $c^p \in \tau(\mathcal{C})$, succeeds by using just one rule r of $\mathcal{R}_k$. Hence, by Lemma 97, we have $(l^p = c^p) \overset{q,r}{\to} (c^p = c^p)$, with $r \in \mathcal{R}_k$, $q \in \overline{O}(l^p)$.

This amounts to saying that there is no defined symbol on the right-hand side of r.

Now, consider a negative example $e^n \equiv l^n = c^n$, $l^n \in \tau(\mathcal{F} \cup \mathcal{C})$ and $c^n \in \tau(\mathcal{C})$, and the corresponding proof $\mathcal{D}_{\mathcal{R}_k}(e^n)$. In order to prove the claim, we distinguish two cases:

$|\mathcal{D}_{\mathcal{R}_k}(\mathbf{e^n})| > \mathbf{1}$. In this case, there exists one rule $r \in \mathcal{R}_k$ occurring in $\mathcal{D}_{\mathcal{R}_k}(e^n)$, where the right-hand side of r contains at least one defined function symbol. Hence, r cannot occur in the proof of any positive example, and the claim follows.

$|\mathcal{D}_{\mathcal{R}_k}(\mathbf{e^n})| = \mathbf{1}$. Let $r \in \mathcal{R}_k$ be the rule used to prove e^n. By contradiction, suppose that there exists a positive example $e^p \in E^p$, whose proof $\mathcal{D}_{\mathcal{R}_k}(e^p)$ uses the very same rule r. Since $\mathcal{R}_k$ derives from $\mathcal{R}$ by repeatedly applying unfolding, the application of the rule $r \in \mathcal{R}_k$ can be mimicked in $\mathcal{R}$ by applying the rule application sequence $\langle r_1, \ldots, r_n \rangle$ of $\mathcal{R}$. This means that examples e^p and e^n can be proven in $\mathcal{R}$ by using the same rules sequence. That is,

$$e^p \overset{r_1}{\to} e_1^p \overset{r_2}{\to} \cdots \overset{r_n}{\to} c^p = c^p$$
$$e^n \overset{r_1}{\to} e_1^n \overset{r_2}{\to} \cdots \overset{r_n}{\to} c^n = c^n.$$

This fact contradicts the hypothesis that no $e^p \in E^p$ and $e^n \in E^n$ can be proven in $\mathcal{R}$ by using the same rule application sequence. □

Theorem 71. *Let $\mathcal{R} \in \mathbb{R}^u$. Let $\mathcal{I}_{\mathcal{F}}$ be the intended fixpoint semantics of $\mathcal{R}$, and $(c\mathcal{I}^+, c\mathcal{I}^-)$ be a computed approximation of $\mathcal{I}_{\mathcal{F}}$. Then, if E^p and E^n are two sets of examples generated w.r.t. $(c\mathcal{I}^+, c\mathcal{I}^-)$ such that $\mathcal{R} \vdash_{\varphi} E^p$, $\varphi \in \{inn, out\}$, and there are no $e^p \in E^p$ and $e^n \in E^n$ which can be proven in $\mathcal{R}$ by using the same rule application sequence, then the execution of* TD-Corrector$(\mathcal{R}, c\mathcal{I}^+, c\mathcal{I}^-)$ *yields a corrected program $\mathcal{R}^c$ w.r.t. E^p and E^n.*

Proof. Let us consider a uniform program $\mathcal{R} \in \mathbb{R}^u$ along with two sets of examples E^p and E^n. Any example in $E^p \cup E^n$ is a ground equation of the form $l = c$, with $l \in \tau(\mathcal{F} \cup \mathcal{C})$ and $c \in \tau(\mathcal{C})$. Note that example sets can be effectively computed by using the example generation procedure which we described in Section 7.2.1. In fact, ground sets E^p and E^n are effectively generated by using the computable abstract tests of Theorems 55 and 56. Therefore, the execution of line 2 of TD-Corrector$(\mathcal{R}, c\mathcal{I}^+, c\mathcal{I}^-)$ always terminates delivering a set of positive examples E^p and a set of negative examples E^n.

Since $\mathcal{R} \in \mathbb{R}^u$, (i) $\mathcal{R}$ is terminating; (ii) both the narrowing strategy *out* and the narrowing strategy *inn* are complete w.r.t. $\mathcal{R}$, the overgenerality test of lines 3–4 always terminates and establishes whether $\mathcal{R}$ is overly general w.r.t. E^p. In particular, when this test succeeds, we know that there exists a (finite) leftmost, innermost rewrite sequence which proves e^p in $\mathcal{R}$ for each $e^p \in E^p$.

Now, according to Proposition 70, the unfolding phase (lines 5–11) terminates delivering a program $\mathcal{R}_k$ such that every negative example e^n succeeds in $\mathcal{R}_k$ by means of (at least) one rule which is not used in the proof of any positive example.

On the other hand, since the semantics of $\mathcal{R}$ is preserved by unfolding w.r.t. *inn* and $\mathcal{R} \in \mathbb{R}^u \subseteq \mathbb{R}_{inn}$ [16], program, $\mathcal{R}_k \vdash_{inn} E^p$. Moreover, $\mathcal{R}_k \vdash_{out} E^p$, since $\mathcal{R} \in \mathbb{R}^u$ and the *out* narrowing strategy is complete w.r.t. the class $\mathbb{R}^u$ [60].

Subsequently, the deletion phase (lines 12–15) removes all, and only those rules $r \in \mathcal{R}_k$ which do not appear in the proofs of the positive examples. Hence, a specialized program $\mathcal{R}^c$ is computed such that $\mathcal{R}^c \not\vdash_\varphi E^n$, $\varphi \in \{inn, out\}$, while it still succeeds on the whole E^p, which gives the desired result. □

Appendix C. Auxiliary information of the debugging session of Section 8.1

This appendix presents part of the log file that has been generated as a by-product of the debugging session of Section 8.1. Specifically, we show the under- and over-approximations along with the sets of positive and negative examples generated by the debugger. Variables are specified by using notation _n, where n is a natural number. The over-approximation I+ has been generated by computing the fixpoint of the abstract immediate consequence operator (it took only two iterations to reach the fixpoint). The under-approximation I- has been obtained by computing two iterations of the concrete immediate consequence operator. Both computations took less than 1 s on a Macbook Air 2.13 GHz Intel Core 2 Duo . Positive and negative example sets have been generated by applying an optimized version of the example generation procedure of Section 7.2.1 which increases the number of examples produced without the need of increasing the size of the under-approximation (which is typically one of the most time-consuming tasks of the whole debugging process). Specifically, our optimization allows us to generate ground examples even from non-ground equations by first instantiating them and then normalizing their right-hand sides.

```
************************************************
*** OVER APPROXIMATION I+
************************************************
playdice(s(s(0))) = s(_16184).
playdice(s(0)) = s(_16159).
playdice(s(0)) = sum(s(0),s(0)).
playdice(s(s(0))) = sum(s(s(0)),s(s(0))).
dd(s(_16054)) = s(_16064).
dd(0) = 0.
playdice(s(s(0))) = dd(s(s(0))).
playdice(s(0)) = dd(s(0)).
sum(_15959,s(_15947)) = s(_15957).
sum(_15935,0) = _15935.
dd(_15910) = sum(_15910,_15910).
winface(s(s(0))) = s(s(0)).
winface(s(0)) = s(0).
playdice(_15836) = dd(winface(_15836)).
sum(_15806,_15807) = sum(_15806,_15807).
0 = 0.
s(_15780) = s(_15780).
playdice(_15767) = playdice(_15767).
winface(_15752) = winface(_15752).
dd(_15737) = dd(_15737).
************************************************
************************************************
*** POSITIVE EXAMPLES
************************************************
playdice(s(s(0))) = s(s(s(s(0)))).
dd(s(s(s(s(0))))) = s(s(s(s(s(s(s(s(0))))))))).
dd(s(s(s(s(0))))) = s(s(s(s(s(s(s(0))))))).
playdice(s(0)) = s(s(0)).
dd(s(s(0))) = s(s(s(s(0)))).
dd(0) = 0.
dd(s(0)) = s(s(0)).
winface(s(s(0))) = s(s(0)).
winface(s(0)) = s(0).
```

```
************************************************
*** UNDER APPROXIMATION I-
************************************************
dd(s(_42043)) = s(sum(s(_42043),_42043)).
sum(_42010,s(s(_42002))) = s(s(sum(_42010,_42002))).
dd(0) = 0.
sum(_41942,s(0)) = s(_41942).
playdice(s(s(0))) = dd(s(s(0))).
playdice(s(0)) = dd(s(0)).
sum(_41854,s(_41850)) = s(sum(_41854,_41850)).
sum(_41817,0) = _41817.
dd(_41792) = sum(_41792,_41792).
winface(s(s(0))) = s(s(0)).
winface(s(0)) = s(0).
playdice(_41718) = dd(winface(_41718)).
sum(_41688,_41689) = sum(_41688,_41689).
0 = 0.
s(_41662) = s(_41662).
playdice(_41649) = playdice(_41649).
winface(_41634) = winface(_41634).
dd(_41619) = dd(_41619).
************************************************

************************************************
*** NEGATIVE EXAMPLES
************************************************
winface(0) = 0.
playdice(0) = 0.
winface(s(s(s(0)))) = s(s(s(0))).
winface(s(s(s(s(0))))) = s(s(s(s(0)))).
playdice(s(s(s(s(0))))) = s(s(s(s(s(s(0)))))).
playdice(s(s(s(s(0))))) = s(s(s(s(s(s(s(s(0)))))))).
```

References

[1] Y. Ajiro, K. Ueda, Kima — an automated error correction system for concurrent logic programs, in: Automated Software Engineering, vol. 19, 2002, pp. 67–94.

[2] Y. Ajiro, K. Ueda, K. Cho, Error correcting source code, in: Proc. Int'l Conf. on Principle and Practice of Constraint Programming, in: LNCS, vol. 1520, Springer, 1998, pp. 40–54.

[3] Z. Alexin, T. Gyimóthy, H. Bostrom, Integrating algorithmic debugging and unfolding transformation in an interactive learner, in: Proc. European Conf. on Artificial Intelligence, ECAI'96, John Wiley & Sons, 1996.

[4] M. Alpuente, D. Ballis, F.J. Correa, M. Falaschi, Automated correction of functional logic programs, in: P. Degano (Ed.), Proc. European Symp. on Programming ESOP 2003, in: Springer LNCS, vol. 2618, 2003, pp. 54–68.

[5] M. Alpuente, D. Ballis, M. Falaschi, Transformation and debugging of functional logic programs, in: A. Dovier, E. Pontelli (Eds.), 25 Years of Logic Programming in Italy, in: Springer LNCS, vol. 6125, 2010, pp. 271–299.

[6] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, A compact fixpoint semantics for term rewriting systems, Theoretical Computer Science 411 (37) (2010) 3348–3371.

[7] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, S. Lucas, Abstract diagnosis of functional programs, in: M. Leuschel, F. Bueno (Eds.), Proc. LOPSTR 2002, in: Springer LNCS, vol. 2664, 2003, pp. 1–16.

[8] M. Alpuente, F. Correa, M. Falaschi, Declarative debugging of functional logic programs, in: B. Gramlich, S. Lucas (Eds.), Proc. Int'l Workshop on Reduction Strategies in Rewriting and Programming, WRS 2001, in: Elsevier ENTCS, vol. 57, 2001.
[9] M. Alpuente, F. Correa, M. Falaschi, A debugging scheme for functional logic programs, in: M. Hanus (Ed.), Proc. 10th Int'l Workshop on Functional and (Constraint) Logic Programming, WFLP 2001, in: Elsevier ENTCS, vol. 64, 2002.
[10] M. Alpuente, F. Correa, M. Falaschi, 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.
[11] M. Alpuente, S. Escobar, J. Iborra, Termination of narrowing with dependency pairs, in: Maria Garcia de la Banda, Enrico Pontelli (Eds.), Proceedings of the 24th International Conference on Logic Programming, ICLP'08, in: Lecture Notes in Computer Science, vol. 5366, 2008, pp. 317–331.
[12] M. Alpuente, S. Escobar, J. Meseguer, P. Ojeda, Order-sorted generalization, Electronic Notes in Theoretical Computer Science 246 (2009) 27–38.
[13] M. Alpuente, M. Falaschi, P. Julián, G. Vidal, Uniform lazy narrowing, Journal of Logic and Computation 13 (2) (2003) 287–312.
[14] M. Alpuente, M. Falaschi, F. Manzo, Analyses of inconsistency for incremental equational logic programming, in: M. Bruynooghe, M. Wirsing (Eds.), Proc. PLILP'92, Leuven (Belgium), in: Springer LNCS, vol. 631, 1992, pp. 443–457.
[15] M. Alpuente, M. Falaschi, F. Manzo, Analyses of unsatisfiability for equational logic programming, Journal of Logic Programming 22 (3) (1995) 221–252.
[16] M. Alpuente, M. Falaschi, G. Moreno, G. Vidal, Rules + strategies for transforming lazy functional logic programs, Theoretical Computer Science 311 (1–3) (2004) 479–525.
[17] M. Alpuente, M. Falaschi, M.J. Ramis, G. Vidal, A compositional semantics for conditional term rewriting systems, in: H.E. Bal (Ed.), Proc. 6th IEEE Int'l Conf. on Computer Languages, ICCL'94, IEEE, New York, 1994, pp. 171–182.
[18] M. Alpuente, M. Falaschi, G. Vidal, A compositional semantic basis for the analysis of equational horn programs, Theoretical Computer Science 165 (1) (1996) 97–131.
[19] M. Alpuente, M. Falaschi, G. Vidal, Partial evaluation of functional logic programs, ACM Transactions on Programming Languages and Systems 20 (4) (1998) 768–844.
[20] S. Antoy, Definitional trees, in: Proc. the 3rd Int'l Conf. on Algebraic and Logic Programming, ALP'92, in: Springer LNCS, vol. 632, 1992, pp. 143–157.
[21] S. Antoy, Evaluation strategies for functional logic programming, in: Int'l Workshop on Reduction Strategies in Rewriting and Programming WRS'01, in: Elsevier ENTCS, vol. 57, 2001.
[22] S. Antoy, R. Echahed, M. Hanus, A needed narrowing strategy, Journal of the ACM 47 (4) (2000) 776–822.
[23] S. Antoy, M. Hanus, Functional logic programming, Communications of the ACM 53 (4) (2010) 74–85.
[24] S. Antoy, S. Johnson, TeaBag: a functional logic debugger, in: H. Kuchen (Eds.), Proc. 13th Int'l Workshop on Functional and (Constraint) Logic Programming, WFLP 2004, 2004, pp. 4–18.
[25] P. Arenas, A. Gil, A debugging model for lazy functional languages, Technical Report DIA 94/6, Universidad Complutense de Madrid, April 1994.
[26] T. Arts, J. Giesl, Termination of Term Rewriting using Dependency Pairs, Theoretical Computer Science 236 (1–2) (2000) 133–178.
[27] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge U. Press, 1998.
[28] G. Bacci, M. Comini, Abstract diagnosis of first order functional logic programs, in: M. Alpuente (Eds.), Pre-Proc. LOPSTR 2010, RISC Technical Report 10–14, 2010, pp. 58–72.
[29] D. Bert, R. Echahed, Design and implementation of a generic, logic and functional programming language, in: Proc. ESOP'86, in: Springer LNCS, vol. 213, 1986, pp. 119–132.
[30] D. Bert, R. Echahed, On the operational semantics of the algebraic and logic programming language LPG, in: COMPASS/ADT, in: Lecture Notes in Computer Science, vol. 906, Springer, 1994, pp. 132–152.
[31] D. Bert, R. Echahed, Abstraction of conditional term rewriting systems, in: Proc. ILPS'95, MIT Press, 1995, pp. 162–176.
[32] D. Bert, R. Echahed, On the operational semantics of the algebraic and logic programming language LPG, in: Recent Trends in Data Type Specifications, in: Springer LNCS, vol. 906, 1995, pp. 132–152.
[33] D. Bert, R. Echahed, K. Adi, Resolution of goals with the functional and logic programming language LPG: impact of abstract interpretation, in: Proc. AMAST'96, in: Springer LNCS, vol. 1101, 1996, pp. 629–632.
[34] A. Bockmayr, A. Werner, LSE narrowing for decreasing conditional term rewrite systems, in: Conditional Term Rewriting Systems, CTRS'94, Jerusalem, in: Springer LNCS, vol. 968, 1995.
[35] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, ELAN from a rewriting logic point of view, Theoretical Computer Science 285 (2) (2002) 155–185.
[36] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, An overview of Elan, Electronic Notes in Theoretical Computer Science 15 (1998).
[37] P. Bosco, E. Giovannetti, C. Moiso, Narrowing vs. SLD-resolution, Theoretical Computer Science 59 (1988) 3–23.
[38] A. Bossi, M. Gabbrielli, G. Levi, M. Martelli, The s-semantics approach: theory and applications, Journal of Logic Programming 19–20 (1994) 149–197.
[39] H. Bostrom, P. Idestam-Alquist, Specialization of logic programs by pruning sld-trees, in: Proc of 4th Int'l Workshop on Inductive Logic Programming, ILP-94, 1994, pp. 31–47.
[40] H. Bostrom, P. Idestam-Alquist, Induction of logic programs by example-guided unfolding, Journal of Logic Programming 40 (1999) 159–183.
[41] B. Braßel, O. Chitil, M. Hanus, F. Huch, Observing functional logic computations, in: Proc. PADL'04, in: Springer LNCS, vol. 3057, 2004, pp. 193–208.
[42] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M Hermenegildo, J. Maluszyński, G. Puebla, On the role of semantic approximations in validation and diagnosis of constraint logic programs, in: Proc. 3rd. Int'l Workshop on Automated Debugging, AADEBUG'97, U. of Linkoping, 1997, pp. 155–170.
[43] R. Caballero, N. Martí-Oliet, A. Riesco, A. Verdejo, A declarative debugger for Maude functional modules, in: Proc. WRLA 2008, in: Elsevier ENTCS, 2008.
[44] R. Caballero, M. Rodríguez-Artalejo, R. del Vado Vírseda, Declarative diagnosis of missing answers in constraint functional-logic programming, in: Jacques Garrigue, Manuel V. Hermenegildo (Eds.), Functional and Logic Programming, 9th International Symposium, FLOPS 2008, Ise, Japan, April 14–16, 2008. Proceedings, in: Lecture Notes in Computer Science, vol. 4989, Springer, 2008, pp. 305–321.
[45] R. Caballero-Roldán, M. Rodríquez Artalejo, A declarative debugging system for lazy functional logic programs, in: Proc. WFLP 2001, in: Elsevier ENTCS, vol. 57, 2001.
[46] R. Caballero-Roldán, F.J. López-Fraguas, M. Rodríquez Artalejo, Theoretical foundations for the declarative debugging of lazy functional logic programs, in: Fifth Int'l Symp. on Functional and Logic Programming, in: Springer LNCS, vol. 2024, 2001, pp. 170–184.
[47] R. Caballero-Roldán, F.J. López-Fraguas, M. Rodríquez Artalejo, Declarative Debugging for Encapsulated search, in: Proc. 11th Int'l Workshop on Functional and (Constraint) Logic Programming, WFLP 2002, in: Elsevier ENTCS, vol. 76, 2002.
[48] M. Cameron, M. García de la Banda, K. Marriott, P. Moulder, Vimer: a visual debugger for mercury, in: Proc. 5th ACM SIGPLAN Int'l Conf. on Principles and Practice of Declaritive Programming, ACM Press, 2003, pp. 56–66.
[49] K. Claessen, J. Hughes, QuickCheck: a lightweight tool for random testing of haskell programs, ACM SIGPLAN Notices 35 (9) (2000) 268–279.
[50] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, Jo. Meseguer, C.L. Talcott, Unification and narrowing in Maude 2.4, in: Ralf Treinen (Ed.), Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29–July 1, 2009, Proceedings, in: Lecture Notes in Computer Science, vol. 5595, Springer-Verlag, 2009, pp. 380–390.
[51] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude — A High-Performance Logical Framework, Springer-Verlag, New York, 2007.
[52] M. Codish, M. Falaschi, K. Marriott, Suspension analysis for concurrent logic programs, in: K. Furukawa (Ed.), Proc. Eighth Int'l Conf. on Logic Programming, The MIT Press, Cambridge, MA, 1991, pp. 331–345.
[53] M. Comini, R. Gori, G. Levi, Assertion based inductive verification methods for logic programs, in: A.K. Seda (Ed.), Proc. MFCSIT'2000, in: Elsevier ENTCS, vol. 40, 2001.
[54] M. Comini, G. Levi, M.C. Meo, G. Vitiello, Abstract diagnosis, Journal of Logic Programming 39 (1–3) (1999) 43–93.
[55] M. Comini, G. Levi, G. Vitiello, Declarative diagnosis revisited, in: Proc. 1995 Int'l Symp. on Logic Programming, The MIT Press, 1995, pp. 275–287.

[56] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proc. Fourth ACM Symp. on Principles of Programming Languages, 1977, pp. 238–252.
[57] D. DeGroot, G. Lindstrom (Eds.), Logic Programming, Functions, Relations and Equations, Prentice Hall, Englewood Cliffs, NJ, 1986.
[58] P. Deransart, M.V. Hermenegildo, J. Maluszynski, Debugging of constraint programs: the DiSCiPl methodology and tools, in: Analysis and Visualization Tools for Constraint Programming, in: Lecture Notes in Computer Science, vol. 1870, Springer, 2000, pp. 1–20.
[59] N. Dershowitz, U. Reddy, Deductive and inductive synthesis of equational programs, Journal of Symbolic Computation 15 (1993) 467–494.
[60] R. Echahed, On completeness of narrowing strategies, in: Proc. CAAP'88, in: Springer LNCS, vol. 299, 1988, pp. 89–101.
[61] R. Echahed, Uniform narrowing strategies, in: Proc. ICALP'92, in: Springer LNCS, vol. 632, 1992, pp. 259–275.
[62] S. Escobar, J. Meseguer, R. Sasse, Effectively checking or disproving the finite variant property, in: Proc. RTA'08. Springer LNCS, vol. 5117, 2008, pp. 79–93.
[63] S. Escobar, J. Meseguer, R. Sasse, Variant narrowing and equational unification, in: Proc. 7th Int'l Workshop on Rewriting Logic and its Applications, WRLA08, in: Elsevier ENTCS, 2008.
[64] M. Falaschi, G. Levi, M. Martelli, C. Palamidessi, A new declarative semantics for logic languages, in: R. Kowalski, K. Bowen (Eds.), Proc. Fifth Int'l Conf. on Logic Programming, The MIT Press, Cambridge, MA, 1988, pp. 993–1005.
[65] M. Falaschi, G. Levi, M. Martelli, C. Palamidessi, Declarative modeling of the operational behavior of logic languages, Theoretical Computer Science 69 (3) (1989) 289–318.
[66] M. Falaschi, G. Levi, M. Martelli, C. Palamidessi, A model-theoretic reconstruction of the operational semantics of logic programs, Information and Computation 103 (1) (1993) 86–113.
[67] M. Fay, First order unification in an equational theory, in: Proc of 4th Int'l Conf. on Automated Deduction, CADE'79, 1979, pp. 161–167.
[68] G. Ferrand, Error diagnosis in logic programming, and adaptation of E.Y.Shapiro's method, Journal of Logic Programming 4 (3) (1987) 177–198.
[69] C. Ferri, J. Hernández, M.J. Ramírez, The FLIP system homepage, 2000. Available at: http://www.dsic.upv.es/users/elp/soft.html.
[70] C. Ferri, J. Hernández, M.J. Ramírez, Incremental learning of functional logic programs, in: 5th Int'l Symp. on Functional and Logic Programming, FLOPS'01, in: Springer LNCS, vol. 2024, 2001, pp. 233–247.
[71] L. Fribourg, SLOG: a logic programming language interpreter based on clausal superposition and rewriting, in: Proc. Second IEEE Int'l Symp. on Logic Programming, IEEE, New York, 1985, pp. 172–185.
[72] D. Frutos-Escrig, M.I. Fernández-Camacho, On narrowing strategies for partial non-strict functions, in: S. Abramsky, T. Maibaum (Eds.), Proc. TAPSOFT'01, in: Springer LNCS, vol. 493, 1991, pp. 416–437.
[73] J. Giesl, T. Arts, Verification of erlang processes by dependency pairs, Applicable Algebra in Engineering, Communication and Computing 12 (1/2) (2001) 39–72.
[74] A. Gill, Debugging haskell by observing intermediate data structures, in: Proceedings 2000 ACM SIGPLAN Haskell Workshop, Electronic Notes in Theoretical Computer Science 41 (1) (2001) 1.
[75] E. Giovannetti, G. Levi, C. Moiso, C. Palamidessi, Kernel leaf: a logic plus functional language, Journal of Computer and System Sciences 42 (1991) 363–377.
[76] B. Gramlich, Sufficient conditions for modular termination of conditional term rewriting systems, in: Conditional Term Rewriting Systems, Third International Workshop, CTRS-92, in: Lecture Notes in Computer Science, vol. 656, Springer, 1993, pp. 128–142.
[77] M. Hanus, Compiling logic programs with equality, in: Proc. 2nd Int'l Workshop on Programming Language Implementation and Logic Programming, in: Springer LNCS, vol. 456, 1990, pp. 387–401.
[78] M. Hanus, The integration of functions into logic programming: from theory to practice, Journal of Logic Programming 19, 20 (1994) 583–628.
[79] M. Hanus, Multi-paradigm declarative languages (invited tutorial), in: Proc. of International Conference on Logic Programming, ICLP 2007, in: Lecture Notes in Computer Science, vol. 4670, Springer, 2007, pp. 45–75.
[80] M. Hanus, Call pattern analysis for functional logic programs, in: Sergio Antoy, Elvira Albert (Eds.), Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15–17, 2008, Valencia, Spain, ACM, 2008, pp. 67–78.
[81] M. Hanus, B. Josephs, A debugging model for functional logic programs, in: Proc. PLILP'93, in: Springer LNCS, vol. 714, 1993, pp. 28–43.
[82] M. Hanus, J. Koj, An integrated development environment for declarative multi-paradigm programming, in: Proc. WLPE'01, Paphos, Cyprus, 2001, pp. 1–14.
[83] M. Hanus, H. Kuchen, J.J. Moreno-Navarro, Curry: a truly functional logic language, in: Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming, 1995, pp. 95–107.
[84] M. Hanus, C. Prehofer, Higher-order narrowing with definitional trees, Journal of Functional Programming 9 (1) (1999) 33–75.
[85] Haskell debugging technologies. At http://www.haskell.org/debugging/, October 2008.
[86] J. Hernández, M.J. Ramírez, Induction of functional logic programs, in: Proc. of the JICSLP'98 CompulogNet Area Meeting on Computational Logic and Machine Learning, 1998, pp. 49–55.
[87] J. Hernández, M.J. Ramírez, A strong complete schema for inductive functional logic programming, in: Proc. Ninth Int'l Workshop on Inductive Logic Programming, ILP'99, in: Springer LNAI, vol. 1634, 1999, pp. 116–127.
[88] S. Hölldobler, From paramodulation to narrowing, in: Proc. 5th Int'l Conf. on Logic Programming, Seattle, 1988, pp. 327–342.
[89] S. Hölldobler, Foundations of Equational Logic Programming, in: Springer LNAI, vol. 353, 1989.
[90] J.M. Hullot, Canonical forms and unification, in: Proc of 5th Int'l Conf. on Automated Deduction, in: Springer LNCS, vol. 87, 1980, pp. 318–334.
[91] C. Kirchner, H. Kirchner, A. Santana de Oliveira, Analysis of rewrite-based access control policies, in: Proc. 3rd Int'l Workshop on Security and Rewriting Techniques, SecreT 2008, in: Elsevier ENTCS, 2008.
[92] J.W. Klop, Term Rewriting Systems, in: S. Abramsky, D. Gabbay, T. Maibaum (Eds.), Handbook of Logic in Computer Science, vol. I, Oxford University Press, 1992, pp. 1–112.
[93] G. Levi, C. Palamidessi, P.G. Bosco, E. Giovannetti, C. Moiso, A complete semantics caracterization of K-LEAF, a logic language with partial functions, in: Proc. Second IEEE Symp. on Logic In Computer Science, IEEE, New York, 1987, pp. 318–327.
[94] J.W. Lloyd, Debugging for a declarative programming language, Machine Intelligence 15 (1998).
[95] J.W. Lloyd, Foundations of Logic Programming, Springer-Verlag, Berlin, 1984.
[96] F.J. López-Fraguas, J. Sánchez Hernández, Toy: a multiparadigm declarative system, in: Proc. 10th Int'l Conf. on Rewriting Techniques and Applications, Springer-Verlag, 1999, pp. 244–247.
[97] M.J. Maher, Complete axiomatizations of the algebras of finite, rational and infinite trees, in: Proc. Third IEEE Symp. on Logic In Computer Science, Computer Science Press, New York, 1988, pp. 348–357.
[98] M.J. Maher, Equivalences of logic programs, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, Los Altos, CA, 1988, pp. 627–658.
[99] M.J. Maher, On parameterized substitutions, Technical Report RC 16042, IBM - T.J. Watson Research Center, Yorktown Heights, NY, 1990.
[100] S. Marlow, J. Iborra, B. Pope, A. Gill, A lightweight interactive debugger for haskell, in: Gabriele Keller (Ed.), Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007, ACM, 2007, pp. 13–24.
[101] N. Martí-Oliet, A. Riesco, A. Verdejo, Declarative debugging of missing answers for Maude specifications, in: Proc. RTA 2010, in: Leibniz International Proceedings in Informatics, vol. 6, 2010, pp. 277–294.
[102] J. Meseguer, Multiparadigm logic programming, in: H. Kirchner, G. Levi (Eds.), Algebraic and Logic Programming, Proc. the Third Int'l Conf, ALP'92, Berlin, in: Springer LNCS, vol. 632, 1992, pp. 158–200.
[103] J. Meseguer, P. Thati, Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols, Higher-Order and Symbolic Computation 20 (1–2) (2007) 123–160.

[104] A. Middeldorp, E. Hamoen, Counterexamples to completeness results for basic narrowing, in: H. Kirchner, G. Levi (Eds.), Proc. Third Int'l Conf. on Algebraic and Logic Programming, in: Springer LNCS, vol. 632, 1992, pp. 244–258.
[105] J.J. Moreno-Navarro, M. Rodríguez-Artalejo, Logic programming with functions and predicates: the language Babel, Journal of Logic Programming 12 (3) (1992) 191–224.
[106] S. Muggleton, Inductive logic programming, New Generation Computing 8 (3) (1991) 295–318.
[107] S. Muggleton, L. de Raedt, Inductive logic programming: theory and methods, Journal of Logic Programming 19, 20 (1994) 629–679.
[108] L. Naish, A declarative debugging scheme, Journal of Functional and Logic Programming 1997 (3) (1997).
[109] L. Naish, T. Barbour, Towards a portable lazy functional declarative debugger, Australian Computer Science Communications 18 (1) (1996) 401–408.
[110] P. Padawitz, Computing in Horn Clause Theories, in: EATCS Monographs on Theoretical Computer Science, vol. 16, Springer-Verlag, Berlin, 1988.
[111] A. Pettorossi, M. Proietti, Transformation of logic programs: foundations and techniques, Journal of Logic Programming 19, 20 (1994) 261–320.
[112] A. Pettorossi, M. Proietti, Transformation of logic programs, in: Handbook of Logic in Artificial Intelligence, vol. 5, Oxford University Press, 1998, pp. 697–787.
[113] U.S. Reddy, Narrowing as the operational semantics of functional languages, in: Proc. Second IEEE Int'l Symp. on Logic Programming, IEEE, New York, 1985, pp. 138–151.
[114] U.S. Reddy, Bridging the gap between logic and functional programming, in: Proc. ILPS'95, MIT Press, 1995, pp. 627–628.
[115] T.W. Reps, T. Turnidge, Program specialization via program slicing, in: O. Danvy, R. Glück, P. Thiemann (Eds.), Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12–16, 1996, Selected Papers, in: Lecture Notes in Computer Science, vol. 1110, Springer, 1996, pp. 409–429.
[116] F. Schernhammer, B. Gramlich, Characterizing and proving operational termination of deterministic conditional term rewriting systems, Journal of Logic and Algebraic Programming, 2010 (in press).
[117] E.Y. Shaphiro, Algorithmic Program Debugging, The MIT Press, Cambridge, Massachusetts, 1982, ACM Distinguished Dissertation.
[118] J.R. Slagle, Automated theorem-proving for theories with simplifiers, commutativity and associativity, Journal of the ACM 21 (4) (1974) 622–642.
[119] H. Tamaki, Semantics of a logic programming language with a reducibility predicate, in: Proc. First IEEE Int'l Symp. on Logic Programming, IEEE Press, 1984, pp. 259–264.
[120] K. Ueda, M. Morita, Moded flat GHC and its message-oriented implementation technique, New Generation Computing 13 (1) (1994) 3–43.
[121] P. Wadler, An angry half-dozen, SIGPLAN Notices 33 (2) (1998) 25–30.
[122] F. Zartmann, Denotational abstract interpretation of functional logic programs, in: Proc. SAS'97, in: Springer LNCS, vol. 1302, 1997, pp. 141–159.