Transformation and Debugging of Functional Logic Programs*

M. Alpuente[1], D. Ballis[2], and M. Falaschi[3]

[1] DSIC, Universidad Politécnica de Valencia
Camino de Vera s/n, Apdo. 22012, 46071 Valencia, Spain.
alpuente@dsic.upv.es
[2] Dip. Matematica e Informatica
Via delle Scienze 206, 33100 Udine, Italy.
demis@dimi.uniud.it
[3] Dip. di Scienze Matematiche e Informatiche
Pian dei Mantellini 44, 53100 Siena, Italy.
moreno.falaschi@unisi.it

Abstract. The Italian contribution to functional-logic programming has been significant and influential in a number of areas of semantics, and semantics-based program manipulation techniques. We survey selected topics, with a particular regard to debugging and transformation techniques. These results as usual depend on the narrowing strategy which is adopted and on the properties satisfied by the considered programs. In this paper, we restrict ourselves to first-order functional-logic languages without non-deterministic functions. We start by describing some basic classical transformation techniques, namely folding and unfolding. Then, we recall the narrowing-driven partial evaluation, which is the first generic algorithm for the specialization of functional logic programs. Regarding debugging, we describe a goal-independent approach to automatic diagnosis and correction which applies the immediate consequence operator modeling computed answers to the diagnosis of bugs in functional logic programs. A companion bug-correction program synthesis methodology is described that attempts to correct the erroneous components of the wrong code.

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 integrated languages with a complete semantics is usually based on *narrowing* [37], which consists of the instan-

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

tiation 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 [41, 43] for a survey.)

Functional logic programming is an area which was pioneered by Italian researchers. For instance, the first published survey paper on this subject was [22]. Since then, the Italian contribution has been significant and influential in a number of semantics-based program manipulation techniques. The main purpose of this work is to outline a selection of these techniques, with a particular regard to debugging and transformation. Actually, good programs have to be both correct (w.r.t. a given specification) and efficient, but these two aspects are often in delicate balance.

Program transformations provide a methodology for deriving correct and possibly efficient programs. We recall first a simple transformation methodology based on fold/unfold techniques [12, 13]. Then we recall the narrowing-driven partial evaluation, which was first proposed in [16] and is the first generic algorithm for the specialization of functional logic programs. Regarding program debugging, we offer an up-to-date, comprehensive, and uniform presentation of the declarative debugging of functional logic programs as developed in [6, 7]. 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 considered narrowing strategy, which can be either eager or lazy. 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 by means of inductive learning, without asking the user to answer difficult questions about program semantics as typically happens in algorithmic debugging. A further important advantage of our method is the fact that we develop a finite methodology which is also goal-independent and allows us to perform diagnosis statically. 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 correction technique is driven by a set of evidence examples that are automatically produced as an outcome by the diagnoser, and infers the program corrections by combining top-down (unfolding-based) transformations with a bottom-up (induction-based) program synthesis methodology. Due to the strong relation between program transformation and program synthesis, the cooperation between these two methodologies within the debugging framework is fruitful and extremely smooth.

We do not consider in this paper programs containing non-strict, non-deterministic functions with call-time choice semantics [58, 59], as adopted by some modern functional logic languages like Curry [42, 46] or Toy [56]. This is because there does not exist a simple and adequate notion of narrowing for call-time

choice that can replace existing efficient versions of narrowing like the strategies discussed in this paper, which are well established and appropriate operational procedures for functional logic languages [58].

All the proposed transformation and verification frameworks have been implemented into prototypical systems which have been thoroughly evaluated using large suites of benchmarks in order to assess their usefulness experimentally. Tools and experiments are freely available at the URL:

`http://users.dsic.upv.es/grupos/elp/soft.html`.

Plan of the paper. The rest of the paper is organized as follows. Section 2 presents some preliminary basic definitions. In Section 3, we formalize narrowing along with two well-know narrowing strategies: the leftmost-innermost (*inn*) and the leftmost-outermost (*out*) narrowing strategy. We then formulate both an operational semantics and a fixpoint semantics for functional logic programs which are parametric w.r.t. the chosen narrowing strategy. We also show the correspondence between the two program denotations. Section 4 outlines the rudiments of functional logic program transformation, while Section 5 focuses on the narrowing-driven approach to functional logic program specialization. Section 6 formalizes the diagnosis framework by providing the necessary notions of rule incorrectness and uncoveredness, and describes an effective methodology based on abstract interpretation that can be used to implement declarative debuggers. Moreover, we present a bug-correction program synthesis methodology which, after diagnosing the buggy program, tries to correct the erroneous components of the wrong code automatically. Finally, in Section 7 we discuss some related work.

2 Preliminaries

Let us briefly recall some known results about rewrite systems [51] and functional logic programming (see [41, 47] for extensive surveys). For simplicity, definitions are given in the one-sorted case. The extension to many-sorted signatures is straightforward, see [66].

Throughout this paper, V denotes 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. 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 ground term algebra built on $\Sigma \cup V$ and Σ, respectively. An *equation* is a syntactic expression of the form $t = t'$, where $t, t' \in \tau(\Sigma \cup V)$.

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. $O(t)$ (resp. $\overline{O}(t)$) denotes the set of positions (resp. non-variable 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 non-variable position set of a sequence of equations $g = (t_1 = t'_1, \ldots, t_n = t'_n)$ can be defined as follows: $\overline{O}(g) = \{i.1.u \mid i \in \{1, \ldots, n\}, u \in \overline{O}(t_i)\} \bigcup \{i.2.u \mid i \in \{1, \ldots, n\}, u \in \overline{O}(t'_i)\}$.

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. Syntactic equality is denoted by $=$.

A *substitution* is a mapping from the set of variables V into the set of terms $\tau(\Sigma \cup V)$. We write $\theta_{\restriction s}$ to denote the restriction of the substitution θ to the set of variables in the syntactic object s. The *empty substitution* is denoted by *id*. Composition of substitutions is denoted by juxtaposition, with identity element *id*. A substitution θ is more general than σ, denoted by $\theta \leq \sigma$, if $\sigma = \theta\gamma$ for some substitution γ. We say that a substitution σ is a *unifier* of two terms t and t' if $t\sigma = t'\sigma$. We let $mgu(t, t')$ denote a *most general unifier* of t and t'.

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 $(\lambda \to \rho \Leftarrow C)$, λ, $\rho \in \tau(\Sigma \cup V)$ and $\lambda \notin V$. The condition C is a (possibly empty) sequence $e_1, \ldots, e_n$, $n \geq 0$ of equations. 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 an empty condition, we write $\lambda \to \rho$. A TRS is a CTRS whose rules have no conditions. A *goal* g is a non-empty sequence of equations $\Leftarrow C$, i.e., a rule with no head (consequent). Sometimes we leave out the $\Leftarrow$ symbol when we write goals.

For CTRS $\mathcal{R}$, $r \ll \mathcal{R}$ denotes that r is a new variant of a rule in $\mathcal{R}$ such that r contains only *fresh* variables, i.e. contains no variable previously met during computation (standardized apart). Given a CTRS $(\Sigma, \mathcal{R})$, we assume that the signature Σ is partitioned into two disjoint sets $\Sigma = \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{D} = \{f \mid (f(t_1, \ldots, t_n) \to r \Leftarrow C) \in \mathcal{R}\}$ and $\mathcal{C} = \Sigma \setminus \mathcal{D}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions*. The elements of $\tau(\mathcal{C} \cup \mathcal{V})$ are called *constructor terms*. A *constructor* substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ 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 *pattern* is a term of the form $f(\bar{d})$ where $f/n \in \mathcal{D}$ and $\bar{d}$ are constructor terms. We say that a CTRS is *constructor-based* (CB) if the left-hand sides of $\mathcal{R}$ are patterns.

A rewrite step is the application of a rewrite rule to an expression. A term s *conditionally rewrites* to a term t, $s \to_{\mathcal{R}} t$, if there exist $u \in \overline{O}(s)$, $(\lambda \to \rho \Leftarrow s_1 = t_1, \ldots, s_n = t_n) \in \mathcal{R}$, and substitution σ such that $s|_u = \lambda\sigma$, $t = s[\rho\sigma]_u$, and for all $i \in \{1, \ldots, n\}$ 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 $\to_{\mathcal{R}}$. The term $s|_u$ is said to be a *redex* of s. When no confusion can arise, we omit the subscript $\mathcal{R}$. A term s is a *normal form*, if there is no term t with $s \to_{\mathcal{R}} t$. A CTRS $\mathcal{R}$ is *strongly terminating* if there are no infinite sequences of the form $t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \ldots$ A CTRS $\mathcal{R}$ is *confluent* if, whenever a term s reduces to two terms t_1 and t_2,

both t_1 and t_2 reduce to the same common term. The program $\mathcal{R}$ is said to be canonical if the binary one-step rewrite relation $\rightarrow_{\mathcal{R}}$ defined by $\mathcal{R}$ is strongly terminating and confluent [51].

3 Evaluating Functional Logic Programs by Narrowing

Functional logic languages are extensions of functional languages with principles derived from logic programming [53, 68]. The computation mechanism of functional logic languages is based on *narrowing* [37], 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 in the sense that 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' in $\mathcal{R}$ if there exist a position $u \in \overline{O}(g)$, $r = (\lambda \rightarrow \rho \Leftarrow C) \ll \mathcal{R}$, and a substitution σ such that: $\sigma = mgu(g|_u, \lambda)$, and g' is the sequence $C\sigma, g[\rho]_u\sigma$.*

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

Basically, narrowing steps involve unification while functional reduction employs pattern matching. The condition that the binding substitution σ is a mgu can be relaxed to accomplish with certain narrowing strategies like needed narrowing [20], which use unifiers but not necessarily most general ones.

By using Definition 1, we can define (*successful*) narrowing derivations as follows. We use the symbol $\top$ to denote sequences of the form $true, \ldots, true$, and $\mathcal{R}_+$ denotes $\mathcal{R} \cup \{x = x \rightarrow true\}$, $x \in V$. Using this rule allows us to treat syntactical unification as a narrowing step, i.e., we use the rule $r = (x = x \rightarrow true)$ to compute mgu's: $s = t \overset{\Lambda,r,\sigma}{\leadsto} true$ 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 $\exists\theta_1, \ldots, \exists\theta_n.\ g \overset{\theta_1}{\leadsto} \ldots \overset{\theta_n}{\leadsto} g'$ and $\theta = \theta_1 \ldots \theta_n$, $n > 0$. A* successful *derivation for g in $\mathcal{R}$ is a narrowing derivation $g \overset{\theta}{\leadsto}{}^* \top$ in $\mathcal{R}_+$, and $\theta_{\restriction Var(g)}$ 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 CTRSs, including canonical CTRSs [48]. Similarly to logic programming, completeness means the ability to compute representatives of all solutions for one or more equations.

Example 1. 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 `nil` (empty list) and [_|_] (cons constructor)):

$$\begin{array}{lll} R1: & \texttt{last(xs)} & \rightarrow \texttt{y} \Leftarrow \texttt{append(zs,[y])} = \texttt{xs}. \\ R2: & \texttt{append(nil,xs)} & \rightarrow \texttt{xs}. \\ R3: & \texttt{append([x|xs],ys)} & \rightarrow \texttt{[x|append(ys,ys)]}. \end{array}$$

Given the input goal $\texttt{last(ys)} = \texttt{0}$, narrowing is able to compute in $\mathcal{R}$ infinitely many answers of the form $\{\texttt{ys} \mapsto \texttt{[0]}\}, \{\texttt{ys} \mapsto \texttt{[z|0]}\}, \ldots$ 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 and rule[4], and the reduced subterm is underlined):

$$\begin{array}{l} \underline{\texttt{last(ys)}} = \texttt{0} \leadsto_{\{\texttt{ys}\mapsto\texttt{xs}\},\texttt{R1}} \underline{\texttt{append(ws,[y])}} = \texttt{xs}, \texttt{y} = \texttt{0} \\ \qquad \leadsto_{\{\texttt{ws}\mapsto\texttt{nil}\},\texttt{R2}} (\texttt{[y] = xs}, \underline{\texttt{y} = \texttt{0}}) \\ \qquad \leadsto_{\{\texttt{y}\mapsto\texttt{0}\},(\texttt{x=x}\rightarrow\texttt{true})} (\mathit{true}, \underline{\texttt{[0] = xs}}) \\ \qquad \leadsto_{\{\texttt{xs}\mapsto\texttt{[0]}\},(\texttt{x=x}\rightarrow\texttt{true})} \top \end{array}$$

Moreover, without assuming canonicity, Meseguer and Thati showed that narrowing is still complete as a procedure to solve reachability problems [62] (that is, to find "more general" solutions σ for the variables of s and t such that $s\sigma$ rewrites to $t\sigma$ in a number of steps). Reachability problems extend narrowing capabilities to a wider spectrum that includes the analysis of concurrent systems. Narrowing has also received much attention due to the many other important applications, such as automated proofs of termination [21], verification of cryptographic protocols [33], equational constraint solving [10], partial evaluation [16], program transformation [14] and model checking [34], among others.

Narrowing Strategies Since unrestricted narrowing has quite a large search[5] space, several strategies to control the selection of redexes have been developed. A *narrowing strategy* (or *position constraint*) is any well-defined criterion which obtains a smaller search space by permitting narrowing to reduce only some chosen positions. A narrowing strategy φ can be formalized as a mapping that assigns a subset $\varphi(g)$ of $\overline{O}(g)$ to every 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 [73]), 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 $\lambda \rightarrow \rho \Leftarrow C$ satisfies $Var(\rho) \cup Var(C) \subset Var(\lambda)$, whereas the weaker condition $Var(\rho) \subset Var(\lambda) \cup Var(C)$ is commonly demanded in lazy programs. A survey of results about the completeness of narrowing strategies

[4] Substitutions are restricted to the input variables.

[5] Actually, there are three sources of non-determinism in narrowing: the choice of the equation within the goal, the choice of the redex within the equation, and the choice of the rewrite rule.

can be found in [19]. To simplify our notation, we let $I\!R_\varphi$ denote the class of programs that satisfy the conditions for the completeness of the strategy φ.

Throughout this paper, we focus our attention on two very common narrowing strategies: the *leftmost-innermost* and the *leftmost-outermost* narrowing strategies. More specifically, we let $inn(g)$ (resp. $out(g)$) denote the narrowing strategy which selects the position p of the leftmost-innermost (resp. leftmost-outermost) narrowing redex of g. [6]

We formulate a conditional narrower with strategy φ, $\varphi \in \{inn, out\}$, as the smallest relation $\leadsto_\varphi$ satisfying

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

For $\varphi \in \{inn, out\}$, $\mathcal{R}^\varphi_+ = \mathcal{R} \cup Eq^\varphi$, where the set of rules Eq^φ models the equality on terms.

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 [63]:

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

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 $\{\mathtt{x} \mapsto \mathtt{b}\}$ for $\mathtt{f(g(x),x)} = \mathtt{true}$ whereas outermost narrowing only computes $\{\mathtt{x} \mapsto \mathtt{a}\}$ for the considered goal $\mathtt{f(g(x),x)} \approx \mathtt{true}$. For the completeness of a narrowing strategy, the following *uniformity* condition is required [66]: a confluent program is uniform iff the position selected by φ is a valid narrowing position for φ for all normalized substitutions (i.e. substitutions that only contain terms in normal form) applied to it. Note that the program $\mathcal{R}$ above does not satisfy the uniformity principle since the top position of the term $\mathtt{f(g(x),x)}$ is not a valid narrowing position if we apply the substitution $\{\mathtt{x} \mapsto \mathtt{b}\}$ to this term. A sufficient condition for uniformity in constructor-based, canonical programs can

[6] The leftmost-innermost position of g is the leftmost position of g that points to a pattern. 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 $i < j$, where i, j are natural numbers and q, q' sequences of natural numbers.

be found in [32]. Moreover, there are methodologies which allow one to transform non-uniform programs into programs fulfilling the uniformity condition (e.g., see [9]).

Innermost narrowing is the foundation of several functional logic programming languages like SLOG [40], LPG [23] and (a subset of) ALF [41]. Also, the multi-paradigm language Maude [29] is equipped with a (kind of) innermost narrowing strategy (called *variant narrowing* [29]) that is part of an equational unification procedure. Moreover, reachability analyses for programs written in Maude rely on the so-called *topmost theories* [62], 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 [26]. 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 [26].

Modern functional logic languages like Curry [44] and Toy [56] 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 [43, 41]. Needed narrowing [20] 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, Needed narrowing [20] can be easily and efficiently implemented by means of a transformation proposed in [45], which permits leftmost outermost narrowing to be used on the transformed program while preserving the answers computed by needed narrowing in the original program. Thanks to the possibility to use 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.

Similarly to the other strategies discussed in this paper, needed narrowing adopts the classical theory of rewriting (that corresponds to run-time choice [49]) as underlying theory. However, in a run-time choice semantics, the values of the arguments are fixed as they are used, and the copies of the arguments created by parameter-passing may evolve independently afterwards [57]. Hence, classical rewriting is not valid for call-time choice evaluation, which is the operational semantics commonly adopted in functional logic languages dealing with non-strict, non-deterministic functions, and is related, at the operational level, to the sharing mechanism of lazy evaluation in functional languages [57, 58]. Nevertheless, by adding a sharing mechanism to their encoding, needed narrowing implementations are sound for the call-time choice semantics of functional logic programs (for a discussion, see [57, 58]). Moreover, for the deterministic programs considered in this paper, run-time and call-time are able to produce the same outcomes [58, 69].

3.1 Two Functional Logic Program Denotations

The operational semantics $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ of a functional logic program $\mathcal{R}$ w.r.t. the narrowing strategy $\varphi \in \{inn, out\}$ can be defined by considering all the possible successful narrowing derivations which can be obtained by applying the narrowing strategy φ to "most general calls". We denote by $\leadsto_{\varphi}$ the restriction of the narrowing relation that is obtained when the narrowing strategy φ is used.

Definition 3. *Let $\mathcal{R}$ be a program, $\varphi \in \{inn, out\}$. Then,*

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

where $\Im_{\mathcal{R}}$ denotes the set of the identical equations $c(x_1,\ldots,x_n) =_{\varphi} c(x_1,\ldots,x_n)$ for all the constructor symbols c/n occurring in $\mathcal{R}$.

It is known that the considered operational semantics can be derived by a fixpoint computation which allows for the (bottom-up) construction of a model that is completely goal-independent. To this respect, in [6, 7] we formalized a fixpoint semantics $\mathcal{F}_{\varphi}(\mathcal{R})$ —parametric w.r.t. the narrowing strategy φ— that can be calculated as the least fixpoint of a generalized version of the usual immediate consequence operator [47] $T^{\varphi}_{\mathcal{R}}$. Since the operator $T^{\varphi}_{\mathcal{R}}$ is continuous over the complete lattice of the Herbrand interpretations [7], the least fixpoint of $T^{\varphi}_{\mathcal{R}}$ (and hence the semantics) is generated by computing at most ω iterations of the operator $T^{\varphi}_{\mathcal{R}}$, that is $lfp(T^{\varphi}_{\mathcal{R}}) = T^{\varphi}_{\mathcal{R}} \uparrow \omega$. Therefore, the fixpoint semantics of a functional logic program can be defined as follows.

Definition 4. *The least fixpoint semantics of a program $\mathcal{R}$ in $\mathbb{R}_{\varphi}$ is defined as*

$$\mathcal{F}_{\varphi}(\mathcal{R}) = lfp(T^{\varphi}_{\mathcal{R}}) = T^{\varphi}_{\mathcal{R}} \uparrow \omega$$

where $\varphi \in \{inn, out\}$.

The fixpoint semantics $\mathcal{F}_{\varphi}(\mathcal{R})$ is more general than the operational semantics $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ in the sense that it models both successful and partial (i.e. intermediate as well as non-terminating) computations, while $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ catches only successful narrowing derivations. Therefore, a fixpoint characterization of the operational semantics can be derived from $\mathcal{F}_{\varphi}(\mathcal{R})$ by removing all those equations representing computations which are still incomplete or not terminating.

Given a set of equations S, let $partial(S)$ be an operator that selects those equations of S that do not model successful computations, i.e., computations that are still incomplete or do not terminate. In other words, we select all equations whose right-hand side is not a constructor term. More formally, $partial(S) = \{l = r \in S \mid r \notin \tau(\mathcal{C} \cup \mathcal{V})\}$.

Theorem 1. *[6] The following relation holds:*

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

4 Narrowing-based Program Transformation

The folding and unfolding transformations, that were first introduced by Burstall and Darlington in [25] for functional programs, are the most basic and powerful techniques for a framework to transform programs. Unfolding is essentially the replacement of a call by its body, with appropriate substitutions. Folding is the inverse transformation, that is, the replacement of some piece of code by an equivalent function call. For functional programs, folding and unfolding steps involve only pattern matching. The fold/unfold transformation approach was first adapted to logic programs by Tamaki and Sato [72] by replacing matching with unification in the transformation rules. A lot of literature has been devoted to proving the correctness of fold/unfold systems w.r.t. the various semantics proposed for functional programs [25], logic programs [72], and constraint logic programs [35]. However, there are several other applications for fold/unfold rules besides providing a general theoretical basis for program transformation. For instance, such transformations have been used to formalize inductive programming frameworks for program synthesis as well as theory revision [24, 4]. To this respect, an example of an unfolding-based theory revision technique for the automated repair of functional logic programs is described in Section 6.2.

Another important application is program analysis. Program analyses can be improved by iterating the unfolding of a program a finite number of times. In fact, an analysis is in general more accurate on the unfolded program than on the original program [11].

Example 2. Consider the following program $\mathcal{R}$ for addition and doubling of natural numbers in Peano's notation.

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

Now, given an equational unification problem $s = t$ in $\mathcal{R}$, consider the unsatisfiability analysis which is based on the idea on *non-joinability* of the root symbols of the normal forms of s and t. Namely, consider the abstract TRS $\mathcal{R}^\alpha$ that is obtained by abstracting the lhs's and rhs's of the rules in $\mathcal{R}$ using the abstraction function $\alpha(t) = f$ for $t = f(t_1, \ldots, t_n)$, whereas $\alpha(x) = c$, with $c \in \mathcal{C}$, for $x \in \mathcal{V}$:

$$\begin{array}{ll} \mathtt{double} & \to \mathtt{add}. \\ \mathtt{add} & \to \mathtt{0}. \\ \mathtt{add} & \to \mathtt{s}. \end{array}$$

Then, the analysis consists in proving that $(s\downarrow)^\alpha$ and $(t\downarrow)^\alpha$ are not joinable in $\mathcal{R}^\alpha$, where $(u\downarrow)$ denotes the normal form of u in $\mathcal{R}$. Unfortunately, this analysis is too naïve (imprecise) to conclude the unsatisfiability of the equation $\mathtt{double(s(x))} = \mathtt{0}$, since the the normal form of $\mathtt{double(s(x))}$ is $\mathtt{add(s(x),s(x))}$, whose root symbol $\mathtt{add}$ can be reduced to $\mathtt{0}$ in $\mathcal{R}^\alpha$. However, by unfolding the first rule of $\mathcal{R}$ w.r.t. the rules for addition, we get the unfolded program $Unf(\mathcal{R})$ (see Definition 6 below):

$$\begin{array}{ll}
\texttt{double(0)} & \to \texttt{0}. \\
\texttt{double(s(x))} & \to \texttt{s(add(x, s(x)))}. \\
\texttt{add(0, x)} & \to \texttt{x}. \\
\texttt{add(s(x), y)} & \to \texttt{s(add(x, y))}.
\end{array}$$

Now, by running the analysis in the unfolded program $Unf(\mathcal{R})$ instead of $\mathcal{R}$, the unsatisfiability of the considered equation $\texttt{double(s(x))} = \texttt{0}$ follows.

In the functional logic setting, a natural way to program transformation is to use a form of narrowing-driven unfolding/folding, i.e., the expansion and the contraction, by means of narrowing, of program subexpressions using the corresponding definitions. A complete characterization of fold/unfold transformations w.r.t. computed answers in functional logic languages with eager/lazy semantics can be found in [12, 14].

The use of narrowing empowers the fold/unfold system by implicitly embedding the instantiation rule (the operation of the Burstall and Darlington framework [25] which introduces an instance of an existing equation) into the fold/unfold operators by means of unification.

4.1 Unfolding Functional Logic Programs

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 in the right-hand side of a rule may be narrowable.

Definition 5 (Unfolding operators). *Let $\mathcal{R}$ be a program, and $\varphi \in \{inn, out\}$ be a narrowing strategy.*

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

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

where y is a fresh variable.

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

$$Unf^{\varphi}(\mathcal{R}, r) = \begin{cases} r & \text{if } \mathsf{U}^{\varphi}_{r'}(r) = \emptyset \text{ for each } r' \in \mathcal{R} \\ \bigcup_{r' \in \mathcal{R}} \mathsf{U}^{\varphi}_{r'}(r) & \text{otherwise} \end{cases}$$

Under a theoretical viewpoint, given a functional logic program, it is possible to define a semantics based on unfolding which is equivalent to its operational and fixpoint ones. This unfolding semantics helps to prove the equivalence between the operational and the fixpoint semantics of the language.

The formalization of such a semantics is as follows.

Definition 6 (Program Unfolding). *Let $\mathcal{R}$ be a program, and $\varphi \in \{inn, out\}$ be a narrowing strategy. The unfolding of a program $\mathcal{R}$ via φ is the program obtained by unfolding via φ the rules of $\mathcal{R}$ w.r.t. $\mathcal{R}$. Formally,*

$$Unf^{\varphi}(\mathcal{R}) = \bigcup_{r \in \mathcal{R}} \{Unf^{\varphi}(\mathcal{R}, r)\}.$$

The repeated application of the program unfolding operator leads to a sequence of equivalent programs which is inductively defined as follows.

Definition 7. *Let $\mathcal{R}$ be a program, and $\varphi \in \{inn, out\}$ be a narrowing strategy. The sequence:*

$\mathcal{R}^0 \quad = \mathcal{R}$
$\mathcal{R}^{i+1} = Unf^{\varphi}(\mathcal{R}^i), i \geq 0$

is called the unfolding sequence starting from $\mathcal{R}$ via φ.

The unfolding semantics of a program is defined as the limit of the unfolding process described in Definition 7. Let us now formally define the *unfolding semantics* $\mathcal{U}^{ca}_{\varphi}(\mathcal{R})$ of a program $\mathcal{R}$. The main point of this definition is in compelling the right-hand sides of the equations in the denotation to be constructor terms. Recall that $\Im_{\mathcal{R}}$ be the set of identical equations $c(x_1, \ldots, x_n) =_{\varphi} c(x_1, \ldots, x_n)$, for each $c/n \in \mathcal{C}$.

Definition 8. *Let $\mathcal{R}$ be a program, and $\varphi \in \{inn, out\}$ be a narrowing strategy. Then,*

$$\mathcal{U}^{ca}_{\varphi}(\mathcal{R}) = \Im_{\mathcal{R}} \cup \bigcup_{i \in \omega} \{(s = d) \mid (s \to d) \in \mathcal{R}^i \quad and\ d \in \tau(\mathcal{C} \cup \mathcal{V})\}$$

where $\mathcal{R}^0, \mathcal{R}^1, \ldots$ is the unfolding sequence starting from $\mathcal{R}$ via φ.

Finally, the following theorem formalizes a useful alternative characterization of the computed answers semantics $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ in terms of unfolding.

Theorem 2. *Let $\mathcal{R} \in I\!R_{\varphi}$. Then, $\mathcal{U}^{ca}_{\varphi}(\mathcal{R}) = \mathcal{O}^{ca}_{\varphi}(\mathcal{R})$.*

4.2 Folding Functional Logic Programs

In the following, we introduce a folding transformation for the *inn* narrowing strategy that can be seen as an extension to functional logic programs of the reversible folding of [67] for logic programs. We have chosen this form of folding since it exhibits the useful, pursued property that the answer substitutions computed by innermost narrowing are preserved through the transformation. Actually, such a result does not hold for the *out* narrowing strategy.

Let us introduce the innermost folding operation. We use the following auxiliary notation. Let $\{r_1, \ldots r_n\}$ be a set of program rules and $\mathcal{R}$ be a program, then by $\{r_1, \ldots r_n\} \ll \mathcal{R}$, we denote the fact that $r_i \ll \mathcal{R}$, for each $i = 1, \ldots, n$.

Definition 9 (Innermost fold). *Let $\mathcal{R}$ be a program. Let $\{r_1, \ldots, r_n\} \ll \mathcal{R}$ (the "folded rules") and $R_{def} = \{r'_1, \ldots, r'_n\} \ll \mathcal{R}$ (the "folding rules") be two disjoint subsets of program rules (modulo renaming), with $r'_i = (\lambda'_i \to \rho'_i \Leftarrow C'_i)$, $i = 1, \ldots, n$. Let r be a rule[7], $u \in O(r)$ be a position of the rule r, and t be a pattern such that, for all $i = 1, \ldots, n$:*

1. $\theta_i = mgu(\{\lambda'_i = t\})$,
2. $r_i = (\lambda \to \rho_i \Leftarrow C'_i, C_i)\theta_i$ and $r[\rho'_i]_u = (\lambda \to \rho_i \Leftarrow C_i)$, and
3. for any rule $r' = (\lambda' \to \rho' \Leftarrow C') \ll \mathcal{R}$ not in R_{def}, λ' does not unify with t.

Then, we define the folding of $\{r_1, \ldots, r_n\}$ in $\mathcal{R}$ using R_{def} as follows:

$$Fold(\mathcal{R}, \{r_1, \ldots, r_n\}, R_{def}) = (\mathcal{R} - \{r_1, \ldots, r_n\}) \cup \{r_{fold}\}$$

where $r_{fold} = r[t]_u$.

Intuitively, the folding operation proceeds in a contrary direction to the narrowing steps. In narrowing steps, for a given unifier of the redex and the left-hand side of the applied rule, a reduction step is performed on the instantiated redex, then the conditions of the unfolding rule are added to the unfolded one, and finally the narrowing substitution is applied. Here, first of all, folded rules are "deinstantiated" (generalized). Next, one gets rid of the conditions of the applied folding rules, and, finally, a reduction step is performed against the reversed heads of the folding rules.

Note that the folding operation has two sources of non-determinism. The first is in the choice of the folded calls; the second is in the choice of a generalization (folding call) of the heads of the instantiated function definitions which are used to substitute the folded calls.

Example 3. Let us consider the following program $\mathcal{R}$:

$$\begin{array}{llll}
\mathtt{f(x)} & \to \mathtt{s(x)} & \Leftarrow \mathtt{h(s(x))} = \mathtt{0} & (r_1) \\
\mathtt{f(s(z))} & \to \mathtt{s(s(0))} & \Leftarrow \mathtt{z} = \mathtt{0} & (r_2) \\
\mathtt{num(y)} & \to \mathtt{y} & \Leftarrow \mathtt{h(y)} = \mathtt{0} & (r_3) \\
\mathtt{num(s(s(z)))} & \to \mathtt{s(s(0))} & \Leftarrow \mathtt{z} = \mathtt{0} & (r_4)
\end{array}$$

Now, we can fold the rules $\{r_1, r_2\}$ of $\mathcal{R}$ w.r.t. $R_{def} = \{r_3, r_4\}$ using $r = (\mathtt{f(x)} \to \Box)$ and $t = \mathtt{num(s(x))}$, obtaining the resulting program $\mathcal{R}'$:

$$\begin{array}{llll}
\mathtt{f(x)} & \to \mathtt{num(s(x))} & & (r_{fold}) \\
\mathtt{num(y)} & \to \mathtt{y} & \Leftarrow \mathtt{h(y)} = \mathtt{0} & (r_3) \\
\mathtt{num(s(s(z)))} & \to \mathtt{s(s(0))} & \Leftarrow \mathtt{z} = \mathtt{0} & (r_4)
\end{array}$$

In [12], it has been shown that the proposed fold transformation preserves the operational semantics $\mathcal{O}^{ca}_{inn}(\mathcal{R})$ of computed answer substitutions of functional logic programs under the usual conditions for the completeness of the *inn* strategy.

[7] Roughly speaking, r is the "common skeleton" of the rules that are folded in the folding step. The occurrence u in r acts as the pointer to the "hole" where the folding call is let fall.

Theorem 3 (Strong correctness). *[12] Let $\mathcal{R} \in I\!R_{inn}$ be a program and $\mathcal{R}' = Fold(\mathcal{R}, \{r_1, \ldots, r_n\}, R_{def})$ be a folding of $\{r_1, \ldots, r_n\}$ in $\mathcal{R}$ using R_{def}. Then, we have that $\mathcal{O}^{ca}_{inn}(\mathcal{R}) = \mathcal{O}^{ca}_{inn}(\mathcal{R}')$.*

An extension of the narrowing-based fold/unfold transformation framework of [12, 13] to rewriting logic theories as implemented in the functional programming language Maude [29] can be found in [3]. It allows one to deal with (nondeterministic) rules, equations, sorts and algebraic laws (like commutativity and associativity). This program transformation framework is also applied to to the problem of securing the transfer of code from a code producer to a code consumer by implement a Code Carrying Theory (CCT) system based on folding/unfolding transformations. CCT is an approach for securing delivery of code from a producer to a consumer where only a certificate (usually in the form of assertions and proofs) is transmitted from the producer to the consumer who can check its validity and then extract executable code from it. In the approach of [3], the certificate consists of a sequence of transformation steps which can be applied to a given consumer specification in order to automatically synthesize safe code in agreement with the original requirements. The key idea behind our CCT methodology is as follows. Assuming the code consumer provides the requirements in the form of a rewrite theory, the code producer can (semi-) automatically obtain an efficient implementation of the specified functions by applying a sequence of transformation rules. Moreover, having proved the correctness of the transformation system, the code producer can transmit as the required certificate just a compact representation of the sequence of transformation rules to the consumer so he does not need to manually construct any other correctness proof. By applying the transformation rules to the initial requirements, the code consumer can inexpensively obtain the executable code that can be eventually compiled to a different target language if needed.

5 Functional Logic Program Specialization

The aim of *partial evaluation* (PE) is to specialize a given program w.r.t. part of its input data (hence also called *program specialization*). PE has been widely applied in the field of functional programming (FP) [50] and logic programming (LP) [55]. Although the objectives are similar, the general methods are often different due to the distinct underlying computation models. This separation has the negative consequence of duplicated work since developments are not shared and many similarities are overlooked.

Narrowing-driven PE (NPE) [15] is the first generic algorithm for the specialization of functional logic programs. The method is formalized within the theoretical framework established in [55, 61] for the partial evaluation of logic programs (also known as *partial deduction*, PD), although a number of concepts have been generalized to deal with nested function calls. The NPE approach has better opportunities for optimization thanks to the functional dimension (e.g. by the inclusion of deterministic simplification steps). Also, since unification is

embedded into narrowing, it is able to automatically propagate syntactic information on the partial input (term structure) and not only constant values. The different instances of the framework which can be obtained by considering different narrowing strategies preserve some logical, strong (computed answers) program semantics under conditions easily ascertained by reusing methods and results developed for narrowing.

Given a program P and a set S of atoms, the aim of PD [55] is to derive a new program P' which computes the same answers for any input goal which is an instance of an atom in S. The program P' is obtained by gathering together the set of *resultants*, which are constructed as follows: for each atom A of S, i) first construct a finite SLD-tree, $T(A)$, for $P \cup \{\Leftarrow A\}$, then ii) consider the leaves of the non-failing branches of $T(A)$, say $G_1, \ldots, G_r$, and the computed substitutions along these branches, say $\theta_1, \ldots, \theta_r$, and finally iii) construct the clauses: $A\theta_1 \Leftarrow G_1, \ldots, A\theta_r \Leftarrow G_r$. The basic correctness of the transformation is ensured whenever P' is *S-closed*, i.e. every atom in P' is an instance of an atom in S. An *independence* condition, which holds if no two atoms in S have a common instance, is needed to guarantee that P' does not produce additional answers. The constructed SLD-trees can be viewed as (i) *symbolic computations* for the atoms in S; the S-closedness of P' illustrates the idea of (ii) *regularity* of a symbolic computation; and finally, (iii) *program extraction* from a set of SLD-trees consists basically in building up the associated set of resultant rules.

We now identify these three categories for narrowing-driven PE [15, 17].

Symbolic Execution It is similar to PD, but we use narrowing in the place of SLD-resolution. For a set S of terms (possibly with nested function calls) and a functional logic program $\{\lambda_i \rightarrow \rho_i \Leftarrow C_i\}_{i=1}^{n}$, a partial (finite) narrowing tree is constructed for each term in S. The inclusion of a deterministic, normalization process between narrowing steps improves the elimination of intermediate data structures and reduces the size of the specialized program since less choices are unfolded [8]. By exploiting the results on *normalizing narrowing* [41], this is achieved in a principled way which does not compromise termination. Control issues are managed by using standard techniques as in [61].

Search for Regularities Our notion of regularity is similar to the PD closedness condition, which we have generalized to recurse over the terms in order to handle nested function calls. Informally, a term t is considered S-closed iff it only contains constructors and variables, or i) there exists a substitution θ such that $t\theta \in S$, and ii) the terms in θ are recursively S-closed. For instance, the term $f(g(0))$ is closed w.r.t. the set of calls $\{f(x), g(x)\}$.

Program Extraction In order to extend the notion of resultant to our setting, we specialize single terms s, and consider derivations for initial goals $s = y$, where y is a fresh variable not occurring in s, that we extend down to the leaves $(C, t = y)$ (where C are the equations brought by the conditions of the applied program rules), and we extract the resultant as $(s\theta \rightarrow t \Leftarrow C)$.

There are two issues of correctness for a PE procedure: termination, i.e., given any input goal, execution should always reach a stage for which there is no

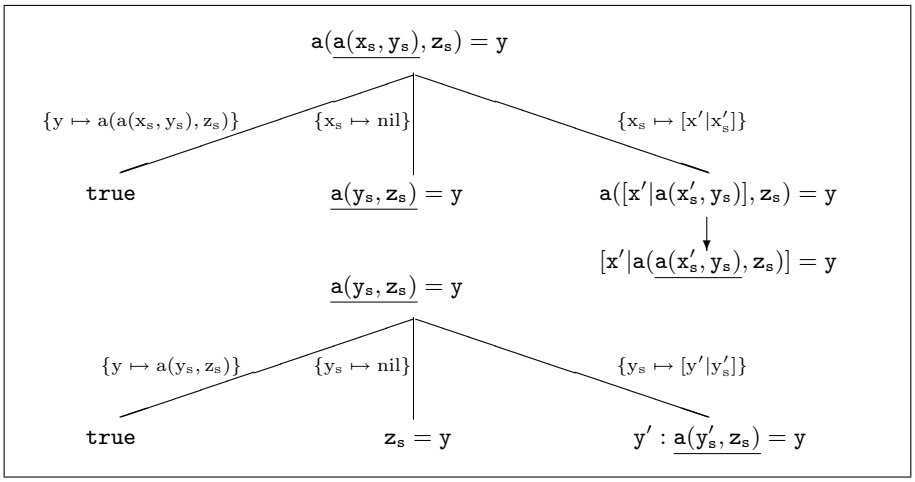

Fig. 1. Narrowing trees for the goals $\mathtt{a}(\mathtt{a}(\mathtt{x_s},\mathtt{y_s}),\mathtt{z_s}) = \mathtt{y}$ and $\mathtt{a}(\mathtt{x_s},\mathtt{y_s}) = \mathtt{y}$.

way to continue; and (partial) correctness, i.e., (if execution terminates, then) the operational semantics of the goal with respect to the residual program and with respect to the original program should coincide.

As for termination, NPE involves two classical termination problems: the so-called *local* termination problem (the termination of unfolding, or how to control and keep the expansion of the narrowing trees which provide partial evaluations for individual calls finite), and the *global* termination (which concerns termination of recursive unfolding, or how to stop recursively constructing narrowing trees while still guaranteeing that the desired amount of specialization is retained and that the closedness condition is reached). Actually, the set of terms S appearing in the goals with which the specialization is performed usually needs to be augmented in order to fulfill the closedness condition. This brings up the problem of how to keep this set finite throughout the PE process by means of some appropriate abstraction operator which guarantees termination. Control issues in narrowing-driven partial evaluation can be controlled by using standard techniques as in [61]. A detailed algorithm for the partial evaluation of functional logic programs can be found in [15], which is able to guarantee the termination of the specialization process.

A partial evaluation is defined as the set of resultants extracted from the derivations of the constructed partial narrowing trees, as illustrated in the following example.

Example 4. Consider again the function `append` of Example 1 with initial goal $\mathtt{append}(\mathtt{append}(\mathtt{x_s},\mathtt{y_s}),\mathtt{z_s}) = \mathtt{y}$. This goal appends three lists by appending the first two, yielding an intermediate list, and then appending the last one to that. We evaluate the goal by using *normalizing* conditional narrowing (that is, each narrowing step is followed by the normalization of the narrowed goal w.r.t. the

given CTRS). Starting with the sequence $q = \mathtt{append}(\mathtt{append}(\mathtt{x_s}, \mathtt{y_s}), \mathtt{z_s})$, we compute the trees depicted in Figure 1 for the sequence of terms

$$q' = (\mathtt{append}(\mathtt{append}(\mathtt{x_s}, \mathtt{y_s}), \mathtt{z_s}), \mathtt{append}(\mathtt{x_s}, \mathtt{y_s})).$$

Note that `append` has been abbreviated to `a` in the picture. Then we get the following residual program $\mathcal{R}'$:

$$\begin{array}{l}
\mathtt{append}(\mathtt{append}(\mathtt{nil}, \mathtt{y_s}), \mathtt{z_s}) \to \mathtt{append}(\mathtt{y_s}, \mathtt{z_s}) \\
\mathtt{append}(\mathtt{append}([\mathtt{x}|\mathtt{x_s}], \mathtt{y_s}), \mathtt{z_s}) \to [\mathtt{x}|\mathtt{append}(\mathtt{append}(\mathtt{x_s}, \mathtt{y_s}), \mathtt{z_s})] \\
\mathtt{append}(\mathtt{nil}, \mathtt{z_s}) \to \mathtt{z_s} \\
\mathtt{append}([\mathtt{y}|\mathtt{y_s}], \mathtt{z_s}) \to [\mathtt{y}|\mathtt{append}(\mathtt{y_s}, \mathtt{z_s})]
\end{array}$$

which is able to append the three lists by passing over its input only once. This result has been obtained thanks to using normalization. Note that no specific strategy has been employed for executing the goal, while the intended specialization has been achieved.

The use of efficient forms of narrowing can significantly improve the accuracy of the specialization method and increase the efficiency of the resulting program, because runtime optimizations are also performed at specialization time.

The behavior of a concrete narrowing-driven partial evaluator greatly depends on the narrowing strategy being used, since different strategies have quite different semantic properties. It is accepted that the use of an eager narrowing strategy is less convenient than a lazy one regarding the elimination of intermediate data structures, although the use of normalization may alleviate the problem, as said before. Generally speaking, if the operational semantics $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ is to be preserved by program transformations, then the only reasonable class (for eager as well as for lazy PE) is that of left-linear, constructor-based (CB) programs, which are known to produce only constructor answers. These programs are generalized to the more general class of left-linear *rnf-based* programs, where all arguments of the left-hand sides of the rules are rigid normal forms, i.e. unnarrowable. Unfortunately, the construction of resultants may produce rewrite rules whose left-hand side contain nested function symbols, if the terms to be partially evaluated contain nested function symbols. If this kind of programs are allowed by the narrowing strategy being considered (e.g. unrestricted narrowing), there is no problem at all. However, when dealing with narrowing strategies which require constructor-based programs, a post-processing renaming transformation is mandatory in order to have an executable residual program [8]. Complex terms are 'folded' recursively, by replacing them by calls to new functions which satisfy the CB constraint. Furthermore, it can automatically guarantee that no additional answer is computed in the specialized program, which is otherwise ensured by an *independence condition* on the set of partially evaluated terms (as explained in [15]) guaranteeing that no *overlaps* exist between the specialized function calls.

Example 5. In Example 4, the resulting set of terms

$$\{\mathtt{append}(\mathtt{append}(\mathtt{x_s}, \mathtt{y_s}), \mathtt{z_s}), \mathtt{append}(\mathtt{x_s}, \mathtt{y_s})\}$$

in q' is not independent. This example illustrates the need for an extra renaming phase able to produce an independent set of terms such as $\{\texttt{app\_3}(\texttt{x}_\texttt{s}, \texttt{y}_\texttt{s}, \texttt{z}_\texttt{s}), \texttt{app\_2}(\texttt{x}_\texttt{s}, \texttt{y}_\texttt{s})\}$ and associated specialized program

$$\begin{array}{l}\texttt{app\_3}(\texttt{nil}, \texttt{y}_\texttt{s}, \texttt{z}_\texttt{s}) \rightarrow \texttt{app\_2}(\texttt{y}_\texttt{s}, \texttt{z}_\texttt{s}) \\ \texttt{app\_3}([\texttt{x}|\texttt{x}_\texttt{s}], \texttt{y}_\texttt{s}, \texttt{z}_\texttt{s}) \rightarrow [\texttt{x}|\texttt{app\_3}(\texttt{x}_\texttt{s}, \texttt{y}_\texttt{s}, \texttt{z}_\texttt{s})] \\ \texttt{app\_2}(\texttt{nil}, \texttt{z}_\texttt{s}) \rightarrow \texttt{z}_\texttt{s} \\ \texttt{app\_2}([\texttt{y}|\texttt{y}_\texttt{s}], \texttt{z}_\texttt{s}) \rightarrow [\texttt{y}|\texttt{app\_2}(\texttt{y}_\texttt{s}, \texttt{z}_\texttt{s})]\end{array}$$

which has the same computed answers as the original program `append` for the query $\texttt{app\_3}(\texttt{x}_\texttt{s}, \texttt{y}_\texttt{s}, \texttt{z}_\texttt{s})$ (modulo the renaming transformation).

The use of lazy narrowing during partial evaluation gives a better overall behavior regarding both the elimination of intermediate data structures and the propagation of information. Unfortunately, this approach introduces new drawbacks into the partial evaluation process. Firstly, the class of programs is not preserved by the transformation; for instance, orthogonality may be destroyed. On the other hand, the quality of the partially evaluated program may be degraded by introducing e.g. infinite computations which could not be proven in the original program [18]. The use of needed narrowing during partial evaluation overcomes both problems, since the structure of programs is preserved and no redundant or undesirable derivations are encoded in the residual program. Nevertheless, a new difficulty arises when the operation principle of *residuation* is integrated within the NPE framework. Namely, the difficulty lies in preserving the *floundering* behaviour of the original program, ensuring that there is a precise correspondence between the computations that suspend in the original and the specialized programs [2].

6 Declarative debugging

Debugging programs with the combination of user-defined functions and logic variables is a difficult but important task which has deserved some interest in recent years, and different debugging techniques have been proposed. 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.

Abstract diagnosis [30] is a declarative debugging framework that extends the methodology in [38, 70] (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 [6, 7], we generalized the declarative diagnosis methodology of [30] to the debugging of wrong as well as missing answers of functional logic programs.

In our setting, correctness as well as completeness of a program $\mathcal{R}$ are established by comparing the operational semantics $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ to an intended semantics $\mathcal{I}_{ca}$ modeling the successful narrowing derivations that a programmer has in mind. More formally,

Definition 10. *Let $\mathcal{I}_{ca}$ be the intended success set semantics for program $\mathcal{R}$.*

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 [36].

Definition 11. *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 detection of buggy rules, however, we need to consider a "well-furnished" intended fixpoint semantics $\mathcal{I}_{\mathcal{F}}$ (such that $\mathcal{I}_{ca} \subseteq \mathcal{I}_{\mathcal{F}}$), which models successful as well as "in progress" (i.e., partial) computations, and enjoys the semantic properties of the denotation formalized in Definition 4, that is, $\mathcal{I}_{\mathcal{F}}$ should correspond to the fixpoint semantics of the correct program and $\mathcal{I}_{ca} = \mathcal{I}_{\mathcal{F}} - partial(\mathcal{I}_{\mathcal{F}})$.

An equation e is*uncovered* if it cannot be derived by any program rule using the intended fixpoint semantics, in symbols $e \in \mathcal{I}_{\mathcal{F}}$ and $e \notin T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$. Having such a semantics, the diagnosis of buggy rules as well as the detection of uncovered equations can be performed by exploiting the following definitions.

Definition 12. *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 covered 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 13. *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 covered 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 covered by $T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$, since such equations represent missing computed answers.

Partial correctness of a program is established by the following proposition.

Proposition 1. *[7] 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 $\mathcal{I}_{ca}$.*

Assuming that $\mathcal{I}_{\mathcal{F}}$ is finite, Proposition 1 shows a simple methodology to prove partial correctness. In the case when $\mathcal{I}_{\mathcal{F}}$ is not finite, this methodology can be still applied by considering finite approximations of the program semantics, as explained in Section 6.1 below. Completeness is harder, since it not possible to detect all possible uncovered equations by comparing the specification of the intended fixpoint semantics $\mathcal{I}_{\mathcal{F}}$ to $T_{\mathcal{R}}^{\varphi}(\mathcal{I}_{\mathcal{F}})$. In other words, the absence of uncovered equations does not allow us to derive that the program under examination is complete.

It is worth noting that checking the conditions of Definitions 12 and 13 requires just one application of $T_{\mathcal{R}}^{\varphi}$ to $\mathcal{I}_F$, while the standard detection based on symptoms [70] would require either an external oracle or the construction of the semantics, and therefore a fixpoint computation.

6.1 Abstract Diagnosis

In general, the diagnosis methodology we presented in Section 6 cannot be used to directly derive practical debuggers, since the correctness as well as completeness tests of Definitions 12–13 cannot be implemented in an effective way when the intended semantics $\mathcal{I}_{\mathcal{F}}$ is infinite, which is a very common case.

Following an idea inspired by [30], we defined an effective diagnosis methodology in [6, 7], which is based on abstract interpretation [31].

Abstract interpretation formalizes the idea of "approximate computation" in which computation is performed with descriptions of data rather than with the data themselves. In particular, the semantics operators are replaced by abstract operators that are shown to 'safely' approximate the standard ones. In this context, our abstract diagnosis framework allows one to work on finite representations of the intended semantics $\mathcal{I}_F$ giving support to the implementation of finite diagnosis procedures.

More specifically, the basic idea is to consider two *finite* sets: $\mathcal{I}^+$ which over-approximates the intended fixpoint semantics $\mathcal{I}_{\mathcal{F}}$ and $\mathcal{I}^-$ which under-approximates $\mathcal{I}_{\mathcal{F}}$. In our methodology, an executable specification R_{spec} is given in order to effectively compute over- and under-approximations of the intended fixpoint semantics. Basically, we take the set which results from a finite number of iterations of the concrete immediate consequence operator $T_{\mathcal{R}_{Spec}}^{\varphi}$ as under-approximation $\mathcal{I}^-$, while $\mathcal{I}^+$ corresponds to the abstract fixpoint semantics of the abstract specification $R_{spec}^{\sharp}$, which is obtained from R_{spec} by replacing recursive function calls appearing in the specification's rules with occurrences of the special symbol $\sharp$. Such an abstraction allows us to avoid non-termination of the fixpoint computation, and provides a simple methodology for computing $\mathcal{I}^+$ which is satisfactory in practice.

We then use these sets $\mathcal{I}^+$ and $\mathcal{I}^-$ as shown in Theorems 4–5 in order to implement the abstract effective versions of the correctness/completeness tests

of Definitions 12–13. Basically, the immediate consequence operator, $T^{\varphi}_{\mathcal{R}}$, (w.r.t. the program $\mathcal{R}$) is applied to $\mathcal{I}^-$ to check incorrectness w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$ and the abstract version of the immediate consequence operator, $T^{\sharp\varphi}_{\mathcal{R}}$ is applied to $\mathcal{I}^+$ to check incompleteness w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$.

Theorem 4. *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 5. *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 $\mathcal{R}$.*

The previous theorems provide a compact description of the results proved in [6, 7] and are the basis of the correctness of our abstract diagnosis framework.

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.* $(\mathcal{I}^+, \mathcal{I}^-)$. 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. $(\mathcal{I}^+, \mathcal{I}^-)$ does not generally imply the total correctness of $\mathcal{R}$ w.r.t. $\mathcal{I}$. The method is sound[8] in the sense that each error which is found by using $\mathcal{I}^+, \mathcal{I}^-$ is really a bug w.r.t. $\mathcal{I}$.

Example 6. 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)).
```

The abstract specification $\mathcal{R}^{\sharp}_{Spec}$ is

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

Let $\varphi = inn$; then. After 2 iterations of the $T^{inn}_{\mathcal{R}_{Spec}}$ operator, we get the following under-approximation.

[8] This is in contrast with the abstract diagnosis methodologies of [5, 30], 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.

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

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

$$\begin{aligned}\mathcal{I}^+ = \{&\mathtt{0}=\mathtt{0}, \mathtt{s(x)}=\mathtt{s(x)},\ \mathtt{add(x,y)}=\mathtt{add(x,y)},\\ &\mathtt{fib(x)}=\mathtt{fib(x)}, \mathtt{add(0,x)}=\mathtt{x}, \mathtt{add(s(x),y)}=\mathtt{s(\sharp)},\\ &\mathtt{fib(0)}=\mathtt{s(0)}, \mathtt{fib(s(0))}=\mathtt{s(0)}, \mathtt{fib(s(s(x)))}=\mathtt{add(\sharp,\sharp)},\\ &\mathtt{fib(s(s(x)))}=\sharp, \mathtt{fib(s(s(x)))}=\mathtt{s(\sharp)}\}\end{aligned}$$

Now, consider the equation $\mathtt{fib(x)}=\mathtt{fib(x)}$ of $\mathcal{I}^-$. By applying $T^{inn}_{\{fib(0)\to 0\}}$ to this equation, we get the equation $e = \mathtt{fib(0)}=\mathtt{0}$, which is not covered by $\mathcal{I}^+$, i.e., it is not subsumed by any abstract equation of $\mathcal{I}^+$. This proves that r is incorrect on e.

6.2 Automated 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 [64]. A challenging subfield of ILP is known as *inductive theory revision,* which is close to program debugging under the *competent programmer* assumption of [70]. In other words, the initial program $\mathcal{R}$ is assumed to be written with the intention of being correct and, if it is not, then a close variant $\mathcal{R}^c$ of it is. Automatic program correction attempts to find such a variant.

In this context, the correction problem can be stated as follows. Let $\mathcal{R}$ be a wrong program such that $\mathcal{R}' \subseteq \mathcal{R}$ is a set of wrong rules w.r.t. an intended semantics $\mathcal{I}_\mathcal{F}$. Let E^p and E^n be two disjoint sets of ground equations witnessing the correct as well as the wrong computational behaviour of $\mathcal{R}$. Equations in E^p (respectively, E^n) are called *positive examples* (respectively, *negative examples*).

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

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

where $\mathcal{R}$ *entails* E using the narrowing strategy $\varphi \in \{inn, out\}$ (in symbols, $\mathcal{R} \vdash_\varphi E$) iff each $e \in E$ is successfully derived in $\mathcal{R}$ using the narrowing strategy

φ (that is, $e \leadsto^*_\varphi \top$ in $\mathcal{R}$), and $\mathcal{R}$ *disproves* E using the narrowing strategy (in symbols, $\mathcal{R} \not\vdash_\varphi E$) iff no $e \in E$ can be successfully derived in $\mathcal{R}$ using the narrowing strategy φ.

Program $\mathcal{R}^c$ is called *corrected* program (w.r.t. E^p and E^n). Roughly speaking, a corrected program $\mathcal{R}^c$ is a program that entails all the positive examples and disproves all the negative examples.

In [4], we developed an automated procedure for program correction which mainly follows the top-down, inductive learning approach known as example-guided unfolding [24], which uses unfolding as specialization operator to discriminate positive from negative examples. The basic idea of the method is to first specialize the program $\mathcal{R}$ by unfolding function calls in the right-hand sides of the rules yielding a close variant $\mathcal{R}'$ of $\mathcal{R}$. Then, we obtain $\mathcal{R}^c$ by removing those rules of $\mathcal{R}'$ which are responsible for the derivation of the negative examples.

For example, consider the following program $\mathcal{R}$

$$\begin{aligned} \texttt{even(0)} &\rightarrow \texttt{true} && (r_1)\\ \texttt{even(s(x))} &\rightarrow \texttt{even(x)} && (r_2) \end{aligned}$$

which is wrong w.r.t. the usual intended semantics of the `even` function. Moreover, let E^p be $\{\texttt{even(0)} = \texttt{true},\ \texttt{even(s}^2\texttt{(0))} = \texttt{true},\ \texttt{even(s}^4\texttt{(0))} = \texttt{true}\}$ and E^n be $\{\texttt{even(s(0))} = \texttt{true},\ \texttt{even(s}^3\texttt{(0))} = \texttt{true},\ \texttt{even(s}^5\texttt{(0))} = \texttt{true}\}$.

The wrong program $\mathcal{R}$ can be first transformed into an equivalent program $\mathcal{R}'$ by unfolding rule (r_2) as follows:

$$\begin{aligned} \texttt{even(0)} &\rightarrow \texttt{true} && (r_1)\\ \texttt{even(s(0)} &\rightarrow \texttt{true} && (r_2{}')\\ \texttt{even(s(s(x)))} &\rightarrow \texttt{even(x)} && (r_2{}'') \end{aligned}$$

Then, note that we can obtain the desired corrected program by simply removing rule r'_2 from $\mathcal{R}'$.

Soundness of this approach has been proven in [4].

The unfolding-based correction procedure presented above is known to produce a correction when the initial program is *overly general* (with some extra outfit which is needed to specialize recursive definitions [24]); that is, it allows us to prove all positive examples and some incorrect ones. Unfortunately, most of the programs to be debugged are not overly general, and hence our correction methodology cannot be directly applied. Therefore, we coupled the example-guided unfolding approach with a generalization technique in order to correct programs that do not fulfill 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.

Roughly speaking, we extend the original erroneous 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.

The generalization method exploits the bottom-up technique for the inductive learning of functional logic programs developed by Ferri, Hernández and Ramírez [39] which automatically infers new program rules from sets of ground examples. The induction process is based on *inverse narrowing* — a variant of Muggleton's inverse resolution operator [64]— 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 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 Related Work and Concluding Remarks

Finding program bugs is a long-standing problem in software construction. Unfortunately, the debugging support is rather poor for functional languages (see [60] 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. Due to the complexity of the operational semantics of (functional) logic programs, the information obtained by tracing the execution is difficult to understand. Several authors follow the idea of algorithmic declarative debugging in the style proposed by Shapiro [70]: an oracle (typically the user) is supposed to endow 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 debugger for the functional logic language Escher based on this methodology is proposed in [54]. Unfortunately, when debugging 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 [65], a procedure for the declarative debugging of wrong answers in higher-order functional logic programs is proposed in [28]. 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 and adapted to Curry. The methodology in [28] 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 [27].

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 [70], 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], a framework for the diagnosis and the correction of Moded flat GHC programs 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.

We are not aware of any formal antecedent of the narrowing-driven approach in the PE literature. A closer, automatic approach is that of positive supercompilation [71], whose basic operation is *driving* [74], a unification-based transformation mechanism which is somewhat similar to (lazy) narrowing. Another related work is the framework of *conjunctive partial deduction* (CPD), which aims at achieving unfold/fold-like transformations within fully automated PD [52]. Similarly to conjunctive partial deduction [52] and supercompilation [74], NPE combines some good features of deforestation [75], partial evaluation [50], and PD [55, 61].

All the narrowing–based techniques for program transformation and debugging that are overviewed in this paper have been implemented in a collection of tools that are publicly available at `www.dsic.upv.es/users/elp/soft.html`.

Acknowledgements

This paper is a modest attempt to summarize twenty years of research on narrowing-based program manipulation in Italy by reviewing the main lines of research and contributions of the authors in the following fields: program transformation, partial evaluation and program debugging of functional logic programs. Many thanks are due to Francisco Correa, Ginés Moreno, and Germán Vidal, large part of the material here reported was developed with their collaboration.

References

1. Y. Ajiro and K. Ueda. Kima — an Automated Error Correction System for Concurrent Logic Programs. In *Automated Software Engineering*, volume 19, pages 67–94, 2002.
2. E. Albert, M. Alpuente, M. Hanus, and G. Vidal. A Partial Evaluation Framework for Curry Programs. In *Proc. 6th Int'l Conf. on Logic for Programming and Automated Reasoning, LPAR'99*, pages 376–395. Springer LNAI 1705, 1999.
3. M. Alpuente, M. Baggi amd D. Ballis, , and M. Falaschi. A Fold/Unfold Transformation Framework for Rewrite Theories and its Application to CCT. In *Proc. 2010 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 43–52. ACM, 2010.

4. M. Alpuente, D. Ballis, F. J. Correa, and M. Falaschi. Automated Correction of Functional Logic Programs. In P. Degano, editor, *Proc. European Symp. on Programming, ESOP 2003*, pages 54–68. Springer LNCS 2618, 2003.
5. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In M. Leuschel and F. Bueno, editors, *Proc. LOPSTR 2002*, pages 1–16. Springer LNCS 2664, 2003.
6. M. Alpuente, F. Correa, and M. Falaschi. Declarative Debugging of Functional Logic Programs. In B. Gramlich and S. Lucas, editors, *Proc. Int'l Workshop on Reduction Strategies in Rewriting and Programming, WRS 2001*, Elsevier ENTCS 57, 2001.
7. M. Alpuente, F. Correa, and M. Falaschi. A Debugging Scheme for Functional Logic Programs. In M. Hanus, editor, *Proc. 10th Int'l Workshop on Functional and (Constraint) Logic Programming, WFLP 2001*, Elsevier ENTCS 64, 2002.
8. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, Sigplan Notices 32(12), pages 151–162, New York, 1997. ACM Press.
9. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Uniform Lazy Narrowing. *Journal of Logic and Computation*, 13(2):287–312, 2003.
10. M. Alpuente, M. Falaschi, and G. Levi. Incremental Constraint Satisfaction for Equational Logic Programming. *Theoretical Computer Science*, 142(1):27–57, 1995.
11. M. Alpuente, M. Falaschi, and F. Manzo. Analyses of Unsatisfiability for Equational Logic Programming. *Journal of Logic Programming*, 22(3):221–252, 1995.
12. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe folding/unfolding with conditional narrowing. In *Proc. Int'l Conf. on Algebraic and Logic Programming, ALP'97*, pages 1–15. Springer LNCS 1298, 1997.
13. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. An Automatic Composition Algorithm for Functional Logic Programs. In V. Hlaváč, K. G. Jeffery, and J. Wiedermann, editors, *Sofsem 2000—Theory and Practice of Informatics*, pages 289–297. Springer LNCS 1963, 2000.
14. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Rules + Strategies for Transforming Lazy Functional Logic Programs. *Theoretical Computer Science*, 311(1–3):479–525, 2004.
15. M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven Partial Evaluation of Functional Logic Programs. In H. Riis Nielson, editor, *Proc. 6th European Symp. on Programming, ESOP'96*, pages 45–61. Springer LNCS 1058, 1996.
16. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.
17. M. Alpuente, M. Falaschi, and G. Vidal. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 30(3es):9es, 1998.
18. M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of functional logic programs based on needed narrowing. *TPLP*, 5(3):273–303, 2005.
19. S. Antoy. Evaluation strategies for functional logic programming. *J. Symb. Comput.*, 40(1):875–903, 2005.
20. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
21. T. Arts and J. Giesl. Termination of Term Rewriting using Dependency Pairs. *TCS*, 236(1-2):133–178, 2000.
22. M. Bellia and G. Levi. The relation between logic and functional languages. *Journal of Logic Programming*, 3:217–236, 1986.

23. D. Bert and R. Echahed. On the Operational Semantics of the Algebraic and Logic Programming Language LPG. In *Recent Trends in Data Type Specifications*, pages 132–152. Springer LNCS 906, 1995.
24. H. Bostrom and P. Idestam-Alquist. Induction of Logic Programs by Example–guided Unfolding. *Journal of Logic Programming*, 40:159–183, 1999.
25. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
26. A. Santana de Oliveira C. Kirchner, H. Kirchner. Analysis of Rewrite-Based Access Control Policies. In *Proc. 3rd Int'l Workshop on Security and Rewriting Techniques, SecreT 2008*. Elsevier ENTCS, 2008.
27. R. Caballero, M. Rodríguez-Artalejo, and R. del Vado Vírseda. Declarative Diagnosis of Missing Answers in Constraint Functional-Logic Programming. In *Proc. of FLOPS 2008*, volume 4989 of *LNCS*, pages 305–321. Springer, 2008.
28. R. Caballero-Roldán, F.J. López-Fraguas, and M. Rodríquez Artalejo. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In *Fifth Int'l Symp. on Functional and Logic Programming*, pages 170–184. Springer LNCS 2024, 2001.
29. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*. Springer-Verlag, New York, 2007.
30. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.
31. P. Cousot and 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*, pages 238–252, 1977.
32. R. Echahed. On completeness of narrowing strategies. In *Proc. CAAP'88*, pages 89–101. Springer LNCS 299, 1988.
33. S. Escobar, C. Meadows, and J. Meseguer. A Rewriting-Based Inference System for the NRL Protocol Analyzer and its Meta-Logical Properties. *TCS*, 367(1-2):162–202, 2006.
34. S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *18th Int'l Conference on Rewriting Techniques and Applications, RTA'07*, LNCS 4533: 153–168.
35. S. Etalle and M. Gabbrielli. Modular Transformations of CLP Programs. In L. Sterling, editor, *Proc. 12th Int'l Conf. on Logic Programming*. The MIT Press, Cambridge, MA, 1995.
36. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A new Declarative Semantics for Logic Languages. In R. Kowalski and K. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 993–1005. The MIT Press, Cambridge, MA, 1988.
37. M. Fay. First Order Unification in an Equational Theory. In *Proc of 4th Int'l Conf. on Automated Deduction, CADE'79*, pages 161–167, 1979.
38. G. Ferrand. Error Diagnosis in Logic Programming, and Adaptation of E.Y.Shapiro's Method. *Journal of Logic Programming*, 4(3):177–198, 1987.
39. C. Ferri, J. Hernández, and M.J. Ramírez. Incremental Learning of Functional Logic Programs. In *5th Int'l Symp. on Functional and Logic Programming, FLOPS'01*, pages 233–247. Springer LNCS 2024, 2001.
40. L. Fribourg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. Second IEEE Int'l Symp. on Logic Programming*, pages 172–185. IEEE, New York, 1985.

41. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
42. M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24th ACM Symp. on Principles of Programming Languages (Paris)*, pages 80–93. ACM, New York, 1997.
43. M. Hanus. Multi-paradigm Declarative Languages (invited tutorial). In *Proc. of International Conference on Logic Programming (ICLP 2007)*, volume 4670 of *Lecture Notes in Computer Science*, pages 45–75. Springer, 2007.
44. M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.
45. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
46. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (ver. 0.8.2). Accesible en `http://www.informatik.uni-kiel.de/~curry`, 2006.
47. S. Hölldobler. *Foundations of Equational Logic Programming*. Springer LNAI 353, 1989.
48. J.M. Hullot. Canonical Forms and Unification. In *Proc of 5th Int'l Conf. on Automated Deduction*, pages 318–334. Springer LNCS 87, 1980.
49. H. Hussman. Unification in Conditional-Equational Theories. In *Proc. European Conf. on Computer Algebra EUROCAL'85*, pages 543–553. Springer LNCS 204, 1985.
50. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
51. J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.
52. M. Leuschel, D. De Schreye, and A. de Waal. A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration. In M. Maher, editor, *Proc. the Joint Int'l Conf. and Symp. on Logic Programming, JICSLP'96*, pages 319–332. The MIT Press, Cambridge, MA, 1996.
53. G. Levi, C. Palamidessi, P.G. Bosco, E. Giovannetti, and 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*, pages 318–327. IEEE, New York, 1987.
54. J. W. Lloyd. Debugging for a declarative programming language. *Machine Intelligence 15*, 1998.
55. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
56. F. J. López-Fraguas and J. Sánchez Hernández. Toy: A multiparadigm declarative system. In *Proc. 10th Int'l Conf. on Rewriting Techniques and Applications*, pages 244–247. Springer-Verlag, 1999.
57. F. J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rwwrite notion for call-time choice semantics. In *Proc. 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2007)*, pages 197–208. ACM, 2007.
58. F. J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A flexible framework for programming with non-deterministicfunctions. In *Proc. 2009 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 91–100. ACM, 2009.

59. F. J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Narrowing for first order functional logic programs with call-time choice semantics. In *Proc. Applications of Declarative Programming and Knowledge Management, 17th International Conference (INAP 2007)*, volume 5437 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 2009.
60. S. Marlow, J. Iborra, B. Pope, and A. Gill. A Lightweight Interactive Debugger for Haskell. In Gabriele Keller, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 13–24. ACM, 2007.
61. B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In L. Sterling, editor, *Proc. ICLP'95*, pages 597–611. MIT Press, 1995.
62. J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.
63. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.
64. S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(3):295–318, 1991.
65. L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), April 1997.
66. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.
67. A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
68. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. Second IEEE Int'l Symp. on Logic Programming*, pages 138–151. IEEE, New York, 1985.
69. A. Riesco and J. Rodríguez-Hortalá. Programming with Singular and Plural Nondeterministic Functions. In *Proc. 2010 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 83–92. ACM, 2010.
70. E. Y. Shaphiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Massachusetts, 1982. ACM Distinguished Dissertation.
71. M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
72. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In S. Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming, Uppsala, Sweden*, pages 127–139, 1984.
73. TeReSe, editor. *Term Rewriting Systems*. Cambridge University Press, Cambridge, UK, 2003.
74. V.F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.
75. P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.