Declarative Debugging of Functional Logic Programs *

M. Alpuente[1] F. J. Correa[2] M. Falaschi[3]

[1] DSIC, U. Politécnica Valencia, Camino de Vera s/n, 46022 Valencia, Spain.
alpuente@dsic.upv.es
[2] DIS, U. EAFIT , Cra. 49 N 7 Sur 50, 3300 Medellín, Colombia.
fcorrea@eafit.edu.co
[3] Dipartimento di Matematica e Informatica, University of Udine,
Via delle Scienze 206, 33100 Udine, Italy.
falaschi@dimi.uniud.it

Abstract. We present a general framework for the declarative debugging of functional logic programs, which is valid both for eager as well as lazy programs. We associate to our programs a semantics based on a (continuous) immediate consequence operator which models computed answers. Then we show that, given the intended specification of a program P, it is possible to check the correctness of P by a single step of the immediate consequence operator.

We also present a more effective methodology which is based on abstract interpretation. By approximating the intended specification of the success set we derive a finitely terminating debugging method, which can be used statically. Our framework is parametric w.r.t. to the chosen approximation of the success set. We present one specific example of approximation. We provide an implementation of our debugging system which shows experimentally on a wide set of benchmarks that we are able to find some common errors in the user programs.

1 Introduction

Declarative programming is supported both by functional and logic programming. However, each of these programming styles has different advantages w.r.t. practical applications. Functional languages provide sophisticated abstraction facilities, module systems and clean solutions for integrating I/O into declarative programming as well as for efficient program execution. Logic languages allow for computing with partial information and provide built-in search facilities which have strong applications for knowledge-based systems and operations research. However, recent results show that the advantages of these styles can be efficiently and usefully combined into a single language. Modern functional logic languages offer features from both styles. The operational semantics of integrated languages is usually based on *narrowing*, a combination of unification for parameter passing and reduction as evaluation mechanism which subsumes

* This work has been partially supported by CICYT under grant TIC 98-0445-C03-01.

rewriting and SLD-resolution. Essentially, narrowing consists of the instantiation of goal variables, followed by a reduction step on the instantiated goal. Narrowing is complete in the sense of functional programming (computation of normal forms) as well as logic programming (computation of answers). Due to the huge search space of unrestricted narrowing, steadily improved strategies have been proposed (see [29] for a survey.)

How to debug functional logic programs is an important practical problem which has hardly been addressed in the previous literature. Only a few functional logic languages are equipped with a debugging tool (e.g. ALF [28], Babel [39] and Curry [31]). However, these debuggers consist of tracers which are based on suitable extended box models which help display the execution [30, 5]. Due to the complexity of the operational semantics of (functional) logic programs, the information obtained by tracing the execution is difficult to understand. The functional logic programming language NUE-Prolog is endowed with a declarative debugger [40] which works in the style proposed by Shapiro [43], that is, 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 similar declarative debugger for the functional logic language Escher is proposed in [36]. Following the generic scheme which is based on proof trees of [41], a procedure for the declarative debugging of wrong answers is given in [12] for lazy functional logic languages. The methodology in [12] includes a formalization of computation trees which is precise enough to prove the logical correctness for the debugger and which also helps simplify oracle questions.

In the case of pure logic programming, [17] has defined a declarative framework for debugging which extends the methodology in [25, 43] to diagnosis w.r.t. computed answers. The framework does not require the determination the symptoms in advance and is goal independent –it is driven by a set of most general atomic goals. It is based on using the immediate consequences operator T_P to identify program bugs and has the advantage of giving a symptom–independent diagnosis method [17, 16].

In this paper, one of the contributions is to develop a declarative diagnosis method w.r.t. computed answers which generalizes the ideas of [17] to the diagnosis of functional logic programs. The conditions which we impose on the programs which we consider allow us to define a framework for declarative debugging which works for both eager (*call–by–value*) narrowing as well as for lazy (*call–by–name*) narrowing. We associate a (continuous) immediate consequence operator to our programs. Then we show that, given the intended specification $\mathcal{I}$ of a program $\mathcal{R}$, we can check the correctness of $\mathcal{R}$ by a single step of this operator. We illustrate this through examples. We discuss the use of our work for both bottom-up as well as top-down abstract debugging of mixed functional logic code.

We also present a novel, efficient methodology which is based on abstract interpretation. We proceed by approximating the intended specification of the success set. Following an idea inspired in [17, 16, 11], we use *over* and *under* specifications $\mathcal{I}^+$ and $\mathcal{I}^-$ to correctly over- (resp. under-) approximate the intended semantics. We then use these two sets respectively for the functions in the premises and the consequence of the immediate consequence operator, and by a simple static test we can determine whether some of the clauses are wrong.

Plan of the paper The rest of the paper is organized as follows. Section 2 briefly presents some preliminary definitions and notations. Section 3 first formulates a novel, generic immediate consequence operator $T_{\mathcal{R}}^{\varphi}$ for functional logic program $\mathcal{R}$ which is parametric w.r.t. the narrowing strategy φ which can be either eager or lazy. We then define a fixpoint semantics based on $T_{\mathcal{R}}^{\varphi}$ which correctly models the answers computed by a narrower which uses the narrowing strategy φ. In the case of the eager strategy, it is enough to introduce a flattening transformation which eliminates nesting calls and allows goals to run in the semantics by standard unification. However, the lazy strategy is more involved and we need to introduce two kinds of equality in the definition of $T_{\mathcal{R}}^{\varphi}$: the strict equality which $\approx$ models the equality on data terms, and the non–strict $=$ which holds even if the arguments are both undefined or partially defined, similarly to [27]. We also formulate a semantics $\mathcal{O}^{\varphi}(\mathcal{R})$ and we show the correspondence with the fixpoint semantics. In section 4, we introduce the necessary general notions of incorrectness and insufficiency symptoms and uncovered calls. Section 5 provides an abstract semantics which correctly approximates the fixpoint semantics of $\mathcal{R}$. In Section 6, we present our method of abstract diagnosis and illustrate its use through examples. In Section 7, we present an experimental evaluation of the method on a set of benchmarks. Section 8 concludes and discusses some related work.

2 Preliminaries

We briefly summarize some known results about rewrite systems [6, 34] and functional logic programming (see [29, 33] for extensive surveys). Throughout this paper, V will denote a countably infinite set of variables and Σ denotes a set of function symbols, or signature, each of which has a fixed associated arity. $\tau(\Sigma \cup V)$ and $\tau(\Sigma)$ denote the non-ground word (or term) algebra and the word algebra built on $\Sigma \cup V$ and Σ, respectively. $\tau(\Sigma)$ is usually called the Herbrand universe ($\mathcal{H}_{\Sigma}$) over Σ and it will be denoted by $\mathcal{H}$. $\mathcal{B}$ denotes the Herbrand base, namely the set of all ground equations which can be built with the elements of $\mathcal{H}$. A Σ-equation $s = t$ is a pair of terms $s, t \in \tau(\Sigma \cup V)$.

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. $\bar{O}(t)$ denotes the set of nonvariable positions of a term t. $t_{|u}$ is the subterm at the position u of t. $t[r]_u$ is the term t with the subterm at the position u replaced with r. These notions extend to sequences of equations in a natural way. For instance, the nonvariable position set of a sequence of equations $g \equiv (e_1, \ldots, e_n)$ can be defined as follows: $\overline{O}(g) = \{i.u \mid u \in \overline{O}(e_i), i = 1, \ldots, n\}$. By $Var(s)$ we denote the set of variables occurring in the syntactic object s, while $[s]$ denotes the set of ground instances of s. A *fresh* variable is a variable that appears nowhere else. The symbol $\tilde{}$ denotes a finite sequence of symbols. Identity of syntactic objects is denoted by $\equiv$.

Let Eqn denote the set of possibly existentially quantified finite sets of equations over terms [14]. We write $E \leq E'$ if E' logically implies E. Thus Eqn is a lattice ordered by $\leq$ with bottom element $true$ and top element $fail$. The elements of Eqn are regarded as (quantified) conjunctions of equations and treated modulo logical equivalence. An equation set is *solved* if it is either $fail$ or it has the form $\exists y_1 \ldots \exists y_m.\ \{x_1 = t_1, \ldots, x_n = t_n\}$, where each x_i is a distinct variable not occurring in any of the terms t_i and each y_i occurs in some t_j. Any set of equations E can be transformed into an equivalent one, $solve(E)$, which is solved. We restrict our interest to the set of idempotent substitutions over $\tau(\Sigma \cup V)$, which is denoted by Sub. The empty substitution is denoted by ϵ. There is a natural isomorphism between substitutions $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ and unquantified equation sets $\widehat{\theta} = \{x_1 = t_1, \ldots, x_n = t_n\}$. A substitution $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ is a *unifier* of an equation set E iff $\widehat{\theta} \Rightarrow E$. We let $mgu(E)$ denote the *most general unifier* of the unquantified equation set E. We write $mgu(\{s_1 = t_1, \ldots, s_n = t_n\}, \{s'_1 = t'_1, \ldots, s'_n = t'_n\})$ to denote the most general unifier of the set of equations $\{s_1 = s'_1, t_1 = t'_1, \ldots, s_n = s'_n, t_n = t'_n\}$. We write $\theta_{\upharpoonright s}$ to denote the restriction of the substitution θ to the set of variables in the syntactic object s.

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, $\rho \in \tau(\Sigma \cup V)$, $\lambda \notin V$ and $Var(\rho) \subseteq Var(\lambda)$. The condition C is a (possibly empty) sequence $e_1, \ldots, e_n$, $n \geq 0$, of equations which we handle as a set (conjunction) when we find it convenient. Variables in C that do not occur in λ are called *extra-variables*. We will often write just $\mathcal{R}$ instead of $(\Sigma, \mathcal{R})$. If a rewrite rule has no condition, we write $\lambda \to \rho$. 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 $\mathcal{D} = \{f \mid (f(\tilde{t}) \to r \Leftarrow C) \in \mathcal{R}\}$ and $\mathcal{C} = \Sigma \setminus \mathcal{D}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions*.

A rewrite step is the application of a rewrite rule to an expression. A term s *conditionally rewrites* to a term t, $s \to_{\mathcal{R}} t$, if there exist $u \in O(s)$, $(\lambda \to \rho \Leftarrow s_1 = t_1, \ldots, s_n = t_n) \in \mathcal{R}$, and substitution σ such that $s_{|u} \equiv \lambda\sigma$, $t \equiv s[\rho\sigma]_u$, and for all $i \in \{1, \ldots, n\}$ there exists a term w_i such that $s_i \to^*_{\mathcal{R}} w_i$ and $t_i \to^*_{\mathcal{R}} w_i$, where $\to^*_{\mathcal{R}}$ is the transitive and reflexive closure of $\to_{\mathcal{R}}$. When no confusion can arise, we omit the subscript $\mathcal{R}$. A term s is a *normal form*, if there is no term t with $s \to_{\mathcal{R}} t$. The program $\mathcal{R}$ is said to be canonical if the binary one-step rewriting relation $\to_{\mathcal{R}}$ defined by $\mathcal{R}$ is noetherian and confluent [34].

An equational Horn theory $\mathcal{E}$ consists of a finite set of equational Horn clauses of the form $(\lambda = \rho) \Leftarrow C$. An equational goal is a sequence of equations $\Leftarrow C$, i.e. an equational Horn clause with no head. We usually leave out the $\Leftarrow$ symbol when we write goals. A goal of the form $\Leftarrow x = y$, with $x, y \in V$ is called a *trivial goal.* A Horn equational theory $\mathcal{E}$, satisfying $\lambda \notin V$ and $Var(\rho) \subseteq Var(\lambda)$ for each clause $(\lambda = \rho) \Leftarrow C$, can be viewed as a CTRS $\mathcal{R}$, where the rules are the heads (implicitly oriented from left to right) and the conditions are the respective bodies.

Each equational Horn theory $\mathcal{E}$ generates a smallest congruence relation $=_{\mathcal{E}}$ called *$\mathcal{E}$-equality* on the set of terms $\tau(\Sigma \cup V)$ (the least equational theory which contains all logic consequences of $\mathcal{E}$ under the entailment relation $\models$ obeying the axioms Eq[1] of the equality for $\mathcal{E}$). $\mathcal{E}$ is a presentation or axiomatization of $=_{\mathcal{E}}$. In abuse of notation, we sometimes speak of the equational theory $\mathcal{E}$ to denote the theory axiomatized by $\mathcal{E}$. We will denote by $\mathcal{H}/\mathcal{E}$ the finest partition $\tau(\Sigma)/=_{\mathcal{E}}$ induced by $=_{\mathcal{E}}$ over the set of ground terms $\tau(\Sigma)$. $\mathcal{H}/\mathcal{E}$ is usually called the *initial algebra* of $\mathcal{E}$ [19]. Satisfiability in $\mathcal{H}/\mathcal{E}$ is called *$\mathcal{E}$-unifiability*, that is, given a set of equations E, E is $\mathcal{E}$-unifiable iff there exists a substitution σ such that $\mathcal{E} \models E\sigma$ [19]. The substitution σ is called an $\mathcal{E}$-unifier of E. Let $=_{\mathcal{R}}$ be the reflexive, symmetric, and transitive closure of $\rightarrow_{\mathcal{R}}$. If $\mathcal{E}$ is the set of (conditional) equations corresponding to $\mathcal{R}$, then $=_{\mathcal{E}}$ and $=_{\mathcal{R}}$ coincide. Via this correspondence, the notion of $\mathcal{R}$-unification is implicitly defined. We say that $\theta \leq_R \sigma\ [W]$ if there is a substitution γ such that $\theta\gamma =_{\mathcal{R}} \sigma\ [W]$, i.e. $x\theta\gamma =_{\mathcal{R}} x\sigma$ for all $x \in W$.

2.1 Functional Logic Programming

Functional logic languages are extensions of functional languages with principles derived from logic programming [42]. The computation mechanism of functional logic languages is based on *narrowing*, a generalization of term rewriting where unification replaces matching: both the rewrite rule and the term to be rewritten can be instantiated. Under the narrowing mechanism, functional programs behave like logic programs: narrowing solves equations by computing solutions with respect to a given CTRS, which is henceforth called the "program".

Definition 1. (Narrowing) *Let $\mathcal{R}$ be a program and g be an equational goal. We say that g conditionally narrows into g' if there exists a position $u \in \overline{O}(g)$, a standardized apart variant $r \equiv (\lambda \rightarrow \rho \Leftarrow C)$ of a rewrite rule in $\mathcal{R}$, and a substitution σ such that: σ is the most general unifier of $g_{|u}$ and λ, and $g' \equiv (C, g[\rho]_u)\sigma$. We write $g \overset{[u,r,\sigma]}{\leadsto} g'$ or simply $g \overset{\sigma}{\leadsto} g'$. The relation $\leadsto$ is called (*unrestricted *or* full*) conditional narrowing.*

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 $\theta = \theta_1 \ldots \theta_n$. We say that the derivation has length n. If $n = 0$, then $\theta = \epsilon$. The extension of a CTRS $\mathcal{R}$ with the rewrite rules for dealing with the equality is denoted by $\mathcal{R}_+$. In the case of unrestricted narrowing, $\mathcal{R}_+$ denotes $\mathcal{R} \cup \{x = x \rightarrow true\}$, $x \in V$. This allows us to treat syntactical unification as

[1] The set Eq of equality axioms for a given program $\mathcal{R}$ are: *reflexivity* $(x = x \Leftarrow)$, *symmetry* $(x = y \Leftarrow y = x)$, *transitivity* $(x = z \Leftarrow x = y, y = z)$ and *f-substitutivity* $(f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \Leftarrow x_1 = y_1, \ldots, x_n = y_n$, for $f/n \in \Sigma)$.

a narrowing step, by using the rule $(x = x \to true)$ to compute mgu's. Then $s = t \overset{\sigma}{\rightsquigarrow} true$ holds iff $\sigma = mgu(\{s = t\})$.
We use the symbol $\top$ as a generic notation for sequences of the form $true, \ldots, true$. A *successful* derivation (or refutation) for g in $\mathcal{R}_+$ is a narrowing derivation $g \overset{\theta}{\rightsquigarrow}{}^* \top$, where $\theta_{\lceil Var(g)}$ is the *computed answer substitution (cas)*.

The *narrowing* mechanism is a powerful tool for constructing complete $\mathcal{E}$-unification algorithms for useful classes of equational theories. In this context, completeness means that, for every solution to a given set of equations, a more general solution can be found by narrowing. Formally, a narrowing algorithm is *complete* for (a class of) CTRS's if it generates a solution at least as general as any that satisfies the query (it generates a complete set of $\mathcal{E}$-unifiers).

2.2 Complete Narrowing Strategies

Since unrestricted narrowing has quite a large search space, several strategies to control the selection of redexes have been developed. A *narrowing strategy* (or *position constraint*) is any well-defined criterion which obtains a smaller search space by permitting narrowing to reduce only some chosen positions. A narrowing strategy φ can be formalized as a mapping that assigns a subset $\varphi(g)$ of $\overline{O}(g)$ to every goal g (different from $\top$) such that, for all $u \in \varphi(g)$, the goal g is narrowable at position u. An important property of a narrowing strategy φ is completeness, meaning that the narrowing constrained by φ is still complete. There is an inherited tradeoff coming from functional programming, between the benefits of outside-in evaluation of orthogonal, nonterminating rules and those of inner or eager evaluation with terminating, non orthogonal rules. A survey of results about the completeness of narrowing strategies can be found in [4, 21, 22, 29]. To simplify our notation, we let $I\!\!R_\varphi$ denote the class of CTRS's which satisfy the conditions for the completeness of the narrowing strategy φ.

We let $inn(g)$ (resp. $out(g)$) denote the narrowing strategy which assigns the position p of the leftmost-innermost (resp. leftmost-outermost) narrowing redex of g to the goal g. [2] We formulate a conditional narrower with strategy φ, $\varphi \in \{inn, out\}$, as the smallest relation $\rightsquigarrow_\varphi$ satisfying

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

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

$$c =_\varphi c \to true \qquad \% \; c/0 \in \mathcal{C}$$
$$c(x_1, \ldots, x_n) =_\varphi c(y_1, \ldots, y_n) \to (x_1 =_\varphi y_1) \wedge \ldots \wedge (x_n =_\varphi y_n) \qquad \% \; c/n \in \mathcal{C}$$

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

Here $=_\varphi$ is the standard equality $=$ of terms whenever $\varphi = inn$, whereas for the case when φ is out, and nonterminating rules are considered, we need to distinguish the standard (non-strict) equality $=$, which is defined on partially determined or infinite data structures from the strict equality $\approx$, which is only defined on finite and completely determined data structures, and which gives to equality the weak meaning of identity of finite objects (e.g., see [39]). We also assume that equations in g and C have the form $s = t$ whenever we consider $\varphi = inn$, whereas the equations have the form $s \approx t$ when we consider $\varphi = out$. Note that a non–strict equation like $f(a) = g(a)$ is not an acceptable goal when $\varphi = out$.

Innermost narrowing is the foundation of several functional logic programming languages like SLOG [26], LPG [7, 8] and (a subset of) ALF [28]. Innermost narrowing corresponds to the eager evaluation strategies in functional programming. Modern functional logic languages like Curry [31], Escher [35] and Toy [13] are based on lazy evaluation principles, which delay the evaluation of function arguments until their values are needed to compute some result. This avoids unnecessary computations and allows one to deal with infinite data structures [29]. Needed narrowing is a complete lazy narrowing strategy which is optimal w.r.t. the length of the derivations and the number of computed solutions in inductively sequential programs. Informally, inductive sequentiality amounts to the existence of discriminating left-hand sides, i.e. typical functional programs. Needed narrowing can be easily and efficiently implemented by translating definitional trees into case expressions as proposed in [32], which also proves that there is a strong equivalence of needed narrowing derivations in the original program and leftmost-outermost narrowing derivations in the transformed program. A similar transformation is presented in [44], where inductively sequential programs are translated to *uniform* form, which has only flat rules with pairwise non-subunifiable left-hand sides, where the strong equivalence between needed narrowing and leftmost-outermost narrowing derivations also holds.

3 Denotation of a Functional Logic Program

A Herbrand interpretation I for a program $\mathcal{R}$ is a set of ground equations, with the understanding that $s = t$ is true w.r.t. I iff $s = t \in I$. A Herbrand interpretation satisfies a program clause iff, for each ground instance $\lambda = \rho \Leftarrow C$ of the clause, we have that $\lambda = \rho \in I$ whenever $C \subseteq I$. A Herbrand E-interpretation for $\mathcal{R}$ is a Herbrand interpretation for $\mathcal{R}$ obeying the equality axioms for $\mathcal{R}$. A Herbrand model for $\mathcal{R}$ is a Herbrand interpretation for $\mathcal{R}$ which satisfies each program clause in $\mathcal{R}$. A Herbrand E-model for $\mathcal{R}$ is a Herbrand model for $\mathcal{R}$ which satisfies the equality axioms for $\mathcal{R}$. The intersection of all Herbrand E-models for $\mathcal{R}$ is also a Herbrand E-model for $\mathcal{R}$ (the least Herbrand E-model), and it was proposed as the declarative semantics for positive programs [33]. This semantics is known to be isomorphic to the initial algebra $\mathcal{H}/\mathcal{E}$ of the program, and in the following will be denoted by $\mathcal{M}(\mathcal{R})$.

For canonical programs, $\mathcal{M}(\mathcal{R})$ is equivalent to the operational semantics given by the ground success set, i.e. the set of all ground equations $s = t$ such that s and t have a common $\mathcal{R}$-normal form, and to the fixpoint semantics given by the least fixpoint $T_{\mathcal{R}}\!\uparrow\!\omega$ of the following transformation $T_{\mathcal{R}}$ (immediate consequence operator), which is continuous on the complete lattice of Herbrand interpretations ordered by set inclusion [33].

$$
\begin{aligned}
T_{\mathcal{R}}(I) = \{t = t \in \mathcal{B}\ \} \ \ \cup\ \{e \in \mathcal{B} \mid\ & (\lambda \to \rho \Leftarrow C) \in [\mathcal{R}], \\
& \{e[\rho]_u\} \cup C \subseteq I, u \in Occ(e), e_{|u} = \lambda\ \}.
\end{aligned}
$$

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

In order to formulate a semantics for functional logic programs modeling computed answers, the usual Herbrand base has to be extended to the set of all (possibly) non-ground equations modulo variance [23, 24]. $\mathcal{H}_V$ denotes the V-*Herbrand universe* which allows variables in its elements, and is defined as $\tau(\Sigma \cup V)/_{\approx}$. 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 $[\mathcal{B}_V]$. The preorder $\leq$ on $\tau(\Sigma \cup V)$ induces an order relation on $\tau(\Sigma \cup V)/_{\approx}$ (and therefore on $\mathcal{H}_V$). The ordering on $\mathcal{H}_V$ induces an ordering on $\mathcal{B}_V$, namely $s' = t' \leq s = t$ if $s' \leq s$ and $t' \leq t$. The power set of $\mathcal{B}_V$ is a complete lattice under set inclusion.

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

Definition 2 (Flat goal w.r.t. φ). *A* flat *equation is an equation of the form $f(d_1, \ldots, d_n) = d$ or $d_1 =_{\varphi} d_2$, where $d, d_1, \ldots, d_n \in \tau(\mathcal{C} \cup V)$ are constructor terms. A* flat *goal is a set of flat equations.*

For the outermost strategy, $\varphi = out$, the only non–strict equations in a flat goal are of the form $f(\tilde{d}) = x$. These equalities are treated differently from those originally present in the bodies of program rules and in the goal (denoted by $\approx$). In particular, the clauses for $=$ must allow the elimination of $f(a) = x$, whenever $f(a)$ would not have been selected by narrowing (i.e., when its value is not required to reduce $g(f(a))$.

Any sequence of equations E can be transformed into an equivalent one, $flat_{\varphi}(E)$, which is flat. The *flattening* procedures for equation sets which produce flat goals w.r.t. *inn* and *out*, respectively, can be found in [10, 27].

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

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

In non–strict languages, if the compositional character of meaning has to be preserved in presence of infinite data structures and partial functions, then non-normalizable terms, which may occur as subterms within normalizable expressions, also have to be assigned a denotation. Such a denotation is bound to the class of all partial results of the infinite computation along with the usual approximation ordering on them [27, 39] or, equivalently, the infinite data structure defined as the least upper bound of this class. Following [27, 39], we introduce a fresh constant symbol $\perp$ into Σ to represent the value of expressions which would otherwise be undefined.

Definition 3. *Let $\mathcal{I}$ be a Herbrand interpretation and $\mathcal{R} \in I\!\!R_{\varphi}$. Then,*

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

where $\mathcal{R}^{\varphi} = \mathcal{R}$ if $\varphi = inn$, whereas $\mathcal{R}^{\varphi} = \mathcal{R} \cup \{f(\tilde{t}) \to \perp \mid f/n \in \mathcal{D}\}$ if $\varphi = out$.

The following proposition allows us to define the fixpoint semantics.

Proposition 1. *The $T^{\varphi}_{\mathcal{R}}$ operator is continuous on the complete lattice of Herbrand interpretations, $\varphi \in \{inn, out\}$. The least fixpoint $lfp(T^{\varphi}_{\mathcal{R}}) = T^{\varphi}_{\mathcal{R}} \uparrow \omega$.*

Definition 4. *The least fixpoint semantics of a program $\mathcal{R}$ in $I\!\!R_{\varphi}$ is defined as $\mathcal{F}_{\varphi}(\mathcal{R}) = lfp(T^{\varphi}_{\mathcal{R}})$, $\varphi \in \{inn, out\}$.*

Definition 5. *We let $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ denote $\{l = r \in lfp(T^{\varphi}_{\mathcal{R}}) \mid r$ does not contain any defined function symbol $f/n \in \mathcal{D}\}$, $\varphi \in \{inn, out\}$.*

Theorem 1. (strong soundness and strong completeness)
Let $\mathcal{R}$ be a program in $I\!\!R_{\varphi}$ and g be a (non–trivial) goal according to φ. Then θ is an answer substitution computed by $\leadsto_{\varphi}$ for g in $\mathcal{R}$ iff there exist $g' \equiv (e_1,\ldots,e_n) \ll \mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ such that $\theta = mgu(flat_{\varphi}(g), g')_{\restriction Var(g)}$.

Example 1. Let us consider the program $\mathcal{R} = \{\mathtt{g(x)} \to \mathtt{0},\ \mathtt{f(0)} \to \mathtt{0},\ \mathtt{f(s(x))} \to \mathtt{f(x)}\}$. According to Definition 4, $\mathcal{F}^{ca}_{inn}(\mathcal{R}) = \{\mathtt{0} = \mathtt{0},\ \mathtt{s(x)} = \mathtt{s(x)},\ \mathtt{g(x)} = \mathtt{0},\ \mathtt{f(0)} = \mathtt{0},\ \mathtt{f(s(0))} = \mathtt{0}, \ldots, \mathtt{f(s^n(0))} = \mathtt{0}, \ldots\}$. Given the goal $g \equiv (\mathtt{y} = \mathtt{f(z)})$, innermost narrowing computes the answers $\{\ \{\mathtt{y/0}, \mathtt{z/0}\},\ \{\mathtt{y/0}, \mathtt{z/s(0)}\}, \ldots, \{\mathtt{y/0}, \mathtt{z/s^n(0)}\}, \ldots\}$ in $\mathcal{R}$, which exactly coincides with the set of substitutions computed by unifying the flat goal $\mathtt{f(z)} = \mathtt{y}$ with the equations in $\mathcal{F}^{ca}_{inn}(\mathcal{R})$.

Example 2. Now consider the program $\mathcal{R} = \{\texttt{from(x)} \rightarrow [\texttt{x}|\texttt{from(s(x))}], \texttt{first}([\texttt{x}|\texttt{y}]) \rightarrow \texttt{x}\}$. According to Definition 4, $\mathcal{F}^{ca}_{out}(\mathcal{R}) = \{\texttt{s(x)} \approx \texttt{s(x)}, \texttt{from(x)} = \perp, \texttt{from(x)} = [\texttt{x}|\perp], \ldots, \texttt{from(x)} = [\texttt{x}|[\texttt{s(x)}|\ldots[\texttt{s}^\texttt{n}\texttt{(x)}|\perp]]], \ldots, \texttt{first(x)} = \perp, \texttt{first}([\texttt{x}|\texttt{y}]) = \texttt{x}\}$, with $n \in \omega$. Given the goal $g \equiv (\texttt{first(from(s(x)))} \approx \texttt{z})$, outermost narrowing only computes the answer $\{\texttt{z/s(x)}\}$ in $\mathcal{R}$, which is also the only substitution which can be computed by unifying the flat goal $(\texttt{first(y)} = \texttt{w}, \texttt{from(s(x))} = \texttt{y}, \texttt{w} \approx \texttt{z})$ in $\mathcal{F}^{ca}_{out}(\mathcal{R})$.

According to Theorem 1, $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ can be used to simulate the execution for any (non–trivial) goal g, that is, $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ can be viewed as a (possibly infinite) set of 'unit' clauses, and the computed answer substitutions for g in $\mathcal{R}$ can be determined by 'executing' $flat(g)$ in the program $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ by standard unification, as if the equality symbol were an ordinary predicate. In the following, we show the relation between the semantics $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ and a novel operational "computed answer" semantics $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ which correctly models the behavior of single equations, which we introduce in the following.

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

Definition 6. *Let $\mathcal{R}$ be a program in $I\!R_{\varphi}$. Then,*

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

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

Theorem 2. *If $\mathcal{R} \in I\!R_{inn}$, then $\mathcal{O}^{ca}_{inn}(\mathcal{R}) = \mathcal{F}^{ca}_{inn}(\mathcal{R})$.*
If $\mathcal{R} \in I\!R_{out}$, then $\mathcal{O}^{ca}_{out}(\mathcal{R}) = \{l = r \in \mathcal{F}^{ca}_{out}(\mathcal{R}) \mid \perp$ does not occur in $r\}$

The following theorem relates the non-ground semantics $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ to the standard least Herbrand E-model semantics $\mathcal{M}(\mathcal{R})$.

Theorem 3. *Let $\mathcal{R}$ be a program in $I\!R_{\varphi}$ and $r^{\mp}$ be the transitive-symmetric closure of relation r under replacement (the f-*substitutivity *property). Then, $\mathcal{M}(\mathcal{R}) = [\mathcal{O}^{ca}_{\varphi}(\mathcal{R})]^{\mp}$.*

4 Diagnosis of declarative programs

We now introduce some basic definitions on the diagnosis of declarative programs [17]. As operational semantics we consider the set of computed answer substitutions.

Definition 7. *Let $\mathcal{I}$ be the specification of the intended computed answer semantics for $\mathcal{R}$.*

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

If a program contains errors, these are signalled by corresponding *symptoms*.

Definition 8. *Let $\mathcal{I}$ be the specification of the intended computed answer semantics for $\mathcal{R}$. An* incorrectness symptom *is an equation e such that $e \in \mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ and $e \notin \mathcal{I}$. An* incompleteness symptom *is an equation e such that $e \in \mathcal{I}$ and $e \notin \mathcal{O}^{ca}_{\varphi}(\mathcal{R})$.*

In case of errors, in order to determine the faulty rules, we give the following definitions.

Definition 9. *Let $\mathcal{I}$ be the specification of the intended fixpoint semantics for $\mathcal{R}$. If there exists an equation $e \in T^{\varphi}_{\{r\}}(\mathcal{I})$ and $e \notin \mathcal{I}$, 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 $\mathcal{I}$.

Definition 10. *Let $\mathcal{I}$ be the specification of the intended fixpoint semantics for $\mathcal{R}$. An equation e is uncovered if $e \in \mathcal{I}$ and $e \notin T^{\varphi}_{\mathcal{R}}(\mathcal{I})$.*

By the above definition, an equation e is uncovered if it cannot be derived by any program rule using the intended fixpoint semantics.

Proposition 2. *If there are no incorrect rules in $\mathcal{R}$ w.r.t the specification of the intended fixpoint semantics, then $\mathcal{R}$ is partially correct w.r.t. the intended computed answer semantics.*

We now consider a bottom-up abstract debugger for a strict language. Hence, in the rest of this paper, we fix $\varphi = inn$.

5 Abstract success set

The theory of abstract interpretation [18] provides a formal framework for developing advanced data-flow analysis tools. Abstract interpretation formalizes the idea of 'approximate computation' in which computation is performed with descriptions of data rather than with the data itself. The semantics operators are then replaced by abstract operators which are shown to 'safely' approximate the standard ones. In this section, starting from the fixpoint semantics in Section 3, we develop an abstract semantics which approximates the observable behavior of the program and is adequate for modular data-flow analysis, such as the analysis of unsatisfiability of equation sets or any analysis which is based on the program success set. We assume the framework of abstract interpretation for analysis of equational unsatisfiability as defined in [2]. Another approach to constructing an abstract term rewriting system is followed in [9]. We think that another approximation of the fixpoint semantics given in the previous section which is different from the one that we describe in this section can be characterized by following an approach similar to that in [9]. We first recall the abstract domains and the associated abstract operators. Then we describe a novel abstract immediate consequence operator $T^{\sharp}_{\mathcal{R}}$ which is able to approximate the operator $T_{\mathcal{R}}$, and the abstract fixpoint semantics $\mathcal{F}^{\sharp}(\mathcal{R})$. In the following, we denote the abstract analog of a concrete object O by $O^{\sharp}$.

5.1 Abstract Domains and Operators

A *description* is the association of an *abstract domain* $(D, \leq)$ (a poset) with a *concrete domain* $(E, \leq)$ (a poset). When $E = Eqn$ or $E = Sub$, the description is called an *equation description* or a *substitution description*, respectively. The correspondence between the abstract and concrete domain is established through a 'concretization' function $\gamma : D \rightarrow 2^E$. We say that d *approximates* e, written $d \propto e$, iff $e \in \gamma(d)$. The approximation relation can be lifted to relations and cross products as usual [2].

Abstract substitutions are introduced for the purpose of describing the computed answer substitutions for a given goal. Abstract equations and abstract substitutions correspond, in our approach, to abstract program denotations and abstract observable properties, respectively. The domains for equations and substitutions are as follows.

Definition 11 (abstract Herbrand universe). *Let $\sharp$ be an irreducible symbol, where $\sharp \notin \Sigma$. Let $\mathcal{H}_V^\sharp = (\tau(\Sigma \cup V \cup \{\sharp\}), \preceq)$ be the domain of terms over the signature augmented by $\sharp$, where the partial order $\preceq$ is defined as follows:*

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

(b) $\forall s_1, \ldots, s_n, s'_1, \ldots, s'_n \in \mathcal{H}_V^\sharp, \forall f/n \in \Sigma.\ s'_1 \preceq s_1 \wedge \ldots \wedge s'_n \preceq s_n \Rightarrow f(s'_1, \ldots, s'_n) \preceq f(s_1, \ldots, s_n)$ This order can be extended to equations: $s' = t' \preceq s = t$ iff $s' \preceq s$ and $t' \preceq t$ and to sets of equations S, S':

1) $S' \preceq S$ iff $\forall e' \in S'. \exists e \in S$ such that $e' \preceq e$. Note that $S' \preceq \{true\} \Rightarrow S' \equiv \{true\}$.

2) $S' \sqsubseteq S$ iff $(S' \preceq S)$ and $(S \preceq S'$ implies $S' \subseteq S)$.

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

Roughly speaking, the special symbol $\sharp$ introduced in the abstract domains represents any concrete term. The behaviour of the symbol $\sharp$ from a programming viewpoint resembles that of an "anonymous" variable in Prolog. From the viewpoint of logic, $\sharp$ stands for an existentially quantified variable [2, 37, 38]. Define $[\![S]\!] = S'$, where the n-tuple of occurrences of $\sharp$ in S is replaced by an n-tuple of existentially quantified fresh variables in S'.

Definition 12. *An abstract substitution is a set of the form $\{x_1/t_1, \ldots, x_n/t_n\}$ where, for each $i = 1, \ldots, n$, x_i is a distinct variable in V not occurring in any of the terms $t_1, \ldots, t_n$ and $t_i \in \tau(\Sigma \cup V \cup \{\sharp\})$. The ordering on abstract substitutions is given by logical implication: let $\theta, \kappa \in Sub^\sharp$, $\kappa \preceq \theta$ iff $[\![\widehat{\theta}]\!] \Rightarrow [\![\widehat{\kappa}]\!]$.*

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

Definition 13. *Let $\mathcal{H}_V = (\tau(\Sigma \cup V), \leq)$ and $\mathcal{H}_V^\sharp = (\tau(\Sigma \cup V \cup \{\sharp\}), \preceq)$. The* term description *is $\langle \mathcal{H}_V^\sharp, \gamma, \mathcal{H}_V \rangle$ where $\gamma : \mathcal{H}_V^\sharp \rightarrow 2^{\mathcal{H}_V}$ is defined by: $\gamma(t') = \{t \in \mathcal{H}_V | t' \preceq t\}$.*

Let Eqn be the set of finite sets of equations over $\tau(\Sigma \cup V)$ and $Eqn^\sharp$ be the set of finite sets of equations over $\tau(\Sigma \cup V \cup \{\sharp\})$. The equation description *is $\langle (Eqn^\sharp, \sqsubseteq), \gamma, (Eqn, \leq) \rangle$, where $\gamma : Eqn^\sharp \rightarrow 2^{Eqn}$ is defined by: $\gamma(g') = \{g \in Eqn | g' \sqsubseteq g$ and g is unquantified $\}$.*

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

In order to perform computations over the abstract domains, we have to define the notion of *abstract unification.* The abstract most general unifier for our method is very simple and, roughly speaking, it boils down to computing a solved form of an equation set with (possibly) existentially quantified variables. We define the abstract most general unifier for an equation set $S' \in Eqn^\sharp$ as follows. First replace all occurrences of $\sharp$ in S' by existentially quantified fresh variables. Then take a solved form of the resulting quantified equation set and finally replace the existentially quantified variables again by $\sharp$. Formally: let $\exists y_1 \ldots y_n.S = solve([\![S']\!])$ and $\kappa = \{y_1/\sharp, \ldots, y_n/\sharp\}$. Then $\widehat{mgu}^\sharp(S') = S\kappa$. The fact that $\forall \theta \in unif([\![S]\!]).\ mgu^\sharp(S) \preceq \theta$ justifies our use of 'most general'. The safety of the abstract unification algorithm has been proven in [2].

Our analysis is based on a form of simplified (abstract) program which always terminates and in which the query can be executed efficiently. Our notion of abstract program is parametric with respect to a loop-check. Two different instances can be found in [2, 3].

Definition 14. *A loop-check is a graph $\mathcal{G}_\mathcal{R}$ associated with a program $\mathcal{R}$, i.e. a relation consisting of a set of pairs of terms, such that: (1) the transitive closure $\mathcal{G}^+_\mathcal{R}$ is decidable and (2) Let $\mathring{t} = t'$ be a function which assigns to a term t some node t' in $\mathcal{G}_\mathcal{R}$. If there is an infinite sequence:*

$\langle \Leftarrow G_0, \theta_0\rangle \rightsquigarrow \langle \Leftarrow G_1, \theta_1\rangle \rightsquigarrow \ldots$

then $\exists i \geq 0.\ \langle \mathring{t}_i, \mathring{t}_i\rangle \in \mathcal{G}^+_\mathcal{R}$, where $t_i = e_{|u}\theta_i$, $e \in G_i$ and $u \in \bar{O}(e)$. (we refer to $\langle \mathring{t}_i, \mathring{t}_i\rangle$ as a 'cycle' of $\mathcal{G}_\mathcal{R}$.)

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

Definition 15 (abstract rule). *Let $\mathcal{R}$ be a program and let $r = (\lambda \to \rho \Leftarrow C) \in \mathcal{R}$. Let $\mathcal{G}_\mathcal{R}$ be a loop-check for $\mathcal{R}$. We define the abstraction of r as follows: $r^\sharp = (\lambda \to sh(\rho, \mathcal{G}_\mathcal{R}) \Leftarrow sh(C, \mathcal{G}_\mathcal{R}))$ where the shell $sh(x, \mathcal{G})$ of an expression x according to a loop-check $\mathcal{G}$ is defined inductively*

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

We can now formalize the abstract semantics.

5.2 Bottom-up Abstract Semantics

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

Definition 16. (abstract Herbrand base, abstract Herbrand interpretation) *The abstract Herbrand base of equations $\mathcal{B}^\sharp_V$ is defined as the set of equations over the abstract Herbrand universe $\mathcal{H}^\sharp_V$. An abstract Herbrand interpretation is any element of $2^{\mathcal{B}^\sharp_V}$. A partial order $\subseteq^\sharp$ on abstract interpretations can be defined in a way similar to the order $\subseteq$ on interpretations.*

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

Definition 17. *Let $\mathcal{R}$ be a program, $\mathcal{G}_{\mathcal{R}}$ be a loop-check for $\mathcal{R}$ and $\mathcal{R}^\sharp$ be the abstraction of $\mathcal{R}$ using $\mathcal{G}_{\mathcal{R}}$ where we also drop any abstract trivial equation from the body of the rules if there exist. Let $\mathcal{I}$ be an abstract interpretation. Then,*

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

Proposition 3. *The $T^\sharp_{\mathcal{R}}$ operator is continuous on the complete lattice of abstract interpretations.*

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

Definition 18. (abstract least fixpoint semantics) *Let $\mathcal{F}^\sharp(\mathcal{R}) = lfp(T^\sharp_{\mathcal{R}})$. The abstract least fixpoint semantics of a program $\mathcal{R}$ is $\mathcal{F}^{ca\sharp}(\mathcal{R}) = \{l = r \in \mathcal{F}^\sharp(\mathcal{R}) \mid r$ does not contain any defined function symbol $f/n \in \mathcal{D}\}$*

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

Theorem 4. *There exists a finite positive number k such that $\mathcal{F}^\sharp(\mathcal{R}) = T^\sharp_{\mathcal{R}} \uparrow k$.*

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

Theorem 5. *There exists a finite positive number k such that $T^\sharp_{\mathcal{R}} \uparrow k \propto T_{\mathcal{R}} \uparrow \omega$.*

Corollary 1. *$\mathcal{F}^\sharp(\mathcal{R}) \propto \mathcal{F}(\mathcal{R})$ and $\mathcal{F}^{ca\sharp}(\mathcal{R}) \propto \mathcal{F}^{ca}(\mathcal{R})$.*

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

Theorem 6 (strong completeness). *Let $\mathcal{R}$ be a program and $\Leftarrow g$ be a goal. If θ is a computed answer substitution for $\Leftarrow g$ in $\mathcal{R}$, then there exists $g' \equiv e_1, \ldots, e_m \ll \mathcal{F}^{ca\sharp}(\mathcal{R})$ such that $\theta' = mgu^\sharp(flat(g), g')$ and $\theta' \preceq \theta$.*

Example 3. Let us consider the program $\mathcal{R} = \{\ \mathtt{g(0)} \to \mathtt{0},\ \mathtt{g(c(x))} \to \mathtt{c(g(x))},\ \mathtt{f(0)} \to \mathtt{0},\ \mathtt{f(c(x))} \to \mathtt{x} \Leftarrow \mathtt{g(c(x))} = \mathtt{c(x)}\}$. Let us consider the loop-check [3] $\mathcal{G}_\mathcal{R} = \{\langle \mathtt{g(x)}, \mathtt{g(x)}\rangle\}$, and let $t \overset{\circ}{=} t'$ be the (partial) function which, given a graph, assigns to a term t some node t' in the graph such that t' unifies with t, if one such node t' exists. Then, the abstraction of the program $\mathcal{R}$ is $\mathcal{R}^\sharp = \{\ \mathtt{g(0)} \to \mathtt{0},\ \mathtt{g(c(x))} \to \mathtt{c}(\sharp),\ \mathtt{f(0)} \to \mathtt{0},\ \mathtt{f(c(x))} \to \mathtt{x} \Leftarrow \mathtt{g(c(x))} = \mathtt{c(x)}\}$.
$\mathcal{F}(\mathcal{R}) = \{\ \mathtt{0} = \mathtt{0},\ \mathtt{g(x)} = \mathtt{g(x)},\ \mathtt{f(x)} = \mathtt{f(x)},\ \mathtt{c(x)} = \mathtt{c(x)},\ \mathtt{g(0)} = \mathtt{0},\ \mathtt{f(0)} = \mathtt{0},\ \mathtt{g(c(x))} = \mathtt{c(g(x))}, \ldots,\ \mathtt{g(c^n(x))} = \mathtt{c^n(g(x))}, \ldots,\ \mathtt{g(c(0))} = \mathtt{c(0)}, \ldots,\ \mathtt{g(c^n(0))} = \mathtt{c^n(0)}, \ldots, \mathtt{f(c(0))} = \mathtt{0}, \ldots,\ \mathtt{f(c^n(0))} = \mathtt{c^{n-1}(0)}, \ldots\}$ and the corresponding fixpoint abstract semantics is the finite set
$\mathcal{F}^\sharp(\mathcal{R}) = \{\ \mathtt{0} = \mathtt{0},\ \mathtt{g(x)} = \mathtt{g(x)},\ \mathtt{f(x)} = \mathtt{f(x)},\ \mathtt{c(x)} = \mathtt{c(x)},\ \mathtt{g(0)} = \mathtt{0},\ \mathtt{f(0)} = \mathtt{0},\ \mathtt{g(c(x))} = \mathtt{c}(\sharp),\ \mathtt{f(c(x))} = \mathtt{x}\}$, and $\mathcal{F}^{ca\sharp}(\mathcal{R}) = \{\ \mathtt{0} = \mathtt{0},\ \mathtt{c(x)} = \mathtt{c(x)},\ \mathtt{g(0)} = \mathtt{0},\ \mathtt{f(0)} = \mathtt{0},\ \mathtt{g(c(x))} = \mathtt{c}(\sharp),\ \mathtt{f(c(x))} = \mathtt{x}\}$ which approximates the program success set. Given a goal $\Leftarrow g \equiv\ \Leftarrow \mathtt{f(g(x))} = \mathtt{y}$, innermost conditional narrowing computes the infinite set of substitutions $\{\{x/\mathtt{0},\ \mathtt{y/0}\}, \{\mathtt{x/c(0)},\ \mathtt{y/0}\}, \{\mathtt{x/c^2(0)},\ \mathtt{y/c(0)}\}, \ldots, \{\mathtt{x/c^n(0)},\ \mathtt{y/c^{n-1}(0)}\}\}$. The abstract substitutions returned by abstract unification of the equations in the flattened goal $\Leftarrow \mathtt{f(z)} = \mathtt{y},\ \mathtt{g(x)} = \mathtt{z}$ with $\mathcal{F}^{ca\sharp}(\mathcal{R})$ are $\{\{x/\mathtt{0},\ \mathtt{y/0}\}, \{\mathtt{x/c(x')}, \mathtt{y}/\sharp\}\}$, which approximate the computed answers of $\Leftarrow g$.

6 Abstract diagnosis

An efficient debugger can be based on the notion of over-approximation and under-approximation for the intended fixpoint semantics that we have introduced. The basic idea is to consider two sets to verify partial correctness: $\mathcal{I}^+$ which overapproximates the intended semantics $\mathcal{I}$ (that is, $\mathcal{I} \subseteq \gamma(\mathcal{I}^+)$) and $\mathcal{I}^-$ which underapproximates $\mathcal{I}$ (that is, $\mathcal{I} \supseteq \gamma(\mathcal{I}^-)$). We can then use such sets as shown in Table 1.

Semantic Property	Requirement
$\mathcal{R}$ is partially correct w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$	$\nexists e. e \in T_\mathcal{R}(\mathcal{I}^-)$ and $e \notin \mathcal{I}^+$
$\mathcal{R}$ is complete w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$	$\nexists e. e \in \mathcal{I}^-$ and $e \notin T_\mathcal{R}(\mathcal{I}^+)$
$\mathcal{R}$ is incorrect w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$	$\exists e. e \in T_\mathcal{R}(\mathcal{I}^-)$ and $e \notin \mathcal{I}^+$
$\mathcal{R}$ is incomplete w.r.t. $(\mathcal{I}^+, \mathcal{I}^-)$	$\exists e. e \in \mathcal{I}^-$ and $e \notin T_\mathcal{R}(\mathcal{I}^+)$

Table 1. Sufficient Conditions for Correctness and Completeness

Proposition 4. *If there exists an equation e such that $e \notin \mathcal{I}^+$ and $e \in T_{\{r\}}(\mathcal{I}^-)$, then the rule $r \in \mathcal{R}$ is incorrect on e.*

Proposition 5. *If there exists an equation e such that $e \notin T_{\mathcal{R}}(\mathcal{I}^+)$ and $e \in \mathcal{I}^-$, then the equation e is uncovered.*

In the following, by abuse we let $\mathcal{I}$ denote the program that specifies the intended semantics. In the following, we consider $\mathcal{I}^+ = lfp(T^\sharp_{\mathcal{I}})$, i.e. we consider the abstract success set that we have defined in previous section as overapproximation of the success set of a program. We can consider any of the sets defined in the works of [11, 16] as underapproximation of $\mathcal{I}$. Alternatively, we can simply take the set which results from a finite number of iterations of the $T_{\mathcal{I}}$ function (the concrete operator). Let us illustrate this method by an example.

Example 4. Let us consider a program $\mathcal{R} = \{\ \mathtt{g(0)} \to \mathtt{0},\ \mathtt{f(0)} \to \mathtt{0},\ \mathtt{g(c(X))} \to \mathtt{g(X)},\ \mathtt{f(c(X))} \to \mathtt{X} \Leftarrow\ \mathtt{g(c(X))} = \mathtt{c(X)}\}$ which is incorrect when we consider as intended specification the following program
$\mathcal{I} = \{\ \mathtt{g(0)} \to \mathtt{0},\ \mathtt{f(0)} \to \mathtt{0},\ \mathtt{g(c(x))} \to \mathtt{c(g(x))},\ \mathtt{f(c(x))} \to \mathtt{g(x)}\}$.
Then, by using the loop check in Example 3 we have
$\mathcal{I}^\sharp = \{\ \mathtt{g(0)} \to \mathtt{0},\ \mathtt{f(0)} \to \mathtt{0},\ \mathtt{g(c(x))} \to \mathtt{c(\sharp)},\ \mathtt{f(c(x))} \to \mathtt{g(x)}\}$.
After three iterations of the $T_{\mathcal{I}}$ operator, we get:
$\mathcal{I}^- = \{\ \mathtt{0} = \mathtt{0},\ \mathtt{c(x)} = \mathtt{c(x)},\ \mathtt{f(x)} = \mathtt{f(x)},\ \mathtt{g(x)} = \mathtt{g(x)},\ \mathtt{f(0)} = \mathtt{0},\ \mathtt{g(0)} = \mathtt{0},\ \mathtt{f(c(x))} = \mathtt{g(x)},\ \mathtt{g(c(x))} = \mathtt{c(g(x))},\ \mathtt{f(c(0))} = \mathtt{0},\ \mathtt{g(c(0))} = \mathtt{c(0)},\ \mathtt{f(c^2(x))} = \mathtt{c(g(x))},\ \mathtt{g(c^2(x))} = \mathtt{c^2(g(x))},\ \mathtt{f(c^2(0))} = \mathtt{c(0)}, \mathtt{f(c^3(x))} = \mathtt{c^2(g(x))},\ \mathtt{g(c^2(0))} = \mathtt{c^2(0)},\ \mathtt{g(c^3(x))} = \mathtt{c^3(g(x))}\}$.
After two iterations of the $T^\sharp_{\mathcal{I}}$ operator, we get the fixpoint:
$\mathcal{I}^+ = \mathcal{F}^\sharp(\mathcal{I}) = lfp(\mathcal{T}^\sharp_{\mathcal{I}}) = \{\ \mathtt{0} = \mathtt{0},\ \mathtt{c(x)} = \mathtt{c(x)},\ \mathtt{f(x)} = \mathtt{f(x)},\ \mathtt{g(x)} = \mathtt{g(x)},\ \mathtt{f(0)} = \mathtt{0},\ \mathtt{g(0)} = \mathtt{0},\ \mathtt{f(c(x))} = \mathtt{g(x)},\ \mathtt{g(c(x))} = \mathtt{c(\sharp)},\ \mathtt{f(c(0))} = \mathtt{0},\ \mathtt{f(c^2(x))} = \mathtt{c(\sharp)}\}$. And
$T_{\mathcal{R}}(\mathcal{I}^-) = \{\ \mathtt{0} = \mathtt{0},\ \mathtt{c(x)} = \mathtt{c(x)},\ \mathtt{f(x)} = \mathtt{f(x)},\ \mathtt{g(x)} = \mathtt{g(x)},\ \mathtt{f(0)} = \mathtt{0},\ \mathtt{g(0)} = \mathtt{0},\ \mathtt{g(c(x))} = \mathtt{g(x)},\ \mathtt{f(c(0))} = \mathtt{0},\ \mathtt{f(c^2(x))} = \mathtt{g(x)},\ \mathtt{g(c(0))} = \mathtt{c(0)},\ \mathtt{g(c^2(x))} = \mathtt{c(g(x))},\ \mathtt{f(c^2(0))} = \mathtt{c(0)},\ \mathtt{f(c^3(x))} = \mathtt{c(g(x))},\ \mathtt{g(c^2(0))} = \mathtt{c^2(0)},\ \mathtt{f(c^3(0))} = \mathtt{c^2(0)}, \mathtt{f(c^4(x))} = \mathtt{c^2(g(x))},\ \mathtt{g(c^3(x))} = \mathtt{c^2(g(x))},\ \mathtt{g(c^3(0))} = \mathtt{c^3(0)},\ \mathtt{g(c^4(x))} = \mathtt{c^3(g(x))}\}$. Now we can derive that $\mathtt{g(c(x))} = \mathtt{g(x)} \in \mathcal{T}_{\mathcal{R}}(\mathcal{I}^-)$, while $\mathtt{g(c(x))} = \mathtt{g(x)} \notin \mathcal{I}^+$. Hence, the corresponding rule $\mathtt{g(c(x))} \to \mathtt{g(x)}$ is wrong.

7 The System for Declarative Debugging BUGGY

The basic rules presented so far have been implemented by a prototype system BUGGY [1], which is available at

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

It includes a parser for a conditional functional logic language, whose semantics is based on innermost (basic) narrowing, which is a generalization of basic narrowing such that $\mathcal{R}$ does not need to be completely defined, which is quite a restrictive condition which is necessary for the completeness of innermost narrowing [10, 33]. The implementation also includes a module for computing an abstraction of the program based on a given loop-check, an automatic debugger

which requires that the user indicate some parameters, such as the number n of iterations for approximating the success set from the bottom. Then the errors are automatically found by the debugger and the user has to indicate the corrections to be made on the wrong rules. The BUGGY system is written in SICStus Prolog v3.8.1 and the complete implementation consists of about 300 clauses (1260 lines of code). The debugger is expressed by 147 clauses (including the user interface and the code needed to handle the representation), the parser and other utilities are expressed by 65 clauses and basic narrowing is implemented by 88 clauses. Language syntax follows mainly that of a generic conditional functional logic language, with conditional basic narrowing.
The BUGGY main screen allows the user to choose between several alternatives.

1. `File options`: It contains the classical options for opening, loading and saving a file, as well as cleaning and exiting the system.
2. `Edit`: it is possible to load, edit, and visualize the program to be debugged on the screen, as well as its intended semantics. The program is incrementally parsed while it is loaded.
3. `Debug`: the debugger starts debugging the input program w.r.t. the intended semantics. The user has to say how many iterations to apply for approximating from below the intended semantics. The errors are shown one by one to the user who is required to propose the corrections which are in turn tested. The final correct program is shown to the user who can save it.
4. `Help`: contains additional information about the system.

We have tested our debugging methodology over several benchmarks. We have considered programs such as `append` for computing the concatenation of two input lists, `last` which returns the last element of a list, `knapsack` which returns a set of elements of the input list whose weight sum is equal to an input integer value, `fibonacci` which computes the Fibonacci numbers, `fact` which computes the factorial of a positive numer, `sort` which uses the insertion sort for ordering an input list of integers. For all these programs by using the intended semantics we were able to find the errors which were inserted in the program. The final programs have passed the tests of correctness and completeness.

In the following we illustrate the power of our debugging system when functional logic programs are used as specification. The idea goes back to the origins of declarative programming, considering declarative specifications as programs for rapid prototyping [15]. In our case, we go one step further, because the specification can then be automatically abstracted and can be used to automatically debug the final efficient program.

Let us consider the following program, which should order a list of integers using the algorithm of insertion sort.

$$\begin{array}{rl}
\mathtt{sort}([\mathtt{X}]) \rightarrow []. & \\
\mathtt{sort}([\mathtt{X}|\mathtt{Xs}]) \rightarrow \mathtt{Ys} & \Leftarrow \mathtt{sort}(\mathtt{Xs}) = \mathtt{Zs}, \mathtt{insert}(\mathtt{X}, \mathtt{Zs}, \mathtt{Ys}) = \mathtt{true}. \\
\mathtt{insert}(\mathtt{X}, [], [\mathtt{X}]) \rightarrow \mathtt{true}. & \\
\mathtt{insert}(\mathtt{X}, [\mathtt{Y}|\mathtt{Ys}], [\mathtt{Y}|\mathtt{Zs}]) \rightarrow \mathtt{true} & \Leftarrow \mathtt{X} > \mathtt{Y} = \mathtt{true}, \mathtt{insert}(\mathtt{X}, \mathtt{Ys}, \mathtt{Zs}) = \mathtt{true}. \\
\mathtt{insert}(\mathtt{X}, [\mathtt{Y}|\mathtt{Ys}], [\mathtt{X}, \mathtt{Y}|\mathtt{Ys}]) \rightarrow \mathtt{true} & \Leftarrow \mathtt{X} =< \mathtt{Y} = \mathtt{true}.
\end{array}$$

The specification $\mathcal{I}$ is given by the (quite) inefficient program which uses the naive sorting algorithm which computes the permutations of the input list.

$$
\begin{array}{rll}
\mathtt{sort(Xs)} \to \mathtt{Ys} & \Leftarrow \mathtt{perm(Xs)} = \mathtt{Ys}, \mathtt{ord(Ys)} = \mathtt{true}. \\
\mathtt{ord([])} \to \mathtt{true}. & \\
\mathtt{ord([X])} \to \mathtt{true}. & \\
\mathtt{ord([X,Y|Xs])} \to \mathtt{true} & \Leftarrow \mathtt{X} =< \mathtt{Y} = \mathtt{true}, \mathtt{ord([Y|Xs])} = \mathtt{true}. \\
\mathtt{perm(Xs)} \to \mathtt{[Z|Zs]} & \Leftarrow \mathtt{select(Z,Xs,Ys)} = \mathtt{true}, \mathtt{perm(Ys)} = \mathtt{Zs}. \\
\mathtt{perm([])} \to \mathtt{[]}. & \\
\mathtt{select(X,[X|Xs],Xs)} \to \mathtt{true}. & \\
\mathtt{select(X,[Y|Xs],[Y|Zs])} \to \mathtt{true} & \Leftarrow \mathtt{select(X,Xs,Zs)} = \mathtt{true}.
\end{array}
$$

The overapproximation $\mathcal{I}^+$ is given by the following set of equations: $\{\mathtt{sort(A)} = \sharp,\ \mathtt{select(X,[Y|Z],[Y|\sharp])} = \mathtt{true},\ \mathtt{select(X,[X|Y],Y)} = \mathtt{true},$ $\mathtt{perm([])} = \mathtt{[]},\ \mathtt{perm(X)} = [\sharp|\sharp],\ \mathtt{ord([X])} = \mathtt{true},\ \mathtt{ord([])} = \mathtt{true},\ \mathtt{select(X,Y,Z)} =$ $\mathtt{select(X,Y,Z)},\ \mathtt{X} =< \mathtt{Y} = \mathtt{X} =< \mathtt{Y},\ \mathtt{[X|Y]} = \mathtt{[X|Y]},\ \mathtt{[]} = \mathtt{[]},\ \mathtt{ord(X)} = \mathtt{ord(X)},$ $\mathtt{true} = \mathtt{true},\ \mathtt{perm(X)} = \mathtt{perm(X)},\ \mathtt{sort(X)} = \mathtt{sort(X)}\}$

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

8 Conclusions

We have presented an approach to declarative debugging of functional logic programs w.r.t. the set of computed answers. We have defined a declarative debugging method which has similarities to others which have been proposed in the literature (such as [11, 16]), but which is original w.r.t. the definition and use of the semantic equations for making the error diagnosis and is useful for both eager as well as for lazy languages. We have presented a novel fixpoint semantics for functional logic programs, which is parametric w.r.t. the narrowing strategy. This semantics characterizes the set of computed answers in a bottom-up manner. Thus, it is a suitable basis for dataflow analyses based on abstract interpretation. We present one example of abstraction of this fixpoint semantics which yields an approximated finite (goal-independent) description of the success patterns of the program and which can then be used in combination with our debugging equations to obtain an efficient and terminating debugging system. In this paper, we have discussed the successful experiments which have been performed with a prototypical implementation of our debugging system which is publicly available. We believe that it is possible to extend our system in several ways; for instance, by integrating other dataflow analyses for approximating term rewriting systems [9]. Another extension can be done by studying the relation and integration with assertion based methods for declarative debugging [20].

References

1. 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/papers.html`.

2. M. Alpuente, M. Falaschi, and F. Manzo. Analyses of Unsatisfiability for Equational Logic Programming. *Journal of Logic Programming*, 22(3):221–252, 1995.
3. M. Alpuente, M. Falaschi, and G. Vidal. A Compositional Semantic Basis for the Analysis of Equational Horn Programs. *TCS*, 165(1):97–131, 1996.
4. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
5. P. Arenas and A. Gil. A debugging model for lazy functional languages. Technical Report DIA 94/6, Universidad Complutense de Madrid, April 1994.
6. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
7. D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *Proc. of First European Symp. on Programming, ESOP'86*, pages 119–132. LNCS 213, 1986.
8. 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.
9. D. Bert, R. Echahed, and B.M. Østvold. Abstract Rewriting. In *Proc. of Third Int'l Workshop on Static Analysis, WSA'93*, pages 178–192. LNCS 724, 1993.
10. P. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23, 1988.
11. F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M Hermenegildo, J. Maluszyński, and G. Puebla. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging-AADEBUG'97*, pages 155–170. U. of Linkoping Press, 1997.
12. R. Caballero-Roldán, F.J. López-Fraguas, and M. Rodríquez Artalejo. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In *Fifth International Symposium on Functional and Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2001. To appear.
13. R. Caballero-Roldán, F.J. López-Fraguas, and J. Sánchez-Hernández. User's manual for Toy. Technical Report SIP-5797, UCM, Madrid (Spain), 1997.
14. M. Codish, M. Falaschi, and K. Marriott. Suspension Analysis for Concurrent Logic Programs. In K. Furukawa, editor, *Proc. of Eighth Int'l Conf. on Logic Programming*, pages 331–345. The MIT Press, Cambridge, MA, 1991.
15. M. Comini, R. Gori, and G. Levi. Logic programs as specifications in the inductive verification of logic programs. In *Proc. of Appia-Gulp-Prode'00*, 2000.
16. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.
17. M. Comini, G. Levi, and G. Vitiello. Declarative Diagnosis Revisited. In *Proc. of ILPS'95*, pages 275–287. The MIT Press, 1995.
18. 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*, pages 238–252, 1977.
19. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.
20. W. Drabent, S. Nadjim-Tehrani, and J. Maluszynski. The use of assertions in algorithmic debugging. In *Proceedings of FGCS'88*, pages 573–581, 1988.
21. R. Echahed. On completeness of narrowing strategies. In *Proc. of CAAP'88*, pages 89–101. Springer LNCS 299, 1988.
22. R. Echahed. Uniform narrowing strategies. In *Proc. of ICALP'92*, pages 259–275. Springer LNCS 632, 1992.

23. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *TCS*, 69(3):289–318, 1989.
24. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 103(1):86–113, 1993.
25. G. Ferrand. Error Diagnosis in Logic Programming, and Adaptation of E.Y.Shapiro's Method. *Journal of Logic Programming*, 4(3):177–198, 1987.
26. L. Fribourg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. of Second IEEE Int'l Symp. on Logic Programming*, pages 172–185. IEEE, New York, 1985.
27. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *JCSS*, 42:363–377, 1991.
28. M. Hanus. Compiling Logic Programs with Equality. In *Proc. of 2nd Int'l Workshop on Programming Language Implementation and Logic Programming*, pages 387–401. Springer LNCS 456, 1990.
29. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
30. M. Hanus and B. Josephs. A debugging model for functional logic programs. In M. Bruynooghe and J. Penjam, editors, *Proc. of PLILP*, volume 714 of *Lecture Notes in Computer Science*, pages 28–43. Springer, 1993.
31. 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.
32. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. In *Proc. of RTA'96*, pages 138–152. Springer LNCS 1103, 1996.
33. S. Hölldobler. *Foundations of Equational Logic Programming*. Springer LNAI 353, 1989.
34. 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.
35. J. W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3, 1999.
36. J.W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Computer Science Department, University of Bristol, 1995.
37. M.J. Maher. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. In *Proc. of LICS*, pages 348–357. IEEE, 1988.
38. M.J. Maher. On Parameterized Substitutions. Technical Report RC 16042, IBM - T.J. Watson Research Center, Yorktown Heights, NY, 1990.
39. 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.
40. L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18(1):401–408, 1996.
41. Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), April 1997.
42. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. of 2nd IEEE Int'l Symp. on Logic Programming*, pages 138–151. IEEE, 1985.
43. E. Y. Shaphiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Massachusetts, 1982. ACM Distinguished Dissertation.
44. F. Zartmann. Denotational Abstract Interpretation of Functional Logic Programs. In *Proc. of SAS'97*, pages 141–159. Springer LNCS 1302, 1997.