

Abstract Diagnosis of Functional Programs^{*}

M. Alpuente¹, M. Comini², S. Escobar¹, M. Falaschi², and S. Lucas¹

¹ Departamento de Sistemas Informáticos y Computación-DSIC
Technical University of Valencia, Camino de Vera s/n, 46022 Valencia, Spain.

{alpuente,sescobar,slucas}@dsic.upv.es

² Dipartimento di Matematica e Informatica
University of Udine, Via delle Scienze 206, 33100 Udine, Italy.

{comini,falaschi}@dimi.uniud.it

Abstract. We present a generic scheme for the declarative debugging of functional programs modeled as term rewriting systems. We associate to our programs a semantics based on a (continuous) immediate consequence operator, $T_{\mathcal{R}}$, which models the (values/normal forms) semantics of \mathcal{R} . Then, we develop an effective debugging methodology which is based on abstract interpretation: by approximating the intended specification of the semantics of \mathcal{R} we derive a finitely terminating bottom-up diagnosis method, which can be used statically. Our debugging framework does not require the user to either provide error symptoms in advance or answer questions concerning program correctness. We have made available a prototypical implementation in Haskell and have tested it on some non trivial examples.

1 Introduction

Finding program bugs is a long-standing problem in software construction. This paper is motivated by the fact that the debugging support for functional languages in current systems is poor [31], and there are no general purpose, good semantics-based debugging tools available. Traditional debugging tools for functional programming languages consist of tracers which help to display the execution [23,30] but which do not enforce program correctness adequately as they do not provide means for finding bugs in the source code w.r.t. the intended program semantics. Declarative debugging of functional programs [22,21,28] is a semi-automatic debugging technique where the debugger tries to locate the node in an execution tree which is ultimately responsible for a visible bug symptom. This is done by asking the user, which assumes the role of the oracle. When debugging real code, the questions are often textually large and may be difficult to answer. Abstract diagnosis [9,10,11] is a declarative debugging framework which extends the methodology in [16,26], based on using the immediate consequence operator T_P , to identify bugs in logic programs, to diagnosis w.r.t. computed

^{*} Work partially supported by CICYT TIC2001-2705-C03-01, Acciones Integradas HI2000-0161, HA2001-0059, HU2001-0019, and Generalitat Valenciana GV01-424.

answers. The framework is goal independent and does not require the determination of symptoms in advance.

In this paper, we develop an abstract diagnosis method for functional programming which applies the ideas of [10] to debug a functional program w.r.t. the semantics of normal forms and (ground) constructor normal forms (or *values*). We use the formalism of term rewriting systems as it provides an adequate computational model for functional programming languages where functions are defined by means of patterns (e.g., Haskell, Hope or Miranda) [4,18,25]. We associate a (continuous) immediate consequence operator $T_{\mathcal{R}}$ to program \mathcal{R} which allows us to derive an input-output semantics for \mathcal{R} , as in the fixpoint finite/angelic relational semantics of [12]. Then, we formulate an efficient debugging methodology, based on abstract interpretation, which proceeds by approximating the $T_{\mathcal{R}}$ operator by means of a *depth(k)* cut [10]. We show that, given the intended specification \mathcal{I} of the semantics of a program \mathcal{R} , we can check the correctness of \mathcal{R} by a single step of the abstract immediate consequence operator $T_{\mathcal{R}}^k$ and, by a simple static test, we can determine all the rules which are wrong w.r.t. the considered abstract property.

The debugging of functional programs via specifications is an important topic in automated program development. For example, in QuickCheck [7], formal specifications are used to describe properties of Haskell programs (which are written as Haskell programs too) which are automatically tested on random input. This means that the program is run on a large amount of arguments which are randomly generated using the specification. A size bound is used to ensure finiteness of the test data generation. A *domain specific* language of *testable specifications* is imposed which is embedded in Haskell, and only properties which are expressible and observable within this language can be considered. The major limitation of Quickcheck is that there is no measurement of structural coverage of the function under test: there is no check, for instance, that every part of the code is exercised as it heavily depends on the distribution of test data.

The debugging methodology which we propose can be very useful for a functional programmer who wants to debug a program w.r.t. a preliminary version which was written with no efficiency concern. Actually, in software development a specification may be seen as the starting point for the subsequent program development, and as the criterion for judging the correctness of the final software product. For instance, the executability of OBJ³ specifications supports prototype-driven incremental development methods [17]. On the other hand, OBJ languages have been provided with equational proving facilities such as a Knuth-Bendix completion tool which, starting with a finite set of equations and a reduction order, attempts to find a finite canonical system for the considered theory by generating critical pairs and orienting them as necessary [8,13]. However, it might happen that the completion procedure fails because there is a critical pair which cannot be oriented. Thus, in many cases, the original code needs to be manipulated by hand, which may introduce incorrectness or incom-

³ By OBJ we refer to the family of OBJ-like equational languages, which includes OBJ3, CafeOBJ, and Maude.

pleteness errors in program rules. Therefore, a debugging tool which is able to locate bugs in the user’s program and provide validation of the user’s intention becomes also important in this context. In general it often happens that some parts of the software need to be improved during the software life cycle, for instance for getting a better performance. Then the old programs (or large parts of them) can be usefully (and automatically) used as a specification of the new ones.

The rest of the paper is organized as follows. Section 3 introduces a novel immediate consequence operator $T_{\mathcal{R}}$ for functional program \mathcal{R} . We then define a fixpoint semantics based on $T_{\mathcal{R}}$ which correctly models the values/normal forms semantics of \mathcal{R} . Section 4 provides an abstract semantics which correctly approximates the fixpoint semantics of \mathcal{R} . In Section 5, we present our method of abstract diagnosis. The diagnosis is based on the detection of *incorrect rules* and *uncovered equations*, which both have a bottom-up definition (in terms of one application of the “abstract immediate consequence operator” $T_{\mathcal{R}}^{\alpha}$ to the abstract specification). It is worth noting that no fixpoint computation is required, since the abstract semantics does not need to be computed. We have developed a prototypical implementation in Haskell (DEBUSSY) which we use for running all the examples we illustrate in this section. Section 6 concludes.

2 Preliminaries

Let us briefly recall some known results about rewrite systems [4,18]. For simplicity, definitions are given in the one-sorted case. The extension to many-sorted signatures is straightforward, see [24]. In the paper, syntactic equality of terms is represented by \equiv . Throughout this paper, \mathcal{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. $\mathcal{T}(\Sigma, \mathcal{V})$ and $\mathcal{T}(\Sigma)$ denote the non-ground word (or term) algebra and the word algebra built on $\Sigma \cup \mathcal{V}$ and Σ , respectively. $\mathcal{T}(\Sigma)$ is usually called the Herbrand universe (\mathcal{H}_{Σ}) 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 $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$, or *true*.

Terms are viewed as labelled trees in the usual way. Positions are represented by sequences of natural numbers denoting an access path in a term. Given $S \subseteq \Sigma \cup \mathcal{V}$, $O_S(t)$ denotes the set of positions of a term t which are rooted by symbols in S . $t|_u$ 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 . By $Var(s)$ we denote the set of variables occurring in the syntactic object s , while $[s]$ denotes the set of ground instances of s . A *fresh* variable is a variable that appears nowhere else.

A *substitution* is a mapping from the set of variables \mathcal{V} into the set of terms $\mathcal{T}(\Sigma, \mathcal{V})$. A substitution θ is more general than σ , denoted by $\theta \leq \sigma$, if $\sigma = \theta\gamma$ for some substitution γ . We write $\theta|_s$ to denote the restriction of the substitution θ to the set of variables in the syntactic object s . The *empty substitution* is denoted by ϵ . A *renaming* is a substitution ρ for which there exists the inverse

ρ^{-1} , such that $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$. An equation set E is unifiable, if there exists ϑ such that, for all $s = t$ in E , we have $s\vartheta \equiv t\vartheta$, and ϑ is called a *unifier* of E . We let $mgu(E)$ denote 'the' *most general unifier* of the equation set E [20].

A *term rewriting system* (TRS for short) is a pair (Σ, \mathcal{R}) , where \mathcal{R} is a finite set of reduction (or rewrite) rule schemes of the form $l \rightarrow r$, $l, r \in \mathcal{T}(\Sigma, \mathcal{V})$, $l \notin \mathcal{V}$ and $Var(r) \subseteq Var(l)$. We will often write just \mathcal{R} instead of (Σ, \mathcal{R}) . For TRS \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 TRS (Σ, \mathcal{R}) , we assume that the signature Σ is partitioned into two disjoint sets $\Sigma := \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{D} := \{f \mid f(t_1, \dots, t_n) \rightarrow r \in \mathcal{R}\}$ 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 $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are called constructor terms. A *pattern* is a term of the form $f(\bar{d})$ where $f/n \in \mathcal{D}$ and \bar{d} is a n -tuple of constructor terms. We say that a TRS is *constructor-based* (CB) if the left hand sides of \mathcal{R} are patterns.

A rewrite step is the application of a rewrite rule to an expression. A term s *rewrites* to a term t , $s \rightarrow_{\mathcal{R}} t$, if there exist $u \in O_{\Sigma}(s)$, $l \rightarrow r$, and substitution σ such that $s|_u \equiv l\sigma$ and $t \equiv s[r\sigma]_u$. 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 $s \rightarrow_{\mathcal{R}} t$. t is the normal form of s if $s \rightarrow_{\mathcal{R}}^* t$ and t is a normal form (in symbols $s \rightarrow_{\mathcal{R}}^! t$). A TRS \mathcal{R} is *noetherian* if there are no infinite sequences of the form $t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} t_3 \rightarrow_{\mathcal{R}} \dots$. A TRS \mathcal{R} is *confluent* if, whenever a term s reduces to two terms t_1 and t_2 , both t_1 and t_2 reduce to the same term. The program \mathcal{R} is said to be canonical if \mathcal{R} is noetherian and confluent [18].

In the following we consider functional languages and thus often will refer to the corresponding TRS of a program with the term program itself.

3 The semantic framework

The relational style of semantic description associates an input-output relation to a program where intermediate computation steps are ignored [12]. In this section, we consider the finite/angelic relational semantics of [12], given in fix-point style. In order to formulate our semantics for term rewriting systems, the usual Herbrand base is extended to the set of all (possibly) non-ground equations [14,15]. $\mathcal{H}_{\mathcal{V}}$ denotes the \mathcal{V} -*Herbrand universe* which allows variables in its elements, and is defined as $\mathcal{T}(\Sigma, \mathcal{V})/\cong$, where \cong is the equivalence relation induced by the preorder \leq of "relative generality" between terms, i.e. $s \leq t$ if there exists σ s.t. $t \equiv \sigma(s)$. For the sake of simplicity, the elements of $\mathcal{H}_{\mathcal{V}}$ (equivalence classes) have the same representation as the elements of $\mathcal{T}(\Sigma, \mathcal{V})$ and are also called terms. $\mathcal{B}_{\mathcal{V}}$ denotes the \mathcal{V} -*Herbrand base*, namely, the set of all equations $s = t$ modulo variance, where $s, t \in \mathcal{H}_{\mathcal{V}}$. A subset of $\mathcal{B}_{\mathcal{V}}$ is called a \mathcal{V} -Herbrand interpretation. We assume that the equations in the denotation are renamed apart. The ordering \leq for terms is extended to equations in the obvious way, i.e. $s = t \leq s' = t'$ iff there exists σ s.t. $\sigma(s) = \sigma(t) \equiv s' = t'$.

The concrete domain \mathbb{E} is the lattice of \mathcal{V} -Herbrand interpretations, i.e., the powerset of $\mathcal{B}_{\mathcal{V}}$ ordered by set inclusion.

In the sequel, a semantics for program \mathcal{R} is a \mathcal{V} -Herbrand interpretation. In term rewriting, the semantics which is usually considered is the set of normal forms of terms, $Sem_{nf}(\mathcal{R}) := \{s = t \mid s \rightarrow_{\mathcal{R}}^! t\}$. On the other hand, in functional programming, programmers are generally concerned with an abstraction of such semantics where only the values (ground constructor normal forms) that input expressions represent are considered, $Sem_{val}(\mathcal{R}) := Sem_{nf}(\mathcal{R}) \cap \mathcal{T}(\Sigma, \mathcal{V}) \times \mathcal{T}(\mathcal{C})$.

Since our framework does not depend on the particular target semantics, following [12], our definitions are parametric by the set of final/blocking state pairs B . Then if we are interested, for example, in the semantics of values we can take as final/blocking state pairs the set $val := \{t = t \mid t \in \mathcal{T}(\mathcal{C})\}$. Moreover if we are interested in the semantics of normal forms we can take B as $nf := \{t = t \mid t$ is a normal form for $\mathcal{R}\}$.

We can give a fixpoint characterization of the shown semantics by means of the following immediate consequence operator.

Definition 1. *Let \mathcal{I} be a Herbrand interpretation, B be a set of final/blocking state pairs and \mathcal{R} be a TRS. Then,*

$$T_{\mathcal{R},B}(\mathcal{I}) := B \cup \{s = t \mid r = t \in \mathcal{I}, s \rightarrow_{\mathcal{R}} r\}$$

The following proposition allows us to define the fixpoint semantics.

Proposition 1. *Let \mathcal{R} be a TRS and B be a set of final/blocking state pairs. The $T_{\mathcal{R},B}$ operator is continuous on \mathbb{E} .*

Definition 2. *The least fixpoint semantics of a program \mathcal{R} w.r.t. a set of final/blocking state pairs B , is defined as $\mathcal{F}_B(\mathcal{R}) = T_{\mathcal{R},B} \uparrow \omega$.*

The following result relates the (fixpoint) semantics computed by the $T_{\mathcal{R}}$ operator with the semantics val and nf .

Theorem 1 (soundness and completeness). *Let \mathcal{R} be a TRS. Then, $Sem_{nf}(\mathcal{R}) = \mathcal{F}_{nf}(\mathcal{R})$ and $Sem_{val}(\mathcal{R}) = \mathcal{F}_{val}(\mathcal{R})$.*

Example 1. Let us consider now the following (wrong) program \mathcal{R} expressed in OBJ for doubling.

```
obj ERRDOUBLE is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op double : Nat -> Nat .
  var X : Nat .
  eq double(0) = 0 .
  eq double(s(X)) = double(X) .
endo
```

The intended specification is given by the following OBJ program \mathcal{I} which uses addition for doubling:

```

obj ADDDOUBLE is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op double : Nat -> Nat .
  op add : Nat Nat -> Nat .
  vars X Y : Nat .
  eq add(0,X) = X .
  eq add(s(X),Y) = s(add(X,Y)) .
  eq double(X) = add(X,X) .
endo

```

According to Definition 2, the val fixpoint semantics of \mathcal{R} is⁴ (we omit the equations for the auxiliary function add):

$$\mathcal{F}_{\text{val}}(\mathcal{R}) = \{0=0, \text{double}(0)=0, \text{double}(s(0))=0, \text{double}(s^2(0))=0, \\ \text{double}(s^3(0))=0, \text{double}(s^4(0))=0, \dots, s(0)=s(0), \\ s(\text{double}(0))=s(0), s(\text{double}(s(0)))=s(0), \\ s(\text{double}(s^2(0)))=s(0), s(\text{double}(s^3(0)))=s(0), \\ s(\text{double}(s^4(0)))=s(0), \dots, s^2(0)=s^2(0), \\ s^2(\text{double}(0))=s^2(0), s^2(\text{double}(s(0)))=s^2(0), \\ s^2(\text{double}(s^2(0)))=s^2(0), s^2(\text{double}(s^3(0)))=s^2(0), \\ s^2(\text{double}(s^4(0)))=s^2(0), \dots \}$$

whereas the nf fixpoint semantics of \mathcal{I} is:

$$\mathcal{F}_{\text{nf}}(\mathcal{I}) = \{0=0, X=X, s(0)=s(0), s(X)=s(X), \text{double}(0)=0, \\ \text{double}(X)=\text{add}(X,X), s^2(0)=s^2(0), s^2(X)=s^2(X), \\ s(\text{double}(0))=s(0), s(\text{double}(X))=s(\text{add}(X,X)), \\ \text{double}(s(0))=s^2(0), \text{double}(s(X))=\text{add}(s(X),s(X)), \\ \text{double}^2(0)=0, \text{double}^2(X)=\text{add}(\text{add}(X),\text{add}(X)), s^3(0)=s^3(0), \\ s^3(X)=s^3(X), s^2(\text{double}(X))=s^2(\text{add}(X,X)), \\ s(\text{double}(s(0)))=s^3(0), s(\text{double}(s(X)))=s(\text{add}(s(X),s(X))), \\ s(\text{double}^2(0))=s(0), s(\text{double}^2(X))=s(\text{add}(\text{add}(X),\text{add}(X))), \\ \text{double}(s^2(0))=s^4(0), \text{double}(s^2(X))=s^2(\text{add}(X,s^2(X))), \dots \}$$

Now, we can “compute in the fixpoint semantics $\mathcal{F}_{\text{val}}(\mathcal{R})$ ” the denotation of the term $t \equiv \text{double}(s(0))$, which yields $s \equiv 0$, since the denotation $\mathcal{F}_{\text{val}}(\mathcal{R})$ contains the equation $t = s$; note that this value is erroneous w.r.t. the intended semantics of the double operation.

4 Abstract semantics

In this section, starting from the fixpoint semantics in Section 3, we develop an abstract semantics which approximates the observable behavior of the pro-

⁴ We use the notation $f^n(x)$ as a shorthand for $f(f(\dots f(x)))$, where f is applied n times.

gram and is adequate for modular data-flow analysis, such as the analysis of unsatisfiability of equation sets.

We will focus our attention now on a special class of abstract interpretations which are obtained from what we call a *term abstraction* $\tau : \mathcal{T}(\Sigma, \mathcal{V}) \rightarrow AT$.

We start by choosing as abstract domain $\mathbb{A} := \mathcal{P}(\{a = a' \mid a, a' \in AT\})$, ordered by a set ordering \sqsubseteq . We will call elements of \mathbb{A} abstract Herbrand interpretations. The concrete domain \mathbb{E} is the powerset of $\mathcal{T}(\Sigma, \mathcal{V}) \times \mathcal{T}(\Sigma, \mathcal{V})$, ordered by set inclusion.

Then we can lift τ to a Galois Insertion of \mathbb{A} into \mathbb{E} by defining

$$\begin{aligned}\alpha(E) &:= \{\tau(s) = \tau(t) \mid s = t \in E\} \\ \gamma(A) &:= \{s = t \mid \tau(s) = \tau(t) \in A\}\end{aligned}$$

The only requirement we put on τ is that $\alpha(\text{Sem}(\mathcal{R}))$ is finite.

Now we can derive the optimal abstract version of $T_{\mathcal{R}}$ simply as $T_{\mathcal{R}}^{\alpha} := \alpha \circ T_{\mathcal{R}} \circ \gamma$. By applying the previous definition of α and γ this turns out to be equivalent to the following definition.

Definition 3. *Let τ be a term abstraction, $X \in \mathbb{A}$ be an abstract Herbrand interpretation and \mathcal{R} be a TRS. Then,*

$$T_{\mathcal{R},B}^{\alpha}(X) = \{\tau(s) = \tau(t) \mid s = t \in B\} \cup \{\tau(s) = \tau(t) \mid \tau(r) = \tau(t) \in X, s \rightarrow_{\mathcal{R}} r\}$$

Abstract interpretation theory assures that $T_{\mathcal{R},B}^{\alpha} \uparrow \omega$ is the best correct approximation of $\text{Sem}_B(\mathcal{R})$. Correct means $T_{\mathcal{R},B}^{\alpha} \uparrow \omega \sqsubseteq \alpha(\text{Sem}_B(\mathcal{R}))$ and best means that it is the maximum w.r.t. \sqsubseteq of all correct approximations.

Now we can define the abstract semantics as the least fixpoint of this (obviously continuous) operator.

Definition 4. *The abstract least fixpoint semantics of a program \mathcal{R} w.r.t. a set of final/blocking state pairs B , is defined as $\mathcal{F}_B^{\alpha}(\mathcal{R}) = T_{\mathcal{R},B}^{\alpha} \uparrow \omega$.*

By our finiteness assumption on τ we are guaranteed to reach the fixpoint in a finite number of steps, that is, there exists a finite natural number h such that $T_{\mathcal{R},B}^{\alpha} \uparrow \omega = T_{\mathcal{R},B}^{\alpha} \uparrow h$.

4.1 A case study: The domain *depth(k)*

Now we show how to approximate an infinite set of computed equations by means of a *depth(k)* cut [29], i.e., by using a term abstraction $\tau : \mathcal{T}(\Sigma, \mathcal{V}) \rightarrow \mathcal{T}(\Sigma, \mathcal{V} \cup \hat{\mathcal{V}})$ which cuts terms having a depth greater than k . Terms are cut by replacing each subterm rooted at depth k with a new variable taken from the set $\hat{\mathcal{V}}$ (disjoint from \mathcal{V}). *depth(k)* terms represent each term obtained by instantiating the variables of $\hat{\mathcal{V}}$ with terms built over \mathcal{V} .

First of all we define the term abstraction t/k (for $k \geq 0$) as the *depth(k)* cut of the concrete term t . We denote by T/k the set of *depth(k)* terms $(\mathcal{T}(\Sigma, \mathcal{V} \cup \hat{\mathcal{V}})/k)$. The abstract domain \mathbb{A} is thus $\mathcal{P}(\{a = a' \mid a, a' \in T/k\})$

ordered by the Smyth extension of ordering \leq to sets, i.e. $X \leq_S Y$ iff $\forall y \in Y \exists x \in X : (x \leq y)$ [27]. The resulting abstraction α is $\kappa(E) := \{s/k = t/k \mid s = t \in E\}$.

We provide a simple and effective mechanism to compute the abstract fixpoint semantics.

Proposition 2. *For $k > 0$, the operator $T_{\mathcal{R},B}^\kappa : T/k \times T/k \rightarrow T/k \times T/k$ obtained by Definition 3 holds the property $\tilde{T}_{\mathcal{R},B}^\kappa(X) \leq_S T_{\mathcal{R},B}^\kappa(X)$ w.r.t. the following operator:*

$$\tilde{T}_{\mathcal{R},B}^\kappa(X) = \kappa(B) \cup \{ \sigma(u[l]_p)/k = t \mid u = t \in X, p \in O_{\Sigma \cup \mathcal{V}}(u), \\ l \rightarrow r \ll \mathcal{R}, \sigma = \text{mgu}(u|_p, r) \}$$

Definition 5. *The effective abstract least fixpoint semantics of a program \mathcal{R} w.r.t. a set of final/blocking state pairs B , is defined as $\tilde{\mathcal{F}}_B^\kappa(\mathcal{R}) = \tilde{T}_{\mathcal{R},B}^\kappa \uparrow \omega$.*

Proposition 3 (Correctness). *Let \mathcal{R} be a TRS and $k > 0$.*

1. $\tilde{\mathcal{F}}_B^\kappa(\mathcal{R}) \leq_S \kappa(\mathcal{F}_B(\mathcal{R})) \leq_S \mathcal{F}_B(\mathcal{R})$.
2. For all $e \in \tilde{\mathcal{F}}_{\text{val}}^\kappa(\mathcal{R})$ that are ground, $e \in \mathcal{F}_{\text{val}}(\mathcal{R})$.
3. For all $e \in \tilde{\mathcal{F}}_{\text{nf}}^\kappa(\mathcal{R})$ such that $\text{Var}(e) \cap \hat{\mathcal{V}} = \emptyset$, $e \in \mathcal{F}_{\text{nf}}(\mathcal{R})$.

Example 2. Consider the correct (i.e., intended) version \mathcal{I} of program in Example 1 and take $k = 2$. We have:

$$\text{val}/_2 = \{0=0, \text{s}(0)=\text{s}(0), \text{s}(\text{s}(\hat{x}))=\text{s}(\text{s}(\hat{x}))\}$$

According to the previous definition, the fixpoint abstract semantics is (without equations for `add`):

$$\tilde{\mathcal{F}}_{\text{val}}^2(\mathcal{I}) = \{0 = 0, \text{s}(0) = \text{s}(0), \text{s}(0) = \text{s}(0), \text{s}(\text{s}(\hat{x})) = \text{s}(\text{s}(\hat{x})), \\ \text{double}(0) = 0, \text{s}(\text{double}(\hat{x})) = \text{s}(0), \\ \text{s}(\text{double}(\hat{x})) = \text{s}(\text{s}(\hat{y})), \text{double}(\text{s}(\hat{x})) = \text{s}(0), \\ \text{double}(\text{s}(\hat{x})) = \text{s}(\text{s}(\hat{y})), \text{double}(\text{double}(\hat{x})) = 0, \\ \text{double}(\text{double}(\hat{x})) = \text{s}(0), \text{double}(\text{double}(\hat{x})) = \text{s}(\text{s}(\hat{y})) \}$$

In particular, note that all ground equations

$$\{0=0, \text{s}(0)=\text{s}(0), \text{double}(0)=0\}$$

in $\tilde{\mathcal{F}}_{\text{val}}^2(\mathcal{I})$ belong to the concrete semantics $\mathcal{F}_{\text{val}}(\mathcal{I})$.

5 Abstract diagnosis of functional programs

Program properties which can be of interest are Galois Insertions between the concrete domain (the set of Herbrand interpretations ordered by set inclusion) and the abstract domain chosen to model the property. The following Definition 6 extends to abstract diagnosis the definitions given in [26,16,19] for declarative diagnosis. In the following, \mathcal{I}^α is the specification of the intended behavior of a program w.r.t. the property α .

Definition 6. Let \mathcal{R} be a program and α be a property.

1. \mathcal{R} is partially correct w.r.t. \mathcal{I}^α if $\mathcal{I}^\alpha \sqsubseteq \alpha(\text{Sem}(\mathcal{R}))$.
2. \mathcal{R} is complete w.r.t. \mathcal{I}^α if $\alpha(\text{Sem}(\mathcal{R})) \sqsubseteq \mathcal{I}^\alpha$.
3. \mathcal{R} is totally correct w.r.t. \mathcal{I}^α , if it is partially correct and complete.

It is worth noting that the above definition is given in terms of the abstraction of the concrete semantics $\alpha(\text{Sem}(\mathcal{R}))$ and not in terms of the (possibly less precise) abstract semantics $\text{Sem}^\alpha(\mathcal{R})$. This means that \mathcal{I}^α is the abstraction of the intended concrete semantics of \mathcal{R} . In other words, the specifier can only reason in terms of the properties of the expected concrete semantics without being concerned with (approximate) abstract computations. Note also that our notion of total correctness does not concern termination. We cannot address termination issues here, since the concrete semantics we use is too abstract.

The *diagnosis* determines the “basic” symptoms and, in the case of incorrectness, the relevant rule in the program. This is captured by the definitions of *abstractly incorrect rule* and *abstract uncovered equation*.

Definition 7. Let r be a program rule. Then r is abstractly incorrect if $\mathcal{I}^\alpha \not\sqsubseteq T_{\{r\}}^\alpha(\mathcal{I}^\alpha)$.

Informally, r is abstractly incorrect if it derives a wrong abstract element from the intended semantics.

Definition 8. Let \mathcal{R} be a program. \mathcal{R} has abstract uncovered elements if $T_{\mathcal{R}}^\alpha(\mathcal{I}^\alpha) \not\sqsubseteq \mathcal{I}^\alpha$.

Informally, e is uncovered if there are no rules deriving it from the intended semantics. It is worth noting that checking the conditions of Definitions 7 and 8 requires one application of $T_{\mathcal{R}}^\alpha$ to \mathcal{I}^α , while the standard detection based on symptoms [26] would require the construction of $\alpha(\text{Sem}(\mathcal{R}))$ and therefore a fixpoint computation.

In this section, we are left with the problem of formally establishing the properties of the diagnosis method, i.e., of proving which is the relation between abstractly incorrect rules and abstract uncovered equations on one side, and correctness and completeness, on the other side.

It is worth noting that correctness and completeness are defined in terms of $\alpha(\text{Sem}(\mathcal{R}))$, i.e., in terms of abstraction of the concrete semantics. On the other hand, abstractly incorrect rules and abstract uncovered equations are defined directly in terms of abstract computations (the abstract immediate consequence operator $T_{\mathcal{R}}^\alpha$). The issue of the precision of the abstract semantics becomes therefore relevant in establishing the relation between the two concepts.

Theorem 2. If there are no abstractly incorrect rules in \mathcal{R} , then \mathcal{R} is partially correct w.r.t. \mathcal{I}^α .

Theorem 3. Let \mathcal{R} be partially correct w.r.t. \mathcal{I}^α . If \mathcal{R} has abstract uncovered elements then \mathcal{R} is not complete.

Abstract incorrect rules are in general just a hint about a possible source of errors. Once an abstract incorrect rule is detected, one would have to check on the abstraction of the concrete semantics if there is indeed a bug. This is obviously unfeasible in an automatic way. However we will see that, by adding to the scheme an under-approximation of the intended specification, something worthwhile can still be done.

Real errors can be expressed as incorrect rules according to the following definition.

Definition 9. *Let r be a program rule. Then r is incorrect if there exists an equation e such that $e \in T_{\{r\}}(\mathcal{I})$ and $e \notin \mathcal{I}$.*

Definition 10. *Let \mathcal{R} be a program. Then \mathcal{R} has an uncovered element if there exist an equation e such that $e \in \mathcal{I}$ and $e \notin T_{\mathcal{R}}(\mathcal{I})$.*

The following theorem shows that if the program has an incorrect rule it is also an abstractly incorrect rule.

Theorem 4. *Any incorrect rule is an abstractly incorrect rule.*

The check of Definition 9 (as claimed above) is not effective. This task can be (partially) accomplished by an automatic tool by choosing a suitable under-approximation \mathcal{I}^c of the specification \mathcal{I} , $\gamma(\mathcal{I}^c) \subseteq \mathcal{I}$ (hence $\alpha(\mathcal{I}) \sqsubseteq \mathcal{I}^c$), and checking the behavior of an abstractly incorrect rule against it.

Definition 11. *Let r be a program rule. Then r is provably incorrect using α if $\mathcal{I}^\alpha \not\sqsubseteq T_{\{r\}}^\alpha(\mathcal{I}^c)$.*

Definition 12. *Let \mathcal{R} be a program. Then \mathcal{R} has provably uncovered elements using α if $T_{\mathcal{R}}^\alpha(\mathcal{I}^\alpha) \not\sqsubseteq \mathcal{I}^c$.*

The name “provably incorrect using α ” is justified by the following theorem.

Theorem 5. *Let r be a program rule and \mathcal{I}^c such that $(\alpha\gamma)(\mathcal{I}^c) = \mathcal{I}^c$. Then if r is provably incorrect using α it is also incorrect.*

Thus by choosing a suitable under-approximation we can refine the check for wrong rules. For all abstractly incorrect rules we check if they are provably incorrect using α . If it so then we report an error, otherwise we can just issue a warning.

As we will see in the following, this property holds (for example) for our case study. By Proposition 3 the condition $(\alpha\gamma)(\mathcal{I}^c) = \mathcal{I}^c$ is trivially satisfied by any ground subset of the over-approximation. Thus we will consider the best choice which is the biggest ground subset of the over-approximation.

Theorem 6. *Let \mathcal{R} be a program. If \mathcal{R} has a provably uncovered element using α , then \mathcal{R} is not complete.*

Abstract uncovered elements are provably uncovered using α . However, Theorem 6 allows us to catch other incompleteness bugs that cannot be detected by using Theorem 3 since there are provably uncovered elements using α which are not abstractly uncovered.

The diagnosis w.r.t. approximate properties is always effective, because the abstract specification is finite. As one can expect, the results may be weaker than those that can be achieved on concrete domains just because of approximation. Namely,

- absence of abstractly incorrect rules implies partial correctness,
- every incorrectness error is identified by an abstractly incorrect rule. However an abstractly incorrect rule does not always correspond to a bug. Anyway,
- every abstractly incorrect rule which is provably incorrect using α corresponds to an error.
- provably uncovered equations always correspond to incompleteness bugs.
- there exists no sufficient condition for completeness.

The results are useful and comparable to those obtained by verification techniques (see, for example, [3,2]). In fact, if we consider the case where specifications consist of post-conditions only, both abstract diagnosis and verification provide *a sufficient condition for partial correctness*, which is well-assertedness in the case of verification and absence of incorrect rules in abstract diagnosis. For both techniques there is no sufficient condition for completeness. In order to verify completeness, we have to rely on a fixpoint (the model of a transformed program or the abstraction of the concrete semantics), which, in general, cannot be computed in a finite number of steps. As expected, abstract diagnosis (whose aim is locating bugs rather than just proving correctness) gives us also information useful for debugging, by means of provably incorrect rules using α and provably uncovered equations using α .

5.1 Our case study

We can derive an efficient debugger which is 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 and determine program bugs: \mathcal{I}^α which over-approximates the intended semantics \mathcal{I} (that is, $\mathcal{I} \subseteq \gamma(\mathcal{I}^\alpha)$) and \mathcal{I}^c which under-approximates \mathcal{I} (that is, $\gamma(\mathcal{I}^c) \subseteq \mathcal{I}$).

Now we show how we can derive an efficient debugger by choosing suitable instances of the general framework described above. We consider as α the *depth*(k) abstraction κ of the set of values of the TRS that we have defined in previous section. Thus we choose $\mathcal{I}^\kappa = \mathcal{F}_{\text{val}}^\kappa(\mathcal{I})$ as an over-approximation of the values of a program. We can consider any of the sets defined in the works of [6,10] as an under-approximation of \mathcal{I} . In concrete, we take the “ground” abstract equations of \mathcal{I}^κ as \mathcal{I}^c . This provides a simple albeit useful debugging scheme which is satisfactory in practice.

The methodology enforced by previous results (in particular Theorem 5) has been implemented by a prototype system DEBUSSY, which is available at

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

The systems is implemented in Haskell and debugs programs written in OBJ style w.r.t. a formal specification also written in OBJ. The current version only considers the evaluation semantics $Sem_{val}(\mathcal{R})$. The tool takes advantage from the sorting information that may be provided within the programs to construct the (least) sets of blocking terms and equations which are used to generate the approximation of the semantics. The user interface uses textual menus which are (hopefully) self-explaining. A detailed description of the system can be found at the same address.

Let us illustrate the method by using the guiding example.

Example 3. Let us reconsider the TRS \mathcal{R} (program ERRDOUBLE) and the intended specification \mathcal{I} (program ADDDOUBLE) in Example 1 and let us see how it can be debugged by DEBUSSY. The debugging session returns:

```

Incorrect rules in ERRDOUBLE :

double(s(X)) -> double(X)

Uncovered equations from ADDDOUBLE :

add(0,0) = 0

```

where the second rule is correctly identified as erroneous.

In this example, we also show the over and under-approximation computed by the system. Here, we consider a cut at depth 2.

1. Over-approximation $\mathcal{I}^\alpha = \mathcal{I}^2 = \tilde{\mathcal{F}}_{val}^2(\mathcal{I}) = \tilde{\mathcal{T}}_{\mathcal{I},val}^2 \uparrow \omega$.

$$\begin{aligned}
\tilde{\mathcal{T}}_{\mathcal{I},val}^2 \uparrow 0 &= \text{val} = \{ 0 = 0, s(0) = s(0), s(s(\hat{x})) = s(s(\hat{x})) \} \\
\tilde{\mathcal{T}}_{\mathcal{I},val}^2 \uparrow 1 &= \tilde{\mathcal{T}}_{\mathcal{I},val}^2 \uparrow 0 \cup \{ \text{add}(0,0) = 0, \text{add}(0,s(\hat{x})) = s(0), \\
&\quad s(\text{add}(\hat{x},\hat{y})) = s(0), \text{add}(0,s(\hat{x})) = s(s(\hat{y})), \\
&\quad s(\text{add}(\hat{x},\hat{y})) = s(s(\hat{z})) \} \\
\tilde{\mathcal{T}}_{\mathcal{I},val}^2 \uparrow 2 &= \tilde{\mathcal{T}}_{\mathcal{I},val}^2 \uparrow 1 \cup \{ \text{double}(0) = 0, \text{add}(0,\text{add}(\hat{x},\hat{y})) = \\
&\quad 0, \text{add}(\text{add}(\hat{x},\hat{y}),0) = 0, \text{add}(0,\text{add}(\hat{x},\hat{y})) = s(0), \\
&\quad \text{add}(\text{add}(\hat{x},\hat{y}),s(\hat{z})) = s(0), s(\text{double}(\hat{x})) = s(0), \\
&\quad \text{add}(s(\hat{x}),\hat{y}) = s(0), \text{add}(0,\text{add}(\hat{x},\hat{y})) = s(s(\hat{z})), \\
&\quad \text{add}(\text{add}(\hat{x},\hat{y}),s(\hat{z})) = s(s(\hat{w})), s(\text{double}(\hat{x})) = s(s(\hat{y})), \\
&\quad \text{add}(s(\hat{x}),\hat{y}) = s(s(\hat{z})) \}
\end{aligned}$$

$$\begin{aligned}
\tilde{T}_{\mathcal{I},\text{val}}^2 \uparrow 3 &= \tilde{T}_{\mathcal{I},\text{val}}^2 \uparrow 2 \cup \{ \text{add}(0, \text{double}(\hat{x})) = 0, \text{double}(\text{add}(\hat{x}, \hat{y})) \\
&= 0, \text{add}(\text{add}(\hat{x}, \hat{y}), \text{add}(\hat{z}, \hat{w})) = 0, \text{add}(\text{double}(\hat{x}), 0) = \\
&= 0, \text{add}(0, \text{double}(\hat{x})) = \text{s}(0), \text{add}(\text{add}(\hat{x}, \hat{y}), \text{add}(\hat{z}, \hat{w})) \\
&= \text{s}(0), \text{add}(\text{double}(\hat{x}), \text{s}(\hat{y})) = \text{s}(0), \text{double}(\text{s}(\hat{x})) \\
&= \text{s}(0), \text{add}(\text{add}(\hat{x}, \hat{y}), \hat{z}) = \text{s}(0), \text{add}(0, \text{double}(\hat{x})) \\
&= \text{s}(\text{s}(\hat{y})), \text{add}(\text{add}(\hat{x}, \hat{y}), \text{add}(\hat{z}, \hat{w})) = \text{s}(\text{s}(\hat{v})), \\
&\text{add}(\text{double}(\hat{x}), \text{s}(\hat{y})) = \text{s}(\text{s}(\hat{z})), \text{double}(\text{s}(\hat{x})) = \\
&\text{s}(\text{s}(\hat{y})), \text{add}(\text{add}(\hat{x}, \hat{y}), \hat{z}) = \text{s}(\text{s}(\hat{w})) \} \\
\tilde{T}_{\mathcal{I},\text{val}}^2 \uparrow 4 &= \tilde{T}_{\mathcal{I},\text{val}}^2 \uparrow 3 \cup \{ \text{add}(\text{add}(\hat{x}, \hat{y}), \text{double}(\hat{z})) = 0, \\
&\text{double}(\text{double}(\hat{x})) = 0, \text{add}(\text{double}(\hat{x}), \text{add}(\hat{y}, \hat{z})) = 0, \\
&\text{add}(\text{add}(\hat{x}, \hat{y}), \text{double}(\hat{z})) = \text{s}(0), \text{double}(\text{add}(\hat{x}, \hat{y})) \\
&= \text{s}(0), \text{add}(\text{double}(\hat{x}), \text{add}(\hat{y}, \hat{z})) = \text{s}(0), \\
&\text{add}(\text{double}(\hat{x}), \hat{y}) = \text{s}(0), \text{add}(\text{add}(\hat{x}, \hat{y}), \text{double}(\hat{z})) \\
&= \text{s}(\text{s}(\hat{w})), \text{double}(\text{add}(\hat{x}, \hat{y})) = \text{s}(\text{s}(\hat{z})), \\
&\text{add}(\text{double}(\hat{x}), \text{add}(\hat{y}, \hat{z})) = \text{s}(\text{s}(\hat{w})), \text{add}(\text{double}(\hat{x}), \hat{y}) \\
&= \text{s}(\text{s}(\hat{w})) \} \\
\tilde{T}_{\mathcal{I},\text{val}}^2 \uparrow 5 &= \tilde{T}_{\mathcal{I},\text{val}}^2 \uparrow 4 \cup \{ \text{add}(\text{double}(\hat{x}), \text{double}(\hat{y})) = 0, \\
&\text{add}(\text{double}(\hat{x}), \text{double}(\hat{y})) = \text{s}(0), \text{double}(\text{double}(\hat{x})) \\
&= \text{s}(0), \text{add}(\text{double}(\hat{x}), \text{double}(\hat{y})) = \text{s}(\text{s}(\hat{z})), \\
&\text{double}(\text{double}(\hat{x})) = \text{s}(\text{s}(\hat{y})) \} \\
\tilde{T}_{\mathcal{I},\text{val}}^2 \uparrow \omega &= \tilde{T}_{\mathcal{I},\text{val}}^2 \uparrow 5
\end{aligned}$$

- Under-approximation \mathcal{I}^c , which is the ground part of \mathcal{I}^α . Note that $\mathcal{I}^c = (\alpha\gamma)(\mathcal{I}^c)$.

$$\mathcal{I}^c = \{ 0 = 0, \text{s}(0) = \text{s}(0), \text{add}(0, 0) = 0, \text{double}(0) = 0 \}$$

The selection of the appropriate depth for the abstraction is a sensitive point of our approach. The following theorem shows that a threshold depth k exists such that smaller depths are unfeasible to consider.

Theorem 7. *Let $l \rightarrow r$ be a program rule and $k > 0$, If $r/k \not\cong r$, then $l \rightarrow r$ is not provably incorrect using k .*

Definition 13. *Let \mathcal{R} be a TRS. Depth k is called admissible to diagnose \mathcal{R} if for all $l \rightarrow r \in \mathcal{R}$, $r/k \cong r$.*

Obviously, admissible depths do not generally guarantee that all program bugs are recognized, since the abstraction might not be precise enough, as illustrated in the following example. The question of whether an optimal depth exists such that no additional errors are detected by considering deeper cuts is an interesting open problem in our approach which we plan to investigate as further work.

Example 4. Consider the following (wrong) OBJ program \mathcal{R} for doubling and the specification of the intended semantics (program `ADDDOUBLE`) of Example 1.

```
obj ERRDOUBLE2 is
  sort Nat .
```

```

op 0 : -> Nat .
op s : Nat -> Nat .
op double : Nat -> Nat .
var X : Nat .
eq double(0) = s(0) .
eq double(s(X)) = s(double(X)) .
endo

```

The execution of DEBUSSY for program ERRDOUBLE2 and specification ADDDOUBLE is:

Incorrect rules in ERRDOUBLE2 :

```
double(0) -> s(0)
```

Uncovered equations from ADDDOUBLE :

```
add(0,0) = 0
double(0) = 0
```

Note that 2 is the smaller admissible depth for this program. When depth $k = 1$ is considered, rule $\text{double}(0) \rightarrow \text{s}(0)$ can not be proven to be incorrect using the under-approximation $\mathcal{I}^c = \{0 = 0\}$ since equation $\text{s}(0) = \text{s}(0)$ does not belong to \mathcal{I}^c . For instance, by using $k = 2$, the debugger is not able to determine that rule $\text{double}(\text{s}(X)) \rightarrow \text{s}(\text{double}(X))$ is incorrect.

6 Conclusions

We have presented a generic scheme for the declarative debugging of functional programs. Our approach is based on the ideas of [10,1] which we apply to the diagnosis of functional programs. We have presented a fixpoint semantics $T_{\mathcal{R}}$ for functional programs. Our semantics allows us to model the (evaluation/normalization) semantics of the TRS in a bottom-up manner. Thus, it is a suitable basis for dataflow analyses based on abstract interpretation as we illustrated. This methodology is superior to the abstract rewriting methodology of [5], which requires canonicity, stratification, constructor discipline, and complete definedness for the analyses. We have developed a prototype Haskell implementation of our debugging method for functional programs, and we have used it for debugging the examples presented in this work. Nevertheless, more experimentation is needed in order to assess how our methodology performs in comparison to other tools for revealing errors in functional programs such as QuickCheck [7].

Some topics for further research are to develop specialized analyses for particular languages, such as those in the OBJ family. We also plan to endow DEBUSSY with some inductive learning capabilities which allow us to repair program bugs by automatically synthesizing the correction from the examples which can be generated as an outcome of the diagnoser.

References

1. M. Alpuente, F. J. Correa, and M. Falaschi. Declarative Debugging of Functional Logic Programs. In B. Gramlich and S. Lucas, editors, *Proceedings of the International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, North Holland, 2001. Elsevier Science Publishers.
2. K. R. Apt. *From Logic Programming to PROLOG*. Prentice-Hall, 1997.
3. K. R. Apt and E. Marchiori. Reasoning about PROLOG programs: from Modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.
4. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
5. D. Bert and R. Echahed. Abstraction of Conditional Term Rewriting Systems. In J. W. Lloyd, editor, *Proceedings of the 1995 Int'l Symposium on Logic Programming (ILPS'95)*, pages 162–176, Cambridge, Mass., 1995. The MIT Press.
6. 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 M. Kamkar, editor, *Proceedings of the AADEBUG'97 (The Third International Workshop on Automated Debugging)*, pages 155–169, Linköping, Sweden, 1997. University of Linköping Press.
7. K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, 35(9):268–279, 2000.
8. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building Equational Proving Tools by Reflection in Rewriting Logic. In K. Futatsugi, A. Nakagawa, and T. Tamai, editors, *Cafe: An Industrial-Strength Algebraic Formal Method*, pages 1–32. Elsevier, 2000.
9. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of Logic Programs by Abstract Diagnosis. In M. Dams, editor, *Proceedings of Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop (LOMAPS'96)*, volume 1192 of *Lecture Notes in Computer Science*, pages 22–50, Berlin, 1996. Springer-Verlag.
10. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract Diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.
11. M. Comini, G. Levi, and G. Vitiello. Declarative Diagnosis Revisited. In J. W. Lloyd, editor, *Proceedings of the 1995 Int'l Symposium on Logic Programming (ILPS'95)*, pages 275–287, Cambridge, Mass., 1995. The MIT Press.
12. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.
13. F. Durán. Termination Checker and Knuth-Bendix Completion Tools for Maude Equational Specifications. Technical report, Universidad de Málaga, July 2000.
14. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.
15. 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.
16. G. Ferrand. Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Method. *Journal of Logic Programming*, 4(3):177–198, 1987.

17. J. A. Goguen and G. Malcom. *Software Engineering with OBJ*. Kluwer Academic Publishers, Boston, 2000.
18. J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.
19. J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.
20. M. J. Maher. Equivalences of Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, Los Altos, Ca., 1988.
21. H. Nilsson. Tracing piece by piece: affordable debugging for lazy functional languages. In *Proceedings of the 1999 ACM SIGPLAN Int'l Conf. on Functional Programming*, pages 36 – 47. ACM Press, 1999.
22. H. Nilsson and P. Fritzon. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(1):337–370, 1994.
23. J. T. O'Donnell and C. V. Hall. Debugging in Applicative Languages. *Lisp and Symbolic Computation*, 1(2):113–145, 1988.
24. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.
25. R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, Reading, MA, 1993.
26. E. Y. Shapiro. Algorithmic Program Debugging. In *Proceedings of Ninth Annual ACM Symp. on Principles of Programming Languages*, pages 412–531. ACM Press, 1982.
27. M.B. Smyth. Power Domains. *Journal of Computer and System Sciences*, 16:23–36, 1978.
28. J. Sparud and H. Nilsson. The architecture of a debugger for lazy functional languages. In M. Ducassé, editor, *Proceedings Second International Workshop on Automated and Algorithmic Debugging, AADEBUG'95*, 1995.
29. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In S. A. Tärnlund, editor, *Proceedings of Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.
30. I. Toyn. *Exploratory Environments for Functional Programming*. PhD thesis, University of York, U.K., 1987.
31. P. Wadler. Functional Programming: An angry half-dozen. *ACM SIGPLAN Notices*, 33(2):25–30, 1998.