Abstract Correction of OBJ-like Programs [⋆]

M. Alpuente[1], D. Ballis[2], S. Escobar[1], M. Falaschi[2], and S. Lucas[1]

[1] DSIC, Universidad Politécnica de Valencia, Camino de Vera s/n, Apdo. 22012, 46071 Valencia, Spain. {alpuente,sescobar,slucas}@dsic.upv.es.
[2] Dip. Matematica e Informatica, Via delle Scienze 206, 33100 Udine, Italy. {demis,falaschi}@dimi.uniud.it.

Abstract. Debussy is an (abstract) declarative diagnosis tool for functional programs which are written in OBJ style. The debugger does not require the user to either provide error symptoms in advance or answer any question concerning program correctness. In this paper, we formalize an inductive learning methodology for repairing program bugs in OBJ-like programs. Correct program rules are automatically synthesized from examples which might be generated as an outcome by the Debussy diagnoser.

1 Introduction

This paper is motivated by the fact that the debugging support for functional languages in current systems is poor [16], 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 [6, 13, 14] but which do not enforce program correctness adequately as they do not provide means for finding nor reparing bugs in the source code w.r.t. the intended program semantics. This is particularly dramatic for equational languages such as those in the OBJ family, which includes OBJ3, CafeOBJ and Maude.

Abstract diagnosis of functional programs [2] is a declarative diagnosis framework extending the methodology of [8], which relies on (an approximation of) the immediate consequence operator $T_{\mathcal{R}}$, to identify bugs in functional programs. Given the intended specification $\mathcal{I}$ of the semantics of a program $\mathcal{R}$, the debugger checks the correctness of $\mathcal{R}$ by a single step of the abstract immediate consequence operator $T^{\kappa}_{\mathcal{R}}$, where the abstraction function κ stands for $depth(k)$ cut [8]. Then, by a simple static test, the system can determine all the rules which are wrong w.r.t. a particular abstract property.

In this paper, we endow the functional debugging method of [2] with a bug-correction program synthesis methodology which, after diagnosing the buggy

[⋆] This work has been partially supported by CICYT under grant TIC2001-2705-C03-01, MCYT grants HA2001-0059, HI2000-0161 and by Generalitat Valenciana under grant GR03/025.

program, tries to correct the erroneous components of the wrong code automatically. The method uses unfolding in order to discriminate positive from negative examples (resp. uncovered and incorrect equations) which are automatically produced as an outcome by the diagnoser. Informally, our correction procedure works as follows. Starting from an *overly general* program (that is, a program which covers all the positive examples as well as some negative ones), the algorithm unfolds the program and deletes program rules until reaching a suitable specialization of the original program which still covers all the positive examples and does not cover any negative one. Both, the example generation and the top-down correction processes, exploit some properties of the abstract interpretation framework of [2] which they rely on. Let us emphasize that we do not require any demanding condition on the class of the programs which we consider. This is particularly convenient in this context, since it should be undesirable to require strong properties, such as termination or confluence, to a buggy program which is known to contain errors.

We would like to clarify the contributions of this paper w.r.t. [1], where a different unfolding-based correction method was developed which applies to synthetizing multiparadigm, functional-logic programs from a set of positive and negative examples. First, the method for automatically generating the example sets is totally new. In [1] (abstract) non-ground examples were computed as the outcome of an abstract debugger based on the *loop-check* techniques of [3], whereas now we compute (concrete) ground examples after a *depth-k* abstract diagnosis phase [8] which is conceptually much simpler and allows us to compute the example sets more efficiently. Regarding the top-down correction algorithm, the one proposed in this paper significantly improves the method in [1]. We have been able to devise an abstract technique for testing the "overgenerality" applicability condition, which saves us from requiring program termination or the slightly weaker condition of *μ-termination* (termination of context-sensitive rewriting [11]). Finally, the new algorithm works for a much larger class of programs, since we do not even need confluence whereas [1] applies only to inductively sequential programs or noetherian constructor systems (depending on the lazy/eager narrowing strategy chosen).

The rest of the paper is organized as follows. Section 2 summarizes some preliminary definitions and notations. Section 3 recalls the abstract diagnosis framework for functional programs of [2]. Section 4 formalizes the correction problem in this framework. Section 5 illustrates the example generation methodology. Section 6 presents the top-down correction method together with some examples. Section 7 concludes.

2 Preliminaries

Term rewriting systems provide an adequate computational model for functional languages. In this paper, we follow the standard framework of term rewriting (see [4]). For simplicity, definitions are given in the one-sorted case. The extension to many–sorted signatures is straightforward, see [15]. In the paper, syntactic equality of terms is represented by $\equiv$. By $\mathcal{V}$ we 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 ($\mathcal{H}_\Sigma$) over Σ and will be simply 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, where Λ denotes the empty sequence. 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 $\theta_{\restriction s}$ to denote the restriction of the substitution θ to the set of variables in the syntactic object s. The *empty substitution* is denoted by 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 $s\theta \equiv t\theta$, and θ is called a *unifier* of E. We let $mgu(E)$ denote 'the' *most general unifier* of the equation set E [12].

A *term rewriting system* (TRS for short) is a pair $(\Sigma, \mathcal{R})$, where $\mathcal{R}$ is a finite set of reduction (or rewrite) rules of the form $\lambda \to \rho$, λ, $\rho \in \mathcal{T}(\Sigma, \mathcal{V})$, $\lambda \notin \mathcal{V}$ and $Var(\rho) \subseteq Var(\lambda)$. Term λ is called the *left-hand side* (lhs) of the rule and ρ is called the *right-hand side* (rhs). We will often write just $\mathcal{R}$ instead of $(\Sigma, \mathcal{R})$ and call $\mathcal{R}$ the program. 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. Given a TRS $(\Sigma, \mathcal{R})$, we assume that the signature Σ is partitioned into two disjoint sets $\Sigma := \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{D} := \{f \mid f(t_1, \ldots, t_n) \to r \in \mathcal{R}\}$ 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*, while elements in $\mathcal{T}(\mathcal{C})$ are called *values*. 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. A TRS $\mathcal{R}$ is a *constructor system* (CS), if all lhs's of $\mathcal{R}$ are patterns. A TRS $\mathcal{R}$ is *left-linear* (LL), if no variable appears more than once in the lhs of any rule of $\mathcal{R}$.

A rewrite step is the application of a rewrite rule to an expression. A term s *rewrites* to a term t via $r \ll \mathcal{R}$, $s \to_r t$, if there exist $u \in O_\Sigma(s)$, $r \equiv \lambda \to \rho$, and substitution σ such that $s|_u \equiv \lambda\sigma$ and $t \equiv s[\rho\sigma]_u$. We say that $\mathcal{S} := t_0 \to_{r_0} t_1 \to_{r_1} t_2 \ldots \to_{r_{n-1}} t_n$ is a *rewrite sequence* from term t_0 to term t_n. When no confusion can arise, we will omit any subscript (i.e. $s \to t$). A term s is a *normal form*, if there is no term t with $s \to_\mathcal{R} t$. t is the normal form of s if $s \to^*_\mathcal{R} t$ and

t is a normal form (in symbols $s \rightarrow^!_{\mathcal{R}} t$). We say that a TRS $\mathcal{R}$ is *terminating*, if there is no infinite rewrite sequence $t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} \ldots$

The narrowing mechanism is commonly applied to evaluate terms containing variables. Narrowing non-deterministically instantiates variables so that a rewrite step is enabled. This is done by computing mgu's. Formally, $s \overset{\sigma,p}{\leadsto}_r t$ is a *narrowing step* via $r \ll \mathcal{R}$, if there exist $p \in O_\Sigma(s)$ and $r \equiv \lambda \rightarrow \rho$ such that $\sigma = mgu(\{\lambda = s|_p\})$ and $t \equiv s[\rho]_p\sigma$.

3 Denotation of functional programs

In this section we first recall the semantic framework introduced in [2]. We will provide a finite/angelic relational semantics [9], given in fixpoint style, which associates an input-output relation to a program, while intermediate computation steps are ignored. Then, we formulate an abstract semantics which approximates the evaluation semantics of the program.

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 [10]. $\mathcal{H}_\mathcal{V}$ 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. $\sigma(s) = \sigma(t) \equiv s' = t'$.

3.1 Concrete semantics

The considered 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. Since in functional programming, programmers are generally concerned with computing values (ground constructor normal forms), the semantics which is usually considered is $Sem_{\mathsf{val}}(\mathcal{R}) := \{s = t \mid s \rightarrow^!_{\mathcal{R}} t, t \in \mathcal{T}(\mathcal{C})\}$. Sometimes, we will call *proof* of equation $s = t$, a rewrite sequence from term s to value t.

Following [9], in order to formalize our evaluation semantics via fixpoint computation, we consider the following immediate consequence operator.

Definition 1. [2] *Let $\mathcal{I}$ be a Herbrand interpretation, $\mathcal{R}$ be a TRS. Then,*

$$T_{\mathcal{R}}(\mathcal{I}) = \{t = t \mid t \in \mathcal{T}(\mathcal{C})\} \cup \{s = t \mid r = t \in \mathcal{I}, s \rightarrow_{\mathcal{R}} r\}.$$

The following proposition is immediate.

Proposition 1. [2] *Let $\mathcal{R}$ be a TRS. The $T_{\mathcal{R}}$ operator is continuous on $\mathbb{E}$.*

Definition 2. [2] *The least fixpoint semantics of a program $\mathcal{R}$ is defined as $\mathcal{F}_{\mathsf{val}}(\mathcal{R}) = T_{\mathcal{R}} \uparrow \omega$.*

Example 1. Suppose you toss a coin after having chosen one of its faces. If the face revealed after the coin flip is the predicted one, you win a prize. The problem can be modeled by the following specification $\mathcal{I}$ (written in OBJ-like syntax):

```
obj GAMESPEC is
sorts Nat Reward .
   op 0 : -> Nat .
   op s : Nat -> Nat .
   op prize : -> Reward .
   op sorry-no-prize : -> Reward .
   op coinflip : Nat -> Reward .
   op win? : Nat -> Reward .
   var X : Nat .
   eq coinflip(X) = win?(X) .
   eq win?(s(s(X))) = sorry-no-prize .
   eq win?(s(0)) = prize .
   eq win?(0) = sorry-no-prize .
endo
```

Face values are expressed by naturals `0` and `s(0)`; besides, specification $\mathcal{I}$ tells us that we win the prize at stake (expressed by the constructor *prize*), if the revealed face is `s(0)`, while we get no prize whenever the revealed face is equal to `0`. The associated least fixpoint semantics is

$$\begin{aligned}\mathcal{F}_{\mathsf{val}}(\mathcal{I}) = \{\ & \texttt{prize} = \texttt{prize}, \texttt{sorry-no-prize} = \texttt{sorry-no-prize}, \\ & \texttt{win?(0)} = \texttt{sorry-no-prize}, \texttt{win?(s(0))} = \texttt{prize}, \\ & \texttt{win?(s(s(X))} = \texttt{sorry-no-prize}, \texttt{coinflip(0)} = \texttt{sorry-no-prize}, \\ & \texttt{coinflip(s(0))} = \texttt{prize}, \texttt{coinflip(s(s(X))} = \texttt{sorry-no-prize}\}.\end{aligned}$$

The following result establishes the equivalence between the (fixpoint) semantics computed by the $T_{\mathcal{R}}$ operator and the evaluation semantics $\mathit{Sem}_{\mathsf{val}}(\mathcal{R})$.

Theorem 1 (soundness and completeness). [2] *Let $\mathcal{R}$ be a TRS. Then, $\mathit{Sem}_{\mathsf{val}}(\mathcal{R}) = \mathcal{F}_{\mathsf{val}}(\mathcal{R})$.*

3.2 Abstract semantics

Starting from the concrete fixpoint semantics of Definition 2, we give an abstract semantics which approximates the concrete one by means of abstract interpretation techniques. In particular, we will focus our attention on abstract interpretations achieved by means of a *depth*(k) cut [8], which allows to finitely approximate an infinite set of computed equations.

First of all we define a *term abstraction* as a function $/_k : (\mathcal{T}(\Sigma, \mathcal{V}), \leq) \to (\mathcal{T}(\Sigma, \mathcal{V} \cup \hat{\mathcal{V}}), \leq)$ 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 $\mathcal{V}$. Note that $/_k$ is finite.

We denote by $T/_k$ the set of *depth*(k) terms $(\mathcal{T}(\Sigma, \mathcal{V} \cup \hat{\mathcal{V}})/_k)$. We choose as abstract domain $\mathbb{A}$ the set $\mathcal{P}(\{a = a' \mid a, a' \in T/_k\})$ ordered by the Smyth's extension of ordering $\leq$ to sets, i.e. $X \leq_S Y$ iff $\forall\, y \in Y \;\exists\, x \in X : x \leq y$. Thus, we can lift the term abstraction $/_k$ to a Galois Insertion of $\mathbb{A}$ into $\mathbb{E}$ by defining

$$\kappa(E) := \{s/_k = t/_k \mid s = t \in E\}$$
$$\gamma(A) := \{s = t \mid s/_k = t/_k \in A\}$$

Now we can derive the optimal abstract version of $T_{\mathcal{R}}$ simply as $T^{\kappa}_{\mathcal{R}} := \kappa \circ T_{\mathcal{R}} \circ \gamma$ and define the abstract semantics of program $\mathcal{R}$ as the least fixpoint of this (obviously) continuous operator, i.e. $\mathcal{F}^{\kappa}_{\mathsf{val}}(\mathcal{R}) := T^{\kappa}_{\mathcal{R}} \uparrow \omega$. Since $/_k$ is finite, we are guaranteed to reach the fixpoint in a finite number of steps, that is, there exists a finite natural number h such that $T^{\kappa}_{\mathcal{R}} \uparrow \omega = T^{\kappa}_{\mathcal{R}} \uparrow h$. Abstract interpretation theory assures that $T^{\kappa}_{\mathcal{R}} \uparrow \omega$ is the best correct approximation of $\mathit{Sem}_{\mathsf{val}}(\mathcal{R})$. Correct means $\mathcal{F}^{\kappa}_{\mathsf{val}}(\mathcal{R}) \leq_S \kappa(\mathit{Sem}_{\mathsf{val}}(\mathcal{R}))$ and best means that it is the maximum w.r.t. $\leq_S$.

By the following proposition, we provide a simple and effective mechanism to compute the abstract fixpoint semantics.

Proposition 2. [2] *For $k > 0$, the operator $T^{\kappa}_{\mathcal{R}} : T/_k \times T/_k \to T/_k \times T/_k$ holds the property $\widetilde{T}^{\kappa}_{\mathcal{R}}(X) \leq_S T^{\kappa}_{\mathcal{R}}(X)$ w.r.t. the following operator:*

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

Definition 3. [2] *The effective abstract least fixpoint semantics of a program $\mathcal{R}$ is defined as $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}) = \widetilde{T}^{\kappa}_{\mathcal{R}} \uparrow \omega$.*

Proposition 3 (Correctness). [2] *Let $\mathcal{R}$ be a TRS and $k > 0$.*

1. $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}) \leq_S \kappa(\mathcal{F}_{\mathsf{val}}(\mathcal{R})) \leq_S \mathcal{F}_{\mathsf{val}}(\mathcal{R})$.
2. *For every $e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$ such that $Var(e) \cap \hat{\mathcal{V}} = \emptyset$, $e \in \mathcal{F}_{\mathsf{val}}(\mathcal{R})$.*

Example 2. Consider again the specification in Example 1. Its effective abstract least fixpoint semantics for $\kappa = 3$ (without considering symbol `win?`) becomes

$$\widetilde{\mathcal{F}}^{3}_{\mathsf{val}}(\mathcal{I}) = \{\ \texttt{prize} = \texttt{prize}, \texttt{sorry-no-prize} = \texttt{sorry-no-prize}, \\ \texttt{coinflip(0)} = \texttt{sorry-no-prize}, \texttt{coinflip(s(0))} = \texttt{prize}, \\ \texttt{coinflip(s(s(}\hat{\texttt{X}}\texttt{)))} = \texttt{sorry-no-prize}\}.$$

4 The Correction Problem

The problem of repairing a faulty functional program can be addressed by using inductive learning techniques guided by appropriate examples. Roughly speaking, given a wrong program and two example sets specifying positive (pursued) and negative (not pursued) computations respectively, our correction scheme aims at synthesizing a set of program rules that replaces the wrong ones in order to deliver a corrected program which is "consistent" w.r.t. the example sets [5]. More formally, we can state the correction problem as follows.

Problem formalization. Let $\mathcal{R}$ be a TRS, $\mathcal{I}$ be the specification of the intended semantics, E^+ and E^- be two finite sets of equations such that

$$E^+ \subseteq Sem_{\mathsf{val}}(\mathcal{I}) \quad \text{and} \quad E^- \cap (Sem_{\mathsf{val}}(\mathcal{R}) \setminus Sem_{\mathsf{val}}(\mathcal{I})) \neq \emptyset.$$

The correction problem consists in constructing a TRS $\mathcal{R}^c$ satisfying the following requirements

$$E^+ \subseteq Sem_{\mathsf{val}}(\mathcal{R}^c) \quad \text{and} \quad E^- \cap Sem_{\mathsf{val}}(\mathcal{R}^c) = \emptyset.$$

Equations in E^+ (resp. E^-) are called *positive* (resp. *negative*) examples. The TRS $\mathcal{R}^c$ is called *correct* program. Note that by construction positive and negative example sets are disjoint, which permits to drive the correction process towards a discrimination between E^+ and E^-.

5 How to generate example sets automatically

Before giving a constructive method to derive a correct program, we present a simple methodology for automatically generating example sets, so that the user does not need to provide error symptoms, evidences or other kind of information which would require a good knowledge of the program semantics that she probably lacks.

In the following, we observe that we can easily compute "positive" equations, i.e. equations which appear in the concrete evaluation semantics $Sem_{\mathsf{val}}(\mathcal{I})$, since all equations in $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I})$ not containing variables in $\hat{\mathcal{V}}$ belong to the concrete evaluation semantics $Sem_{\mathsf{val}}(\mathcal{I})$, as stated in the following lemma.

Lemma 1. *Let $\mathcal{I}$ be a TRS and $E_P := \{e | e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I}) \wedge Var(e) \cap \hat{\mathcal{V}} = \emptyset\}$. Then, $E_P \subseteq Sem_{\mathsf{val}}(\mathcal{I})$.*

Now, by exploiting the information in $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I})$ and $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$, we can also generate a set of "negative" equations which belong to the concrete evaluation semantics $Sem_{\mathsf{val}}(\mathcal{R})$ of the wrong program $\mathcal{R}$ but not to the concrete evaluation semantics $Sem_{\mathsf{val}}(\mathcal{I})$ of the specification $\mathcal{I}$.

Lemma 2. *Let $\mathcal{R}$ be a TRS, $\mathcal{I}$ be a specification of the intended semantics and $E_N := \{e | e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}) \wedge Var(e) \cap \hat{\mathcal{V}} = \emptyset \wedge \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I}) \not\leq_S \{e\}\}$. Then, $E_N \subseteq (Sem_{\mathsf{val}}(\mathcal{R}) \setminus Sem_{\mathsf{val}}(\mathcal{I}))$.*

Starting from sets E_P and E_N, we construct the positive and negative example sets E^+ and E^- which we use for the correctness process, by considering the restriction of E_P and E_N to examples of the form $l = c$ where l is a pattern and c is a value, i.e. a term formed only by constructor symbols and variables. The motivation for this is twofold. On the one hand, it allows us to ensure correctness of the correction algorithm without any further requirements on the class of programs which we consider. On the other hand, by considering these "data" examples, the inductive process becomes independent from the extra auxiliary functions which might appear in $\mathcal{I}$, since we start synthesizing directly from data structures.

The sets E^+ and E^- are defined as follows.

$$\begin{aligned}E^+ &= \{l = c \mid f(t_1,\ldots,t_n) = c \in E_P \wedge f(t_1,\ldots,t_n) \equiv l \text{ is a pattern } \wedge \\ &\qquad\qquad \wedge\, c \in \mathcal{T}(\mathcal{C}) \wedge f \in \Sigma_{\mathcal{R}}\} \\ E^- &= \{l = c \mid l = c \in E_N \wedge l \text{ is a pattern } \wedge c \in \mathcal{T}(\mathcal{C})\}\end{aligned}$$

where $\Sigma_{\mathcal{R}}$ is the signature of program $\mathcal{R}$.

In the sequel, the function which computes the sets E^+ and E^-, according to the above description, is called ExampleGeneration($\mathcal{R},\mathcal{I}$).

6 Program correction via example-guided unfolding

In this section we present a basic top-down correction method which is based on the so-called *example-guided unfolding* [5], which is able to specialize a program by applying unfolding and deletion of program rules until coming up with a correction. The top-down correction process is "guided" by the examples, in the sense that transformation steps focus on discriminating positive from negative examples. The accuracy of the correction improves as the number of positive and negative examples increase as it is common to the learning from examples approach.

In order to successfully apply the method, the semantics of the program to be specialized must include the positive example set E^+ (that is, $E^+ \subseteq \mathit{Sem}_{\mathsf{val}}(\mathcal{R})$). Programs satisfying this condition are called *overly general* (w.r.t. E^+).

The over-generality condition is not generally decidable, as we do not impose program termination [1]. Fortunately, when we consider the abstract semantics framework of [2] we are able to ascertain a useful sufficient condition to decide whether a program is overly general, even if it does not terminate. The following proposition formalizes our method.

Proposition 4. *Let $\mathcal{R}$ be a TRS and E^+ be a set of positive examples. If, for each $e \in E^+$, there exists $e' \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$ s.t. (1) $e' \leq e$ and (2) $\mathit{Var}(e') \cap \hat{\mathcal{V}} = \emptyset$; then, $\mathcal{R}$ is overly general w.r.t. E^+.*

Now, by exploiting Proposition 4, it is not difficult to figure out a procedure OverlyGeneral($\mathcal{R}, E$) testing this condition w.r.t. a program $\mathcal{R}$ and a set of examples E, e.g. a boolean function returning *true* if program $\mathcal{R}$ is overly general w.r.t. E and *false* otherwise.

6.1 The unfolding operator

Informally, *unfolding* a program $\mathcal{R}$ w.r.t. a rule r delivers a new specialized version of $\mathcal{R}$ in which the rule r is replaced with new rules obtained from r by performing a narrowing step on the rhs of r.

The following definition is auxiliary.

Definition 4. *Let $t \in \tau(\Sigma \cup \mathcal{V})$ and $\bot$ be a symbol not in Σ. The* skeleton *of t, in symbols $skel(t)$, is defined as follows (i) $skel(t) = f(skel(t_1), \ldots, skel(t_n))$, if $t \equiv f(t_1, \ldots, t_n)$ and $f \in \mathcal{D}$; (ii) $skel(t) = \bot$ otherwise.*

Given a term t, we define the set of its *top positions* as $TO_\Sigma(t) = O_\Sigma(skel(t))$. In other words, those positions p in t pointing to a defined function symbol such that there is no constructor symbol occurring above p in t.

Example 3. Consider the standard definition of natural numbers given in Peano's syntax. Then, $TO_\Sigma(+(+(\mathtt{X}, \mathtt{Y}), \mathtt{s}(+(\mathtt{X}, \mathtt{Y})))) = \{\Lambda, 1\}$.

Definition 5. *Given two rules $r_1 \equiv \lambda_1 \to \rho_1$ and r_2, we define the* rule unfolding *of r_1 w.r.t. r_2 as $\mathcal{U}_{r_2}(r_1) = \{\lambda_1\sigma \to \rho' \mid \rho_1 \overset{\sigma,p}{\leadsto}_{r_2} \rho', p \in TO_\Sigma(\rho_1)\}$.*

Definition 6. *Given a TRS $\mathcal{R}$ and a rule $r \ll \mathcal{R}$, we define the* program unfolding *of r w.r.t. $\mathcal{R}$ as follows $\mathcal{U}_\mathcal{R}(r) = \left(\mathcal{R} \cup \bigcup_{r' \in \mathcal{R}} \mathcal{U}_{r'}(r)\right) \setminus \{r\}$.*

Note that, by Definition 6, for any TRS $\mathcal{R}$ and rule $r \ll \mathcal{R}$, r is never in $\mathcal{U}_\mathcal{R}(r)$.

Definition 7. *Let $\mathcal{R}$ be a TRS, r be a rule in $\mathcal{R}$. The rule r is* unfoldable *w.r.t. $\mathcal{R}$ if $\mathcal{U}_\mathcal{R}(r) \neq \mathcal{R} \setminus \{r\}$.*

The "transformed" semantics, obtained after applying the unfolding operator to a given left-linear constructor system $\mathcal{R}$ still contains the semantics of $\mathcal{R}$, as stated in the following theorem.

Theorem 2 (unfolding correctness). *Let $\mathcal{R}$ be a left-linear CS, $r \ll \mathcal{R}$ be an unfoldable rule and $\mathcal{R}' = \mathcal{U}_\mathcal{R}(r)$. Let $e \equiv (l = c)$ be an equation such that $l \in \mathcal{T}(\Sigma, \mathcal{V})$ and $c \in \mathcal{T}(\mathcal{C})$. Then, if $e \in Sem_{\mathsf{val}}(\mathcal{R})$, then $e \in Sem_{\mathsf{val}}(\mathcal{R}')$.*

6.2 The top-down correction algorithm

Basically, the idea behind the basic correction algorithm is to eliminate rules from the program in order to get rid of the negative examples without losing the derivations for the positive ones. Clearly, this cannot be done by naïvely removing program rules, since sometimes a rule is used to prove both a positive and a negative example. So, before applying deletion, we need to specialize programs in order to ensure that the deletion phase only affects those program rules which are not necessary for proving the positive examples. This specialization process is carried out by means of the unfolding operator of Definition 6. Considering this operator for specialization purposes has important advantages. First, positive examples are not lost by repeatedly applying the unfolding operator, since unfolding preserves the proper semantics (see Theorem 2). Moreover, the nature of unfolding is to "compile" rewrite steps into the program, which allows us to shorten and distinguish the rewrite rules which occur in the proofs of the positive and negative examples.

Figure 1 shows the correction algorithm, called TDCorrector, which takes as input a program $\mathcal{R}$ and a specification of the intended semantics $\mathcal{I}$, also

```
procedure TDCorrector($\mathcal{R},\mathcal{I}$)
  $(E^+,E^-) \leftarrow$ ExampleGeneration($\mathcal{R},\mathcal{I}$)
  if not OverlyGeneral($\mathcal{R},E^+$) then Halt
  $k \leftarrow 0$; $\mathcal{R}_k \leftarrow \mathcal{R}$
  while $\exists\, e^- \in E^- : \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}_k) \leq_S \{e^-\}$ do
    if $\exists\, r \in \mathcal{R}_k$ s.t. $r$ is unfoldable and $r \in \mathit{First}(E^+)$ then
      $\mathcal{R}_{k+1} \leftarrow \mathcal{U}_{\mathcal{R}_k}(r)$; $k \leftarrow k+1$
      else Halt
    end if
    for each $r \in \mathcal{R}_k$ do
      if OverlyGeneral($\mathcal{R}_k \backslash \{r\}, E^+$) then $\mathcal{R}_k \leftarrow \mathcal{R}_k \backslash \{r\}$
    end for
  end while
end procedure
```

Fig. 1. The top-down correction algorithm.

expressed as a program. First, TDCorrector computes the example sets E^+ and E^- by means of ExampleGeneration, following the method presented in Section 5. Then, it checks whether program $\mathcal{R}$ is overly general following the scheme of Proposition 4, and finally it enters the main correction process.

This last phase consists of a main loop, in which we perform an unfolding step followed by a rule deletion until no negative example is covered (approximated) by the abstract semantics of the current transformed program $\mathcal{R}_n$. This amounts to saying that no negative example belongs to the concrete semantics of $\mathcal{R}_n$. We note that the **while** loop guard is decidable, as the abstract semantics is finitely computable. Note the deep difference w.r.t. the algorithm of [1], where decidability is ensured by requiring both confluence and (μ-termination) of the program.

During the unfolding phase, we select a rule upon which performing a program unfolding step. In order to specialize the program w.r.t. the example sets, we pick up an unfoldable rule which occurs in some proof of a positive example. More precisely, we choose a rule which appears first in a proof of a positive example, i.e. $e \in \mathit{First}(E^+)$, where function *First* is formally defined as follows.

Definition 8. *Let $\mathcal{R}$ be a TRS and E be an example set. Then, we define*

$$\mathit{First}(E) := \bigcup_{e \in E} \{r \mid e \in \widetilde{T}^{\kappa}_{\{r\}}(\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}))\}.$$

Once unfolding has been accomplished, we proceed to remove the "redundant" rules, that is, all the rules which are not needed to prove the positive example set E^+. This can be done by repeatedly testing the overgenerality of the specialized program w.r.t. E^+ and removing one rule at each iteration of the inner **for** loop. Roughly speaking, if program $\mathcal{R}_k \setminus \{r\}$ is overly general w.r.t. E^+, then rule r can be safely eliminated without losing E^+. Then, we can repeat the test on another rule. Let us consider the coin-flip game to illustrate our algorithm.

Example 4. The following OBJ program $\mathcal{R}$ is wrong w.r.t. the specification $\mathcal{I}$ of Example 1.

```
obj GAME is
sorts Nat Reward .
   op 0 : -> Nat .
   op s : Nat -> Nat .
   op prize : -> Reward .
   op sorry-no-prize : -> Reward .
   op coinflip : Nat -> Reward .
   var X : Nat .
   eq coinflip(s(X)) = coinflip(X) .        (1)
   eq coinflip(0) = prize .                 (2)
   eq coinflip(0) = sorry-no-prize .        (3)
endo
```

Note that program $\mathcal{R}$ is non-confluent and computes both values `prize` and `sorry-no-prize` for any natural $\mathtt{s}^n\mathtt{(0)}$, $n > 0$. By fixing $\kappa = 3$, we get the following effective least fixpoint abstract semantics for $\mathcal{R}$.

$$\begin{aligned}\widetilde{\mathcal{F}}^3_{\mathsf{val}}(\mathcal{R}) = \{\, & \mathtt{prize} = \mathtt{prize}, \mathtt{sorry\text{-}no\text{-}prize} = \mathtt{sorry\text{-}no\text{-}prize},\\ & \mathtt{coinflip(0)} = \mathtt{prize}, \mathtt{coinflip(0)} = \mathtt{sorry\text{-}no\text{-}prize},\\ & \mathtt{coinflip(s(0))} = \mathtt{prize}, \mathtt{coinflip(s(0))} = \mathtt{sorry\text{-}no\text{-}prize},\\ & \mathtt{coinflip(s(s(\hat{X})))} = \mathtt{prize}, \mathtt{coinflip(s(s(\hat{X})))} = \mathtt{sorry\text{-}no\text{-}prize}\}.\end{aligned}$$

Cosidering the abstract fixpoint semantics $\widetilde{\mathcal{F}}^3_{\mathsf{val}}(\mathcal{I})$ computed in Example 2 and following the methodology of Section 5, we obtain the example sets below:

$$\begin{aligned}E^+ &= \{\mathtt{coinflip(s(0))} = \mathtt{prize}, \mathtt{coinflip(0)} = \mathtt{sorry\text{-}no\text{-}prize}\}\\ E^- &= \{\mathtt{coinflip(s(0))} = \mathtt{sorry\text{-}no\text{-}prize}, \mathtt{coinflip(0)} = \mathtt{prize}\}.\end{aligned}$$

Now, since program $\mathcal{R}$ fulfills the condition for overgenerality expressed by Proposition 4, the algorithm proceeds and enters the main loop. Here, program rule (1) is unfolded, because (1) is unfoldable and is in *First*(E^+). So, the transformed program is

```
eq coinflip(s(s(X))) = coinflip(X) .     (4)
eq coinflip(s(0)) = prize .              (5)
eq coinflip(s(0)) = sorry-no-prize .     (6)
eq coinflip(0) = prize .                 (7)
eq coinflip(0) = sorry-no-prize .        (8)
```

Subsequently, a deletion phase is executed in order to check whether there are rules not needed to cover the positive example set E^+. The algorithm discovers that rules (4), (6), (7) are not necessary, and therefore are removed producing the correct program which consists of rules (5) and (8).

7 Conclusions

In this paper, we have proposed an example-guided methodology for synthesizing (partially) correct functional programs written in OBJ style, which complements the diagnosis method which was developed previously in [2]. Specifications of the intended semantics, expressed as programs, are used to carry out the diagnosis as well as the correction. This is not only a common practice in logic as well as equational (or term rewriting) languages, but also in functional programming (e.g. QuickCheck [7]).

We want to point out that this method is not comparable to [1] as it is lower-cost and it works for a much wider class of TRSs. In particular, it is able to repair non-confluent programs. This is not only theoretically more challenging, but also convenient in our framework, where it is not reasonable to expect that confluence holds for an erroneous program (even if program confluence was in the programmer's intention). We are currently extending the prototypical implementation of the diagnosis system Debussy [2] (available at `http://www.dsic.upv.es/users/elp/soft.html`) with a correction tool which is based on the proposed abstract correction methodology, and use it for an experimental evaluation of the system.

References

1. M. Alpuente, D. Ballis, F. J. Correa, and M. Falaschi. Automated Correction of Functional Logic Programs. In *Proc. of ESOP 2003*, vol. 2618 of *LNAI*, 2003.
2. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In *Proc. of LOPSTR'02*, Springer LNCS, 2003.
3. M. Alpuente, M. Falaschi, and F. Manzo. Analyses of Unsatisfiability for Equational Logic Programming. *JLP*, 22(3):221–252, 1995.
4. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
5. H. Bostrom and P. Idestam-Alquist. Induction of Logic Programs by Example–guided Unfolding. *JLP*, 40:159–183, 1999.
6. O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood - A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In *Proc. of IFL 2000*, pages 176–193. Springer LNCS 2011, 2001.
7. K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. of ICFP'00*, 35(9):268–279, 2000.
8. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *JLP*, 39(1-3):43–93, 1999.
9. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *TCS*, 277(1-2):47–103, 2002.
10. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *TCS*, 69(3):289–318, 1989.
11. S. Lucas. Termination of (Canonical) Context-Sensitive Rewriting. In *Proc. RTA'02*, pages 296–310. Springer LNCS 2378, 2002.
12. M. J. Maher. Equivalences of Logic Programs. In *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, 1988.
13. H. Nilsson. How to look busy while being as lazy as ever. *JFP*, 11(6):629–671, 2001.
14. J. T. O'Donell and C. V. Hall. Debugging in Applicative Languages. *Lisp and Symbolic Computation*, 1(2):113–145, 1988.
15. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag,1988.
16. P. Wadler. Functional Programming: An angry half-dozen. *ACM SIGPLAN Notices*, 33(2):25–30, 1998.