Correction of Functional Logic Programs[⋆]

Maria Alpuente[1], Demis Ballis[2], Francisco J. Correa[3], and Moreno Falaschi[2]

[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,falaschi}@dimi.uniud.it.

[3] DIS, U. EAFIT, Cra. 49 N. 7 Sur 50, 3300 Medellín, Colombia. fcorrea@eafit.edu.co.

Abstract. We propose a new methodology for synthesizing correct functional logic programs. We aim to create an integrated development environment in which it is possible to debug a program and correct it automatically. We start from a declarative diagnoser that we have developed previously which allows us to identify wrong program rules w.r.t. an intended specification. Then a bug-correction, program synthesis methodology tries to correct the erroneous components of the wrong code. We propose a hybrid, top-down (unfolding–based) as well as bottom-up (induction–based), approach for the automatic correction of functional logic programs which is driven by a set of evidence examples which are automatically produced as an outcome by the diagnoser. The resulting program is proven to be correct and complete w.r.t. the considered example sets. Finally, we also provide a prototypical implementation which we use for an experimental evaluation of our system.

1 Introduction

The main motivation for this work is to provide a methodology for developing advanced debugging and correction tools for functional logic languages. Functional logic programming is now a mature paradigm and as such there exist modern environments which assist in the design, development and debugging of integrated programs. However, there is no theoretical foundation for integrating debugging and synthesis into a single unified framework. We believe that such an integration can be quite productive and hence develop useful techniques and new results for the process of automatically synthesizing correct programs.

In a previous work [6], a generic diagnosis method w.r.t. computed answers which generalizes the ideas of [11] to the diagnosis of functional logic programs has been proposed. The method works for eager (*call–by–value*) as well as for lazy (*call–by–name*) integrated languages. Given the intended specification $\mathcal{I}$ of a program $\mathcal{R}$, we can check the correctness of $\mathcal{R}$ w.r.t. $\mathcal{I}$ by a single step

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

of a (continuous) immediate consequence operator which we associate to our programs. This specification $\mathcal{I}$ may be partial or complete, and can be expressed in several ways: for instance, by (another) functional logic program [6,2], by an assertion language [10] or by equation sets (in the case when it is finite). Our methodology is based on abstract interpretation: we construct *over* and *under* specifications $\mathcal{I}^+$ and $\mathcal{I}^-$ to correctly over- (resp. under-) approximate the intended semantics $\mathcal{I}$. We then use these two sets respectively for the functions in the premises and the consequences of the immediate consequence operator, and by a simple static test we can determine whether some of the clauses are wrong. The debugging system Buggy[3] is an experimental implementation of the method which allows the user to specify the (concrete) semantics by means of a functional logic program. In [2], we also presented a preliminary correction algorithm based on the deductive synthesis methodology known as *example-guided unfolding* [8]. This methodology uses unfolding in order to discriminate positive from negative examples (resp. uncovered and incorrect equations) which are essentially obtained as an outcome by the diagnoser.

However, this pure deductive learner cannot be applied when the original wrong program is overspecialized (that is, it does not cover all the (positive) examples chosen to describe the pursued behavior). In this paper, we develop a new program corrector based on, and integrated with, the declarative debugger of [6,2], which integrates top–down as well as bottom–up synthesis techniques. The resulting method is conceptually cleaner than more elaborated, purely deductive or inductive learning procedures, and combines the advantages of both styles. Furthermore, our method is parametric w.r.t. the chosen bottom-up learner. As an instance of such parameter, we consider for the bottom-up part of the algorithm the functional logic inductive framework of [17,20]. Informally, our correction procedure works as follows. Starting from an overly general program (that is, a program which covers all positive examples as well as some negative ones), the top–down algorithm unfolds the program until a set of rules which only occur in the refutation of the negative examples is identified, and then they are removed from the program. When the original wrong program does not initially cover all positive examples, we first invoke the bottom–up procedure, which "generalizes" the program as to fulfil the applicability conditions. After introducing the new method we prove its correctness and completeness w.r.t. the considered example sets. Finally we present a prototypical implementation of our system and the relative benchmarks. The following example illustrates our method.

Example 1. Let us consider the program:

$$\mathcal{R} = \{od(\mathbf{0}) \rightarrow true, od(\mathbf{s}(\mathbf{X})) \rightarrow od(X), z(0) \rightarrow \mathbf{1}, z(s(X)) \rightarrow z(X)\,\}$$

which is wrong w.r.t. the following specification of the intended semantics (mistakes in $\mathcal{R}$ are marked in bold):

$$\begin{aligned}\mathcal{I} = \{\; & ev(0) \rightarrow true, ev(s(s(X))) \rightarrow ev(X),\\ & od(s(X)) \rightarrow true \Leftarrow ev(X) = true, z(X) \rightarrow 0\;\}.\end{aligned}$$

By running the diagnosis system Buggy, we are able to isolate the wrong rules of $\mathcal{R}$ w.r.t. the given specification. By exploiting the debugger outcome as described later, the following positive and negative example sets are automatically produced (the user is allowed to fix the cardinality of the example sets by tuning some system parameters):

$$\begin{aligned} E^+ &= \{od(s^3(0)) = true, od(s(0)) = true, z(s^2(0)) = 0, z(s(0)) = 0, z(0) = 0\,\} \\ E^- &= \{od(s^2(0)) = true, od(0) = true, z(0) = 1, z(s(0)) = 1, z(s^2(0)) = 1\,\}. \end{aligned}$$

We observe that unfolding the rule $r \equiv od(s(X)) \to od(X)$ w.r.t. $\mathcal{R}$ results in replacing r by two new rules $r_1 \equiv od(s(0)) \to true$ and $r_2 \equiv od(s^2(X)) \to od(X)$. Now, by getting rid of rule $od(0) \to true$, we obtain a new recursive definition for function od covering the positive examples while no negative example can be proven, which corrects the bug on function od.

However, note that this approach cannot be used for correcting function z: unfolding the rules defining z does not contribute to demonstrate the positive examples since the original program is overspecialized and unfolding can only specialize it further. Nevertheless, by generalizing function z as in the bottom-up inductive framework of [20], we get the new rule $z(X) \to 0$. Now, by eliminating rule $z(0) \to 1$, which does not contribute to any positive example, we obtain the final outcome

$$\mathcal{R}^c = \{od(s(0)) \to true, od(s(s(X))) \to od(X), z(X) \to 0, z(s(X)) \to z(X)\,\}$$

which is correct w.r.t. the computed example sets.

The rest of the paper is organized as follows. Section 2 summarizes some preliminary definitions and notations. Section 3 recalls the framework for the declarative debugging of functional logic programs defined in [2]. In Section 4, we present the basic, top-down automatic correction procedure. Section 5 integrates this algorithm with a bottom-up inductive learner which allows us to apply the correction methodology when the original program is overly specialized. In Section 6, we present an experimental evaluation of the method on a set of benchmarks. Section 7 discusses some related work and concludes. Proofs of all technical results can be found in [1].

2 Preliminaries

Let us briefly recall some known results about rewrite systems [7,22] and functional logic programming (see [19,21] for extensive surveys). For simplicity, definitions are given in the one-sorted case. The extension to many–sorted signatures is straightforward, see [27]. Throughout this paper, V will denote a countably infinite set of variables and Σ denotes a set of function symbols, or *signature*, each of which has a fixed associated arity. $\tau(\Sigma \cup V)$ and $\tau(\Sigma)$ denote the non-ground word (or term) algebra and the word algebra built on $\Sigma \cup V$ and Σ, respectively. $\tau(\Sigma)$ is usually called the *Herbrand universe* ($\mathcal{H}_\Sigma$) over Σ and it will be denoted by $\mathcal{H}$. $\mathcal{B}$ denotes the *Herbrand base*, namely the set of all ground equations which can be built with the elements of $\mathcal{H}$. A *equation* $s = t$ is a pair of terms $s, t \in \tau(\Sigma \cup V)$. Terms are viewed as labelled trees in the usual way. Term *positions* are represented by sequences of natural numbers, where Λ denotes the empty sequence. $O(t)$ denotes the set of positions of a term t, while $\overline{O}(t)$ is the set of nonvariable positions of 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. 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. Identity of syntactic objects is denoted by $\equiv$. A *substitution* is a mapping from the set of variables V to the set $\tau(\Sigma \cup V)$. Given a set of equations E, $mgu(E)$ denotes the *most general unifier* of E [25].

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 $(\lambda \rightarrow \rho \Leftarrow C)$, λ, $\rho \in \tau(\Sigma \cup V)$, $\lambda \notin V$ and $Var(\rho) \subseteq Var(\lambda)$. The condition C is a (possibly empty) sequence $e_1, \ldots, e_n$, $n \geq 0$, of equations. We will often write just $\mathcal{R}$ instead of $(\Sigma, \mathcal{R})$. If a rewrite rule has no condition, we write $\lambda \rightarrow \rho$. A goal $\Leftarrow g$ is a rewrite rule with no head, and we simply denote it by g.

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 $\langle \Sigma, \mathcal{R} \rangle$, we assume that the signature Σ is partitioned into two disjoint sets $\Sigma = \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{D} = \{f \mid (f(\tilde{t}) \rightarrow r \Leftarrow C) \in \mathcal{R}\}$ 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 *constructor* terms. A pattern is a term $f(l_1, \ldots, l_n)$ such that $f \in \mathcal{D}$ and $l_1, \ldots, l_n$ are constructor terms. A term s is a *normal form*, if there is no term t with $s \rightarrow_{\mathcal{R}} t$, where $\rightarrow_{\mathcal{R}}$ denotes the (conditional) rewriting relation. We omit the subscript $\mathcal{R}$ when no confusion can arise. In the remainder of this paper, a *(functional logic) program* is a finite CTRS. The program $\mathcal{R}$ is said to be canonical if the binary one-step rewriting relation $\rightarrow_{\mathcal{R}}$ defined by $\mathcal{R}$ is noetherian and confluent [22]. A *successful conditional rewriting sequence* (also called *proof*) for a goal g in $\mathcal{R}$ (extended with the rules for the equality) is a sequence $\mathcal{D} : g \equiv g_1 \rightarrow g_2 \rightarrow \ldots \rightarrow true$.

The standard operational semantics of functional logic programs is based on narrowing [15,29], a combination of unification for parameter passing and reduction as evaluation mechanism which subsumes rewriting and SLD-resolution. Essentially, narrowing consists of the instantiation of goal variables, followed by a reduction step on the instantiated goal. Narrowing is complete in the sense of functional programming (computation of normal forms) as well as logic programming (computation of answers). Due to the huge search space of unrestricted narrowing, steadily improved strategies have been proposed. A *narrowing strategy* (or *position constraint*) φ is any well-defined criterion that obtains a smaller search space by permitting narrowing to reduce only some chosen positions. We denote by $\leadsto_{\varphi}$ the narrowing relation with strategy φ (see [19] for a survey on narrowing strategies.) $I\!R_{\varphi}$ denotes the class of CTRSs which satisfy the conditions for the completeness of the strategy φ. For instance, needed narrowing is known to be an optimal narrowing strategy for inductively sequential programs, a class of TRS's following the constructor discipline with discriminating left-hand side, that is, typical functional programs. For the completeness of "lazy strategies" such as needed narrowing, the strict equality $\approx$ is considered, which is only defined on finite and completely determined data structures, and gives to equality the weak meaning of identity of finite objects (e.g., see [26]). Hence, we also assume that equations in g and C have the form $s = t$ (where $=$ denotes the standard equality) whenever we consider "eager strategies" such as innermost conditional narrowing ($\varphi = inn$), whereas the equations have the form $s \approx t$ when we consider needed narrowing ($\varphi = needed$).

2.1 Denotation of a Functional Logic Program

In order to formulate a semantics modeling computed answers, the usual Herbrand base has to be extended to the set of all (possibly) non-ground equations modulo variance [14]. $\mathcal{H}_V$ denotes the *V-Herbrand universe* which allows variables in its elements, and is defined as $\tau(\Sigma \cup V)/_{\cong}$, where $\cong$ is the equivalence relation induced by preorder $\leq$ of "relative generality" between terms. For the sake of simplicity, the elements of $\mathcal{H}_V$ have the same representation as the elements of $\tau(\Sigma \cup V)$ and are also called terms. $\mathcal{B}_V$ denotes the *V-Herbrand base*, namely, the set of all equations $s = t$ modulo variance, where $s, t \in \mathcal{H}_V$. Note that the standard Herbrand base $\mathcal{B}$ is equal to $[\mathcal{B}_V]$.

In non-strict languages, if the compositional character of meaning has to be preserved in presence of infinite data structures and partial functions, then non-normalizable terms, which may occur as subterms within normalizable expressions, also have to be assigned a denotation. Following [18,26], we introduce a fresh constant symbol $\perp$ into Σ to represent the value of expressions which would otherwise be undefined.

In the following we recall two useful semantics for functional logic programs (we refer to [6] for details).

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

$\mathcal{O}^{ca}_{\varphi}(\mathcal{R}) = \Im^{\varphi}_{\mathcal{R}} \cup \{(f(x_1, \ldots, x_n) = x_{n+1})\theta \mid (f(x_1, \ldots, x_n) =_{\varphi} x_{n+1}) \overset{\theta}{\leadsto}{}^{*}_{\varphi} \top$ s.t. $f/n \in \mathcal{D}$, x_{n+1} and x_i are distinct variables, for $i = 1, \ldots, n\}$, where $\Im^{\varphi}_{\mathcal{R}}$ denotes the set of identical equations $c(x_1, \ldots, x_n) =_{\varphi} c(x_1, \ldots, x_n)$, c/n constructor symbol in $\mathcal{R}$.

Fixpoint Semantics. The (bottom-up) fixpoint semantics $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$, modeling computed answers w.r.t. a narrowing strategy φ, is defined as the least fixpoint $\mathcal{F}^{ca}_{\varphi}(\mathcal{R}) = T^{\varphi}_{\mathcal{R}} \uparrow \omega$ of a parametric immediate consequence operator $T^{\varphi}_{\mathcal{R}} : 2^{\mathcal{B}_V} \rightarrow 2^{\mathcal{B}_V}$ which generalizes the ground immediate consequences operator in [21] in order to model computed answers.

The relationship between the operational and fixpoint semantics is established by the following theorem.

Theorem 1. *[2] $\mathcal{O}_{\varphi}^{ca}(\mathcal{R}) = \mathcal{F}_{\varphi}(\mathcal{R}) \setminus inprogress(\mathcal{F}_{\varphi}(\mathcal{R}))$, where, for equation set S, $inprogress(S) = \{\lambda = \rho \in S \mid \bot$ occurs in ρ or ρ contains a defined function symbol of $\Sigma\}$.*

For the sake of clarity, let us summarize the relation among the two different program denotations $\mathcal{F}_{\varphi}(\mathcal{R})$ and $\mathcal{O}_{\varphi}^{ca}(\mathcal{R})$ introduced above. The compositional, fixpoint semantics $\mathcal{F}_{\varphi}(\mathcal{R})$ which models successful as well as partial (nonterminating as well as intermediate computations, i.e. those equations $f(\bar{t}) = s$ where s "has not reached its value") is obtained by computing the least fixpoint of the immediate consequences operator $T_{\mathcal{R}}^{\varphi}$. On the other hand, the operational success set semantics $\mathcal{O}_{\varphi}^{ca}(\mathcal{R})$ only catches successful derivations, that is, it models the computed answers observable.

3 Diagnosis of Declarative Programs

First we recall some basic definitions on the declarative diagnosis [11].

Definition 1. *Let $\mathcal{I}_{ca}$ be the specification of the intended success set semantics for $\mathcal{R}$. An* incorrectness symptom *is an equation e such that $e \in \mathcal{O}_{\varphi}^{ca}(\mathcal{R})$ and $e \notin \mathcal{I}_{ca}$. An* incompleteness symptom *is an equation e such that $e \in \mathcal{I}_{ca}$ and $e \notin \mathcal{O}_{\varphi}^{ca}(\mathcal{R})$.*

In case of errors, in order to determine the faulty rules, we make use of the following definitions. We need to consider a fixpoint intended semantics $\mathcal{I}_{\mathcal{F}}$, that models both successful and "in progress" computations. The relation between $\mathcal{I}_{\mathcal{F}}$ and the intended operational meaning is given by $\mathcal{I}_{ca} = \mathcal{I}_{\mathcal{F}} \setminus inprogress(\mathcal{I}_{\mathcal{F}})$.

Definition 2. *Let $\mathcal{I}_{\mathcal{F}}$ be the specification of the intended fixpoint semantics for $\mathcal{R}$. If there exists an equation $e \in T_{\{r\}}^{\varphi}(\mathcal{I}_{\mathcal{F}})$ and $e \notin \mathcal{I}_{\mathcal{F}}$, then the rule $r \in \mathcal{R}$ is incorrect on e. We also say that e is* incorrect*. Reciprocally, the equation e is uncovered if $e \in \mathcal{I}_{\mathcal{F}}$ and $e \notin T_{\mathcal{R}}^{\varphi}(\mathcal{I}_{\mathcal{F}})$.*

Since program denotations generally consist of an infinite number of equations, the above conditions for correctness and completeness of a program w.r.t. to a given specification cannot be effectively computed. In [2], an abstract diagnosis methodology based on the abstract interpretation theory [12] was proposed. Abstract diagnosis is a correct approximation of the diagnosis technique presented so far where the semantic domains and operators are replaced by abstract ones. First, we build a suitable abstract immediate consequences operator ($T_{\mathcal{R}}^{\sharp\varphi}$), which uses an abstraction of the program rules where all infinite computations have been removed and is also parametric w.r.t. the narrowing strategy. The approximation is done by using a loop-checker which replaces the calls which are (risky to be) responsible for the infinite derivations by a fresh irreducible symbol $\sharp$. The fixpoint of $T_{\mathcal{R}}^{\sharp\varphi}$ correctly approximates the fixpoint semantics of $\mathcal{R}$ and can be computed finitely. The abstract diagnosis process is performed w.r.t. two abstract (finite) semantics $\mathcal{I}^{-}$ and $\mathcal{I}^{+}$ which under- and over-approximate the intended semantics $\mathcal{I}$.

4 Correction Method

In this section, we present an inductive learning methodology which is able to repair a functional logic program containing buggy rules. The correction problem can be stated as follows. Let $\mathcal{R}$ be a program, $\mathcal{I}$ the intended specification, $\mathcal{R}' \subseteq \mathcal{R}$ a set of incorrect rules w.r.t. $\mathcal{I}$, and $E = E^+ \cup E^-$ two disjoint (ground) example sets which model the pursued (not pursued) computational behaviour. We denote by $\mathcal{R} \vdash E$ the fact that the (ground) equation set E can be reduced to *true* by using the rules of $\mathcal{R}$. Then, we want to determine a set of rules $\mathcal{X}$ such that $\mathcal{R}^c = (\mathcal{R} \setminus \mathcal{R}') \cup \mathcal{X}$, $\mathcal{R}^c \vdash E^+$ and $\mathcal{R}^c \not\vdash E^-$. Program $\mathcal{R}^c$ will be called *correct* program (w.r.t. E^+ and E^-). We will call $\mathcal{R}^- = \mathcal{R} \setminus \mathcal{R}'$ the *diminished* program. We note that $\mathcal{R} \vdash E$ can be checked, even in the case that $\mathcal{R}$ is not terminating, by using the "normalization via μ–normalization" method of [23] to compute, by levels, the 'maximal contexts' of the lhs's of the examples, and then comparing them with the ground constructor term in the corresponding rhs. By this technique, normal forms can be obtained by successively computing μ-normal forms and shifting computations to maximal non-replacing subterms when a μ-normal form has been obtained. The conditions for the completeness of this technique (*csr*–conditions) essentially amount to the termination of "context–sensitive rewriting" (*csr*) [24], which is much easier than the termination of rewriting. A *csr* practical tool for proving termination of *csr* is available at `http://www.dsic.upv.es/users/elp/slucas/muterm`.

The automatic search for a new rule in an induction process can be performed either bottom-up (i.e. from an overly specific rule to a more general) or top-down (i.e. from an overly general rule to a more specific). There are some reasons to prefer the top-down or *backward reasoning* process to the bottom–up or *forward reasoning* process [13]. On the one hand, it eliminates the need for navigating through all possible logical consequences of the program. On the other hand, it integrates inductive reasoning with the deductive process, so that the derived program is guaranteed to be correct. Unfortunately, it is known that the deductive process alone (i.e. unfolding) does not generally suffice for coming up with the corrected program, and inductive generalization techniques are necessary [13,28]. In [20,17], a bottom-up framework for synthesizing correct functional logic programs (w.r.t. the ground success set, Herbrand semantics) is presented which induces program rules from sets of equations which are respectively incorrect and correct w.r.t. the pursued program. Their methodology, however, is not particularly tailored for *theory revision*, and we need to adapt it since the uncontrolled application of the method would produce much speculation in our framework, which we want to avoid. Therefore, we follow a hybrid, top-down as well as bottom-up approach, which is able to infer program corrections that are hard, if not at all impossible, to obtain with a simple deductive learner.

4.1 Automatic Generation of Positive and Negative Example Sets

Let us present a simple method for automatically generating the example sets which exploits the debugger outcome so that the user does not need to provide error symptoms, evidences or other kind of information which would require a good knowledge of the program semantics that she probably lacks.

Consider the diminished program $\mathcal{R}^-$. Due to the absence of faulty rules in $\mathcal{R}^-$, $\mathcal{R}^-$ is already partially correct; however $\mathcal{R}^-$ might be incomplete, as there can be equations which are covered in $\mathcal{I}$, but not in $\mathcal{R}^-$.

By applying the diagnosis method presented in Section 3, we are able to find out the sets of *uncovered* and *incorrect* equations w.r.t. an abstraction of the intended semantics, respectively E_1 and E_2. Considering equations in E_1 seems a sensible way for yielding *positive* examples (missing proofs which should be achieved by $\mathcal{R}$). On the other hand, set E_2 contains equations modeling erroneous behaviours, thus we can take them as *negative* examples.

Since E_1 and E_2 might contain non-ground equations, we find it useful to instantiate (a subset of) them in order to get ground positive/negative example sets E^+ and E^-. This allows us to perform some standard optimizations based on term rewriting which are very satisfactory in practice. On the other hand, since program $\mathcal{R}$ and specification $\mathcal{I}$ might use different auxiliary functions, we only consider ground examples of the form $l = d$ where l is a pattern and d is a constructor term. In this way, the inductive process becomes independent from the extra functions contained in $\mathcal{I}$, since we start synthesizing directly from data structures d. In order to achieve this, we normalize the term in the rhs of (the instantiated) examples. Finally, we disregard those examples which, after normalization, do not have a constructor term at the rhs.

4.2 Specialization Operators

Roughly speaking, *unfolding* a program $\mathcal{R}$ w.r.t. a rule r delivers to 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 rhs or the conditional part of r.

Definition 3 (unfolding). *Let $\mathcal{R}$ be a CTRS and $r \equiv (\lambda \to \rho \Leftarrow C) \ll \mathcal{R}$ be a rule. Let $\{g \overset{\theta_i}{\leadsto}_{\varphi} (C'_i, \rho'_i = y)\}_{i=1}^{n}$ be the set of all one-step narrowing derivations with strategy φ that perform an effective narrowing step for the goal $g \equiv (C, \rho = y)$ in $\mathcal{R}$. Then, $Unf^{\varphi}_{\mathcal{R}}(r) = \{(\lambda\theta_i \to \rho'_i \Leftarrow C'_i) | i = 1 \dots n\}$ (that is, the derived goal $(C'_i, \rho'_i = y)$ is different from g.*

Definition 4 (Unfolding operator). *Let $\mathcal{R}$ be a CTRS, $r \equiv \lambda \to \rho \Leftarrow C$ be a rule in $\mathcal{R}$. The* Rule Unfolding *operator $\mathsf{U}^{\varphi}_{r}(\mathcal{R})$ on $\mathcal{R}$ w.r.t. r is defined by $\mathsf{U}^{\varphi}_{r}(\mathcal{R}) = \mathcal{R} \setminus \{r\} \cup Unf^{\varphi}_{\mathcal{R}}(r)$.*

As it has been proven in [4,5], for $\varphi = inn, needed$, unfolding using strategy φ preserves the semantics (even for the observable of computed answers) in $I\!R_{\varphi}$ programs. When needed narrowing is considered, completeness is only guaranteed under the condition that expressions in the rule are not unfolded beyond their head normal form [5]. On the other hand, the absence of narrowable positions in the rule to be unfolded yields no specialization of r. We just get the removal of r from $\mathcal{R}$. Therefore, we use the following notion of “unfoldable rule”.

Definition 5. *Let $\mathcal{R}$ be a CTRS, r be a rule in $\mathcal{R}$. The rule r is* unfoldable *w.r.t. $\mathcal{R}$ if $\mathsf{U}_r^{\varphi}(\mathcal{R}) \neq \mathcal{R} \setminus \{r\}$. If $\varphi =$ needed, we also require that r is not unfolded beyond its head normal form.*

For the sake of simplicity, in the following we omit φ whenever this does not compromise readability. The *unfolding succession* $\mathcal{S}(\mathcal{R}) \equiv \mathcal{R}_0, \mathcal{R}_1, \ldots$ of program $\mathcal{R}$ is defined as follows: $\mathcal{R}_0 = \mathcal{R}$, $\mathcal{R}_{i+1} = \mathsf{U}_r(\mathcal{R}_i)$ where $r \in \mathcal{R}_i$ is unfoldable.

4.3 Top-Down Correction Algorithm

Following [9], the algorithm below works in two phases: the *unfolding phase* and the *deletion phase*. Roughly speaking, we first perform unfolding upon (arbitrarily selected) unfoldable rules, until we get a specialized version of the program $\mathcal{R}$ where no negative example can be proven by applying only rules used in proofs of positive examples. The following definition is auxiliary.

Definition 6. *Given $\mathcal{D} : g \equiv g_1 \xrightarrow{r_1} g_2 \xrightarrow{r_2} \ldots \xrightarrow{r_n} g_n$, the sequence $\langle r_1, r_2, \ldots, r_n \rangle$ is called the* rewriting rule sequence *of $\mathcal{D}$. The set $\mathsf{OR}(\mathcal{D}) = \{r_1, r_2, \ldots, r_n\}$ is called the set of* occurring rules *of $\mathcal{D}$.*

Given an equation e, let $\mathcal{D}_{\mathcal{R}}^{\varphi}(e)$ denote the successful rewrite sequence which proves e in program $\mathcal{R}$ (if it exists) by using a normalizing rewriting strategy for the class $I\!\!R_{\varphi}$. The key idea of the algorithm is thus applying unfolding until we get a specialized program $\mathcal{R}_i$ satisfying that, for each $e^- \in E^-$ there exists a rule $r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}^{\varphi}(e^-))$ such that, for each example $e^+ \in E^+$, $r \notin \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}^{\varphi}(e^+))$. Now, since the rules which only contribute to the proof of negative examples are useless, in the subsequent phase we just remove these rules from the specialized program $\mathcal{R}_i$. By *discriminable rule* of $\mathcal{R}_i$ we mean an unfoldable rule of $\mathcal{R}_i$ which occurs in the proof of, at least, one positive and one negative example.

```
**Algorithm** $TD\text{-}Corrector(\mathcal{R}, \mathcal{I})$
$(E^+, E^-)$=GenerateExampleSets$(\mathcal{R}, \mathcal{I})$
**if** $\mathcal{R} \not\vdash E^+$ **then** halt
{Unfolding phase}
**let** $i = 0$; $\mathcal{R}_0 = \mathcal{R}$
**while** $\exists\, e^- \in E^-$ s.t. $\forall r (r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^-)) \Rightarrow \exists e^+ \in E^+$ s.t. $r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^+)))$ **do**
        **select** a discriminable rule $r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^-))$ of $\mathcal{R}_i$
        **let** $\mathcal{R}_{i+1} = \mathsf{U}_r(\mathcal{R}_i)$; $i = i + 1$
**end while**
{Deletion phase}
**for each** $e^- \in E^-$ **do**
     **let** $\mathcal{R}_{i+1} = \mathcal{R}_i \setminus \{r\}$, where $r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^-)) \wedge \forall e^+ \in E^+\ r \notin \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^+))$
     **let** $i = i + 1$
**end for**
**let** $\mathcal{R}^c = \mathcal{R}_i$
```

Example 2. Consider again the program $\mathcal{R}$ and specification $\mathcal{I}$ of example 1, with the example sets for learning function *od*. Since the rewriting proof for the negative example $od(s^2(0)) = true \in E^-$ uses the rule $od(s(X)) \rightarrow od(X)$ (either with $\varphi = inn$ or $\varphi = needed$), which is also used in the proofs of positive examples, we enter the main loop. By unfolding $od(s(X)) \rightarrow od(X)$ we get $\mathcal{R}_1 = \{od(0) \rightarrow true, od(s(0)) \rightarrow true, od(s^2(X)) \rightarrow od(X)\ \}$. Now we enter the deletion phase which purifies $\mathcal{R}_1$ by removing the rule $od(0) \rightarrow true$ that only occurs in the proof of a negative example, thus producing the expected correction shown in Section 1.

Example 2 allows us to clarify the differences between the preliminary correction algorithm in [2] and the one in this paper. The algorithm in [2] was based on unfolding the rules which incorrectly cover the negative examples. In our example, this could result in trying to unfold the rule $od(0) \rightarrow true$, which is fruitless, whereas the new correction procedure does consider any discriminable rule for unfolding, which is generally needed in order to achieve the desired correction.

We prove the correctness of the top-down correction algorithm in two steps: first we show that, provided that $\mathcal{R}$ covers E^+, the unfolding phase produces a specialized version $\mathcal{R}'$ of $\mathcal{R}$ (still covering E^+) such that, for each negative example, there is a rule occurring in the corresponding proof which is not used in the proof of any of the positive examples. Next, we demonstrate that the deletion phase yields a corrected version of $\mathcal{R}$ covering E^+ and not covering E^-.

The following proposition states our first result: by a suitable finite number of applications of the unfolding operator to a program in $I\!\!R_\varphi$, we get a specialized version such that, in any successful rewriting derivation of a negative example, there occurs a rule that is not applied in any successful rewriting derivation for the positive examples under the same strategy. A condition is necessary for proving this result: no negative/positive couple of the considered examples can have the same rewriting rule sequence, as shown in the following counterexample.

Example 3. Consider the program $\mathcal{R} = \{f(X) \rightarrow g(X), g(X) \rightarrow 0\}$ with example sets $E^+ = \{f(a) = 0\}$, $E^- = \{f(b) = 0\}$. Then $f(a) = 0$ and $f(b) = 0$ are proven by using the same rewriting rule sequence (using any of the considered rewriting strategies). By applying the top–down algorithm, we unfold rule $f(X) \rightarrow g(X)$, which produces the outcome $\mathcal{R}_1 = \{f(X) \rightarrow 0, g(X) \rightarrow 0\}$ which cannot be purified (by using the rule deletion operator) as removing rule $f(X) \rightarrow 0$ in order to get rid of E^- would cause losing E^+.

Proposition 1. *Let φ be a normalizing rewriting strategy for $I\!\!R_\varphi$ and $\mathcal{R}$ be a program in $I\!\!R_\varphi$. Let E^+ (resp. E^-) be a set of positive (resp. negative) examples. If there are no $e^+ \in E^+$ and $e^- \in E^-$ which can be proven in $\mathcal{R}$ by using the same rules sequence, then, for each unfolding succession $\mathcal{S}(\mathcal{R})$, there exists k such that $\forall e^- \in E^- \exists r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_k}(e^-))$ s.t. r is not discriminable*

We note that Proposition 1 holds for every unfolding succession of the original program; this implies that the rule to be unfolded at each unfolding step can be arbitrarily selected, provided that it is discriminable. Moreover, the termination of the unfolding phase is granted by the finite number k of applications of the unfolding operator that we need to obtain specialization $\mathcal{R}_k$.

After the unfolding phase, the refutation of every negative example contains a rule from $\mathcal{R}_k$ not occurring in the proof of any positive example, thus we can safely remove this rule without jeopardizing completeness. The deletion phase purifies $\mathcal{R}_k$ and yields correctness w.r.t. both positive and negative examples.

Theorem 2 (Correctness). *Let $\mathcal{R} \in I\!R_\varphi$ which satisfies the csr conditions, E^+ and E^- be two sets of examples such that $\mathcal{R} \vdash E^+$. If the rewriting rule sequences for $e^+ \in E^+$ and $e^- \in E^-$ are different, then the TD-Corrector algorithm yields a correct specialization of $\mathcal{R}$ w.r.t. E^+ and E^-.*

As in other approaches for example-guided program correction, the above result does not generally imply that a correction for the wrong program $\mathcal{R}$ w.r.t. the intended semantics is obtained as the outcome of the top-down correction algorithm (that is, a program $\mathcal{R}$ with the same semantics of $\mathcal{I}$, up to the extra auxiliary function symbols which might appear in $\mathcal{I}$), under the conditions required for the correctness of the algorithm, but it might happen that the output program is only correct w.r.t. E^+ and E^-. Therefore, derived programs would be newly diagnosed for correctness at the end.

5 Improving the Algorithm

In the following, we propose a bottom-up correction methodology which we smoothly combine with the deductive one in order to correct programs which do not fulfil the applicability condition (over–generality). Therefore, the methodology just consists of applying a bottom-up pre–processing to "generalize" the initial wrong program, before proceeding to the top-down correction.

5.1 Bottom-up Generation of Overly General (Wrong) Programs

We propose a methodology which is based on extending the original program with new rules, so that the entire set E^+ succeeds w.r.t. the generalized program, and hence the top-down corrector can be effectively applied.

Our generalization method is based on a simplified version of the bottom-up technique for the inductive learning of functional logic programs developed by Ferri, Hernández and Ramírez [17] which is able to produce an intensional description (expressed by a functional logic program) of a set of ground examples. The algorithm is also able to introduce functions, defined as a background theory, in the inferred intensional description (see [17,20] for details). In the following we recall the definitions of *restricted generalization* and *inverse narrowing* which are the heart of the bottom-up procedure of [17,20]. The former allows to generalize program rules, the latter is needed to introduce defined symbols in the right hand sides of the synthesized rules.

Definition 7 (Generalization operator). *The rule $r' \equiv (s' \rightarrow t' \Leftarrow C')$ is a* restricted generalization *of $r \equiv (s \rightarrow t \Leftarrow C)$ if there exists a substitution θ such* *that (i) $\theta(r') \equiv r$; (ii) $Var(t') \subseteq Var(s')$. The* generalization operator $\mathsf{RG}(r)$ *is defined as follows:* $\mathsf{RG}(r) = \{r' | r'$ *is a restricted generalization of* $r\}$.

Definition 8 (Inverse narrowing operator). *The rule* $r \equiv s \to t \Leftarrow C$ reversely narrows *into* $r' \equiv s \to t' \Leftarrow C'$ *(in symbols* $r \overset{u,r'',\theta}{\hookleftarrow} r'$*) iff there exist a position* $u \in O(t)$ *and a rule* $r'' \equiv \lambda \to \rho \Leftarrow C''$ *such that (i)* $\theta = mgu(t_{|u}, \rho)$*; (ii)* $t' = (t[\lambda]_u)\theta$*; (iii)* $C' = (C'', C)\theta$.
The inverse narrowing *operator* $\mathsf{INV}(r, r'')$ *is given by:*

$$\mathsf{INV}(r, r'') = \{r' \mid r \overset{u,r'',\theta}{\hookleftarrow} r' \textit{ and extra-variables in the rhs of } r' \textit{ are instantiated to variables in the rhs of } r\}.$$

The extra instantiation of variables in the rhs of the derived rules is necessary, since inverse narrowing considers the rules oriented reversely and hence extra-variables might be introduced in the synthesized rules, which is not allowed.

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

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

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

Due to the huge search space which this method involves, some heuristics must be implemented to guide the search. Following [20], *Minimum Description Length*[1] (MDL) and *Covering Factor*[2] criteria could be taken into consideration,

[1] *length*$(e) = 1 + n_v/2 + n_f$, where n_v and n_f are the number of variables and functors in the rhs of e.

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

so that inverse narrowing steps are only performed among the best programs and equations w.r.t. these criteria. Moreover, by means of MDL and Covering Factor, only the most concise programs are selected during the induction process. The notion of *Optimality* w.r.t. programs and equations could be defined as a linear combination of these two criteria. For a full discussion see [20]. A detailed description of the algorithm can be retrieved in [1]. Let us consider an example, in which we only pinpoint the relevant outcomes for the sake of clarity.

Example 4. Consider the following (wrong) program and the specification

$$\begin{aligned}\mathcal{R} = \{\, & playdice(X) \to double(winface(X)), dd(0) \to 0, \mathbf{dd(s(X)) \to dd(X)}, \\ & \mathbf{winface(s(X)) \to s(winface(X)), winface(0) \to 0} \,\}\end{aligned}$$

$$\begin{aligned}\mathcal{I} = \{\, & playdice(X) \to dd(winface(X)), dd(X) \to sum(X,X), \\ & sum(X,0) \to X, sum(X,s(Y)) \to s(sum(X,Y)), \\ & winface(s(0)) \to s(0), winface(s(s(0))) \to s(s(0)) \,\}.\end{aligned}$$

Program rules marked in boldface are signalled as incorrect by the diagnosis system. The example generation procedure described in Section 4.1 produces:

$$\begin{aligned}E^+ = \{\, & playdice(s^2(0)) = s^4(0), playdice(s(0)) = s^2(0), dd(s^4(0)) = s^8(0), \\ & dd(s^3(0)) = s^6(0), dd(s^2(0)) = s^4(0), dd(s(0)) = s^2(0) \\ & dd(0) = 0, winface(s^2(0)) = s^2(0), winface(s(0)) = s(0) \,\}.\end{aligned}$$

The analysis for *dd* and *winface* determines that *dd* is overspecialized. The generalization algorithm removes the rule $dd(s(X)) \to dd(X)$. Note that rule $dd(s(0)) \to s^2(0)$ inversely narrows to rule $r_{dd} \equiv dd(s(0)) \to s^2(dd(0))$ by using rule $dd(0) \to 0$. The following restricted generalizations of rule r_{dd} are computed: $dd(s(0)) \to s^2(dd(0))$, $dd(s(X)) \to s^2(dd(0))$, $dd(s(X)) \to s^2(dd(X))$. Now, when the third rule is added to the background knowledge, all the examples in E^+ are covered. Thus, the following overly general definition, which feeds the top-down corrector in order to repair the remaining errors, is delivered

$$\begin{aligned}\mathcal{R} = \{\, & playdice(X) \to dd(winface(X)), dd(0) \to 0, dd(s(X)) \to s(s(dd(X)))), \\ & \mathbf{winface(s(X)) \to s(winface(X)), winface(0) \to 0} \,\}.\end{aligned}$$

6 Automated Correction System

A prototypical implementation of our methodology and a full experimental evaluation are available at `http://www.dsic.upv.es/users/elp/soft.html`. We have improved the preliminary debugging system Buggy by adding the new features. The current implementation, called Nobug, is now able to compute sets of positive and negative examples by using the methodology described in Section 4.1. Besides, we have developed a full implementation of the top-down correction method based on example-guided unfolding for the leftmost-innermost narrowing strategy. We are currently implementing the lazy version of the algorithm. The bottom-up synthesis method has not been integrated into the Nobug system yet. Hence, in order to compute our benchmarks also for initially overspecialized programs, we used the inductive functional logic system FLIP[16].

We have performed some tests by means of our top-down corrector and the bottom-up learner FLIP, in order to repair overly general as well as overspecialized functional logic programs. We have taken into account programs on several domains: naturals, lists and finite domains. In order to systematize the generation of the benchmarks, we have slightly modified correct programs in order to obtain wrong program mutations. We were able to successfully repair incorrect mutations, achieving, in many cases, a correction both w.r.t. the example sets and the intended program semantics.

7 Conclusions

In this paper we have proposed a new methodology for synthesizing (partially) correct functional logic programs which complements the diagnosis method we developed previously in [6,2]. Our methodology is based on the combination, in a single framework, of a diagnoser [6,2] which identifies those parts of the code containing errors, together with a program learner which, once the bug has been located in the program, tries to repair it starting from evidence examples (uncovered as well as incorrect equations) which are essentially obtained as an outcome of the diagnoser. We follow a hybrid, *deductive* (top-down) as well as *inductive* (bottom-up) approach, which is able to infer program corrections that are hard to obtain with a simple (pure deductive or inductive) program learner. We plan to generalize the framework to other paradigms as future work.

Finally, we want to emphasize that this framework supersedes the preliminary approach of [2]. In [2], recursive definitions were sometimes impossible to repair, and no automated correction is provided for overspecialized programs either, whereas the new methodology in this paper overcomes both drawbacks.

References

1. M. Alpuente, D. Ballis, F. J. Correa, and M. Falaschi. Correction of Functional Logic Programs. Technical report, DSIC-II/23/02, UPV, 2002. Available at URL: `http://www.dsic.upv.es/users/elp/papers.html`.
2. M. Alpuente, F. J. Correa, and M. Falaschi. Debugging Scheme of Functional Logic Programs. In *Proc. of WFLP'01*, vol. 64 of *ENTCS*, 2002.
3. M. Alpuente, F. J. Correa, M. Falaschi, and S. Marson. The Debugging System BUGGY. Technical report, UPV, 2001. Available at URL: `http://www.dsic.upv.es/users/elp/soft.html`.
4. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe folding/unfolding with conditional narrowing. In *Proc. ALP'97*, pp. 1–15. Springer LNCS 1298, 1997.
5. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs. In *Proc. of FLOPS'99*, pp. 147–162. Springer LNCS 1722, 1999.
6. M. Alpuente, F. Correa, and M. Falaschi. Declarative Debugging of Funtional Logic Programs. In *Proc. of WRS'01*, vol. 57 of *ENTCS*, 2001.

7. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
8. H. Bostrom and P. Idestam-Alquist. Induction of Logic Programs by Example–guided Unfolding. *Journal of Logic Programming*, 40:159–183, 1999.
9. Henrik Bostrom. Specialization of recursive predicates. In *European Conference on Machine Learning*, pp. 92–106, 1995.
10. M. Comini, R. Gori, and G. Levi. Assertion based Inductive Verification Methods for Logic Programs. In *Proc. of MFCSIT'00*, vol. 40 of *ENTCS*, 2001.
11. M. Comini, G. Levi, and G. Vitiello. Declarative Diagnosis Revisited. In *Proc. of ILP'95*, pp. 275–287. The MIT Press, 1995.
12. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pp. 238–252, 1977.
13. N. Dershowitz and U. Reddy. Deductive and Inductive Synthesis of Equational Programs. *Journal of Symbolic Computation*, 15:467–494, 1993.
14. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *TCS*, 69(3):289–318, 1989.
15. M. Fay. First Order Unification in an Equational Theory. In *Proc of 4th Int'l Conf. on Automated Deduction*, pp. 161–167, 1979.
16. C. Ferri, J. Hernández, and M.J. Ramírez. The FLIP System Homepage. 2000. Available at http://www.dsic.upv.es/users/elp/soft.ht ml.
17. C. Ferri, J. Hernández, and M.J. Ramírez. Incremental Learning of Functional Logic Programs. In *Proc. FLOPS'01*, pp. 233–247. LNCS 2024, 2001.
18. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *JCSS*, 42:363–377, 1991.
19. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
20. J. Hernández and M.J. Ramírez. Inverse Narrowing for the Induction of Functional Logic Programs. In *Proc. of APPIA–GULP–PRODE '98*, pp. 379–393, 1998.
21. S. Hölldobler. *Foundations of Equational Logic Programming.* LNAI 353, 1989.
22. J.W. Klop. Term Rewriting Systems. In *Handbook of Logic in Computer Science*, vol. I, pp. 1–112. Oxford University Press, 1992.
23. S. Lucas. Context-Sensitive Rewriting Strategies. *Information and Computation*, 178(1):294–343, 2002.
24. S. Lucas. Termination of Canonical Context-Sensitive Rewriting. In *Proc. RTA'02*, pp. 296–310. Springer LNCS 2378, 2002.
25. M.J. Maher. Equivalences of Logic Programs. In *Foundations of Deductive Databases and Logic Programming*, pp. 627–658. Morgan Kaufmann, 1988.
26. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *JLP*, 12(3):191–224, 1992.
27. P. Padawitz. *Computing in Horn Clause Theories*, vol. 16 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1988.
28. A. Pettorossi and M. Proietti. Transformation of Logic Programs. In *Handbook of Logic in Artificial Intelligence*, vol. 5, pp. 697–787. Oxford University Press, 1998.
29. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. of Second IEEE Int'l Symp. on Logic Programming*, pp. 138–151, 1985.