Abstract Correction of First-Order Functional Programs

M. Alpuente a D. Ballis b S. Escobar a M. Falaschi b
S. Lucas a

a DSIC, Universidad Politécnica de Valencia, Camino de Vera s/n, Apdo. 22012, 46071 Valencia, Spain. Email: \{alpuente, sescobar, slucas\}@dsic.upv.es.
b Dip. Matematica e Informatica, Via delle Scienze 206, 33100 Udine, Italy. Email: \{demis, falaschi\}@dmi.uniud.it.

Abstract

DEBussy is an (abstract) declarative diagnosis tool for functional programs which are written in OBJ style. The tool 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, which is based on the so-called example-guided unfolding [6]. Correct programs are synthesized by unfolding and removing rules of the faulty program. Rules to be unfolded (deleted) are selected according to the examples, which can be automatically generated as an outcome by the DEBussy diagnoser.

Key words: Abstract Correction, First Order Functional Programs, Inductive Learning, Program Transformation, Abstract Interpretation.

1 Introduction

This paper is motivated by the fact that the debugging support for functional languages in current systems is poor [28], and there are no general purpose, good semantics-based debugging tools available, which allow to debug a program w.r.t. a given formal specification. Traditional debugging tools for functional programming languages consist of tracers which help to display the execution [7,14,21,23] but which do not enforce program correctness adequately as they do not provide means for finding nor repairing 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 [9], which relies on (an approxi-
mation of) the immediate consequence operator \( T_R \) [17], to identify bugs in functional programs. Given the intended specification \( I \) of the semantics of a program \( R \), the debugger checks the correctness of \( R \) by a single step of the abstract immediate consequence operator \( T_R^\kappa \), where the abstraction function \( \kappa \) stands for depth \((k)\) cut [9]. Then, by a simple static test, the system can determine all the rules which are wrong w.r.t. a particular abstract property. The framework is goal independent and does not require the determination of symptoms in advance. This is in contrast with traditional, semi-automatic debugging of functional programs [20,22,27], 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, who assumes the role of the oracle. When debugging real code, the questions are often textually large and may be difficult to answer.

In this paper, we endow the functional debugging method of [2] with a bug-correction program synthesis methodology which, after diagnosing the buggy 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 synthesizing 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 [9] 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 \( \mu \)-termination (termination of context-sensitive rewriting [18]). Finally, we have
been able to demonstrate the correctness of the new algorithm for a much larger class of programs, since we do not even need confluence whereas [1] applies only to inductively sequential programs or canonical, completely defined systems (depending on the lazy/eager narrowing strategy chosen).

The debugging methodology which we consider 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. Therefore, a debugging tool which is able to locate bugs in the user’s program and correct the wrong code becomes also important in this context. In general, it also happens that some parts of the software need to be improved during the software life cycle, e.g. 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. For instance, the executability of OBJ specifications supports prototype-driven incremental development methods [15].

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 and correctness results. Section 7 concludes.

2 Preliminaries

Term rewriting systems provide an adequate computational model for first order functional languages. In this paper, we follow the standard framework of term rewriting (see [5,16]). 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 $\equiv$. By $\mathcal{V}$ we denote a countably infinite set of variables and $\Sigma$ denotes a set of function symbols, or signature, each of which has a fixed associated arity. $T(\Sigma, \mathcal{V})$ and $T(\Sigma)$ denote the non-ground word (or term) algebra and the word algebra built on $\Sigma \cup \mathcal{V}$ and $\Sigma$, respectively. $T(\Sigma)$ is usually called the Herbrand universe ($\mathcal{H}_\Sigma$) over $\Sigma$ 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 $\Sigma$-equation $s = t$ is a pair of terms $s, t \in T(\Sigma, \mathcal{V})$, 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 $\Lambda$ denotes the empty sequence. We let $\text{depth}(t)$ denote the depth of the tree (term) $t$. 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 \( \theta \) is more general than \( \sigma \), denoted by \( \theta \leq \sigma \), if \( \sigma = \theta \gamma \) for some substitution \( \gamma \). We write \( \theta|_s \) to denote the restriction of the substitution \( \theta \) to the set of variables in the syntactic object \( s \). The empty substitution is denoted by \( \epsilon \). A renaming is a substitution \( \rho \) for which there exists the inverse \( \rho^{-1} \), such that \( \rho \rho^{-1} = \rho^{-1} \rho = \epsilon \). An equation set \( E \) is unifiable, if there exists \( \theta \) such that, for all \( s = t \in E \), we have \( s\theta \equiv t\theta \), and \( \theta \) is called a unifier of \( E \). We let \( \text{mgu}(E) \) denote the most general unifier of the equation set \( E \) [19].

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 \( \lambda \rightarrow \rho \), \( \lambda, \rho \in \mathcal{T}(\Sigma, \mathcal{V}) \), \( \lambda \notin \mathcal{V} \) and \( \text{Var}(\rho) \subseteq \text{Var}(\lambda) \). Term \( \lambda \) is called the left-hand side (lhs) of the rule and \( \rho \) 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 \( \Sigma \) is partitioned into two disjoint sets \( \Sigma := \mathcal{C} \uplus \mathcal{D} \), where \( \mathcal{D} := \{ f | f(t_1, \ldots, t_n) \rightarrow r \in \mathcal{R} \} \) 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, while elements in \( \mathcal{T}(\mathcal{C}) \) are called values. A pattern is a term of the form \( f(\overline{d}) \) where \( f/n \in \mathcal{D} \) and \( \overline{d} \) 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 \( r \ll \mathcal{R} \), \( s \rightarrow r t \) (or \( s \rightarrow_\mathcal{R} t \)), if there exist \( u \in \text{O}_\Sigma(s) \), \( r \equiv \lambda \rightarrow \rho \), and substitution \( \sigma \) such that \( s|_u \equiv \lambda \sigma \) and \( t \equiv s[\rho \sigma]|_u \). We say that \( \mathcal{S} := t_0 \rightarrow_{r_0} t_1 \rightarrow_{r_1} t_2 \ldots \rightarrow_{r_{n-1}} t_n \) is a rewrite sequence from term \( t_0 \) to term \( t_n \).

When no confusion can arise, we will omit any subscript (i.e. \( s \rightarrow t \)). 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 \( s \rightarrow_\mathcal{R}^* t \)). We say that a TRS \( \mathcal{R} \) is terminating, if there exists 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, \( s \sim_\mathcal{R}^* t \) is a narrowing step via \( r \ll \mathcal{R} \), if there exist \( p \in \text{O}_\Sigma(s) \) and \( r \equiv \lambda \rightarrow \rho \) such that \( \sigma = \text{mgu}(\{ \lambda = s|_p \}) \) 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 [10], 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 [12,13]. \( \mathcal{H}_V \) denotes the \( V \)-Herbrand universe which allows variables in its elements, and is defined as \( T(\Sigma, V)/\equiv \), where \( \equiv \) is the equivalence relation induced by the preorder \( \leq \) of “relative generality” between terms, i.e. \( s \leq t \) if there exists \( \sigma \) s.t. \( t \equiv \sigma(s) \). For the sake of simplicity, the elements of \( \mathcal{H}_V \) (equivalence classes) have the same representation as the elements of \( T(\Sigma, V) \) and are also called terms. \( \mathcal{B}_V \) denotes the \( V \)-Herbrand base, namely, the set of all equations \( s = t \) modulo variance, where \( s, t \in \mathcal{H}_V \). A subset of \( \mathcal{B}_V \) is called a \( V \)-Herbrand interpretation. We assume that the equations in the denotation are renamed in order to avoid that program variables occur in the denotation. The ordering \( \leq \) for terms is extended to equations in the obvious way, i.e. \( s = t \leq s' = t' \) iff there exists \( \sigma \) s.t. \( \sigma(s) = \sigma(t) \equiv s' = t' \).

3.1 Concrete semantics

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

In the sequel, a semantics for program \( \mathcal{R} \) is a \( V \)-Herbrand interpretation. Since in functional programming, programmers are generally concerned with computing values (ground constructor normal forms), the semantics which is usually considered is \( \text{Sem}_{\mathbb{E}}(\mathcal{R}) := \{ s = t \mid s \rightarrow\!_{\mathcal{R}}^{1} t, t \in T(\mathcal{C}) \} \). Sometimes, the rewrite sequence from term \( s \) to value \( t \) is called proof of the equation \( s = t \).

Following [10], in order to formalize our evaluation semantics via fixpoint computation, we consider the following immediate consequence operator.

**Definition 3.1** [2] Let \( \mathcal{I} \) be a Herbrand interpretation, \( \mathcal{R} \) be a TRS. Then,

\[
T_{\mathcal{R}}(\mathcal{I}) = \{ t = t \mid t \in T(\mathcal{C}) \} \cup \{ s = t \mid r = t \in \mathcal{I}, s \rightarrow\!_{\mathcal{R}}^{1} r \}.
\]

The following proposition is immediate.

**Proposition 3.2** [2] Let \( \mathcal{R} \) be a TRS. The \( T_{\mathcal{R}} \) operator is continuous on \( \mathbb{E} \).

**Definition 3.3** [2] The least fixpoint semantics of a program \( \mathcal{R} \) is defined as \( \mathcal{F}_{\mathbb{E}}(\mathcal{R}) = T_{\mathcal{R}} \uparrow \omega \).

**Example 3.4** Suppose to roll a dice. If the dice face revealed after the roll is an odd one, you win a prize; otherwise you are not rewarded. The problem can be modeled by the following specification \( \mathcal{I} \) (written in OBJ-like syntax):
obj GAMESPEC is
  sorts Nat Reward Bool .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op prize : -> Reward .
  op true: -> Bool .
  op false: -> Bool .
  op sorry-no-prize : -> Reward .
  op playdice : Nat -> Reward .
  op odd : Nat -> Bool .
var X : Nat .
eq playdice(X) = win?(odd(X)) .
eq win?(true) = prize .
eq win?(false) = sorry-no-prize .
eq odd(0) = false .
eq odd(s(0)) = true .
eq odd(s(s(X))) = odd(X) .
endo

Face values are expressed by naturals; besides, specification \( I \) tells us that we win the prize at stake (expressed by the constructor \( \text{prize} \)), if the resulting value of the \( \text{odd} \) function is \( \text{true} \), otherwise the value \( \text{sorry-no-prize} \) is computed.

The associated least fixpoint semantics is

\[
F_{\text{val}}(I) = \{ \text{prize} = \text{prize}, \text{sorry-no-prize} = \text{sorry-no-prize}, \\
\text{true} = \text{true}, \text{false} = \text{false}, 0 = 0, s(0) = s(0), \ldots \} \cup \\
\{ \text{odd}(s^n(0)) = \text{true} \mid n \text{ is odd} \} \cup \\
\{ \text{odd}(s^n(0)) = \text{false} \mid n \text{ is even} \} \cup \\
\{ \text{win?}(\text{true}) = \text{prize}, \text{win?}(\text{false}) = \text{sorry-no-prize} \} \cup \\
\{ \text{win?}(\text{odd}(s^n(0))) = \text{prize} \mid n \text{ is odd} \} \cup \\
\{ \text{win?}(\text{odd}(s^n(0))) = \text{sorry-no-prize} \mid n \text{ is even} \} \cup \\
\{ \text{playdice}(s^n(0)) = \text{prize} \mid n \text{ is odd} \} \cup \\
\{ \text{playdice}(s^n(0)) = \text{sorry-no-prize} \mid n \text{ is even} \}
\]

The following result establishes the equivalence between the (fixpoint) semantics computed by the \( T_R \) operator and the evaluation semantics \( \text{Sem}_{\text{val}}(R) \).

**Theorem 3.5 (soundness and completeness)** [2] Let \( R \) be a TRS. Then,

\[
\text{Sem}_{\text{val}}(R) = F_{\text{val}}(R).
\]

Note that if \( e \equiv (l = c) \) belongs to the (fixpoint) semantics \( S \) of \( R \), then for each proof \( e \rightarrow e_1 \rightarrow e_2 \ldots e_n \rightarrow (c = c) \) of \( e \) in \( R \), \( e_i \) belongs to
of this (obviously) continuous operator, i.e. \( F_{\text{Sem}} \) has the property as abstract domain \( \mathbb{A} \) where

\[
\{2\}
\]

**Proposition 3.8 (Correctness)** Let \( \mathcal{R} \) be a TRS and \( k > 0 \).

\[
\begin{align*}
S_i, i = 1, \ldots, n. \text{ That is, the semantics models all partial computations of equations in } \mathcal{R}.
\end{align*}
\]

### 3.2 Abstract semantics

Starting from the concrete fixpoint semantics of Definition 3.3, 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 \( \text{depth}(k) \) cut \([9]\), which allows to finitely approximate an infinite set of computed equations.

First of all we define a term abstraction as a function \( /k : (T(\Sigma, \mathcal{V}), \leq) \rightarrow (T(\Sigma, \mathcal{V} \cup \hat{\mathcal{V}}), \leq) \) 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 \( \mathcal{V} \)). \( \text{depth}(k) \) 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 \( \text{depth}(k) \) terms \((T(\Sigma, \mathcal{V} \cup \hat{\mathcal{V}}))/k\). 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

\[
\begin{align*}
\kappa(E) &:= \{s/k = t/k \mid s = t \in E\} \\
\gamma(A) &:= \{s = t \mid s/k = t/k \in A\}
\end{align*}
\]

Now we can derive the optimal abstract version of \( T_\mathcal{R} \) simply as \( T_\mathcal{R}^\kappa := \kappa \circ T_\mathcal{R} \circ \gamma \) and define the abstract semantics of program \( \mathcal{R} \) as the least fixpoint of this (obviously) continuous operator, i.e. \( \mathcal{F}_{\text{val}}^\kappa(\mathcal{R}) := T_\mathcal{R}^\kappa \uparrow \omega \). 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 \( T_\mathcal{R}^\kappa \uparrow \omega = T_\mathcal{R}^\kappa \uparrow h \). Abstract interpretation theory assures that \( T_\mathcal{R}^\kappa \uparrow \omega \) is the best correct approximation of \( \text{Sem}_{\text{val}}(\mathcal{R}) \). Correct means \( \mathcal{F}_{\text{val}}^\kappa(\mathcal{R}) \leq_S \kappa(\text{Sem}_{\text{val}}(\mathcal{R})) \) 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 3.6** \([2]\) For \( k > 0 \), the operator \( T_\mathcal{R}^\kappa : T/k \times T/k \rightarrow T/k \times T/k \) has the property \( T_\mathcal{R}^\kappa(X) \leq_S T_\mathcal{R}^\kappa(X) \) w.r.t. the following operator:

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

where \( B = \{t = t \mid t \in T(\mathcal{C})\} \).

**Definition 3.7** \([2]\) The abstract least fixpoint semantics of a program \( \mathcal{R} \) is defined as \( \tilde{\mathcal{F}}_{\text{val}}^\kappa(\mathcal{R}) = \tilde{T_\mathcal{R}^\kappa} \uparrow \omega \).

**Proposition 3.8 (Correctness)** \([2]\) Let \( \mathcal{R} \) be a TRS and \( k > 0 \).
(i) $\tilde{F}_\text{val}^\kappa(R) \leq_S \kappa(F_\text{val}(R)) \leq_S F_\text{val}(R)$.
(ii) For every $e \in \tilde{F}_\text{val}^\kappa(R)$ such that $\text{Var}(e) \cap \hat{V} = \emptyset$, $e \in F_\text{val}(R)$.

**Example 3.9** Consider again the specification in Example 3.4. Its abstract least fixpoint semantics for $\kappa = 3$ becomes

$$\tilde{F}_\text{val}^3(I) = \{\text{prize} = \text{prize}, \text{sorry-no-prize} = \text{sorry-no-prize},
\quad \text{true} = \text{true}, \text{false} = \text{false}, 0 = 0 \} \cup
\{s^i(0) = s^i(0) \mid i = 1, \ldots 3\} \cup \{s^4(\hat{x}) = s^4(\hat{x})\} \cup
\{\text{odd}(0) = \text{false}, \text{odd}(s(0)) = \text{true}, \text{odd}(s^2(0)) = \text{false},
\quad \text{odd}(s^3(\hat{x})) = \text{true}, \text{odd}(s^3(\hat{x})) = \text{false}\} \cup
\{\text{win}(\text{true}) = \text{prize}, \text{win}(\text{false}) = \text{sorry-no-prize},
\quad \text{win}(\text{odd}(0)) = \text{sorry-no-prize}, \text{win}(\text{odd}(s(0))) = \text{prize},
\quad \text{win}(\text{odd}(s^2(\hat{x}))) = \text{sorry-no-prize},
\quad \text{win}(\text{odd}(s^2(\hat{x}))) = \text{prize}\} \cup
\{\text{playdice}(0) = \text{sorry-no-prize},
\quad \text{playdice}(s(0)) = \text{prize},
\quad \text{playdice}(s^2(\hat{x})) = \text{sorry-no-prize},
\quad \text{playdice}(s^2(\hat{x})) = \text{prize}\}$$

**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) equations 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 [6]. More formally, we can state the correction problem as follows.

**4.1 Problem formalization**

Let $R$ be a TRS, $I$ be a specification of the intended semantics of $R$, $E^+$ and $E^-$ be two finite sets of equations such that

(i) $E^+ \subseteq \text{Sem}_\text{val}(I)$;
(ii) $E^- \subseteq (\text{Sem}_\text{val}(R) \setminus \text{Sem}_\text{val}(I)) \neq \emptyset$.

The correction problem consists in constructing a TRS $R^c$ satisfying the following requirements.
Equations in \( E^+ \) (resp. \( E^- \)) are called positive (resp. negative) examples. The TRS \( R^c \) is called correct program w.r.t. \( E^+ \) and \( E^- \). Note that, by construction, positive and negative example sets are disjoint, which permits to drive the correction process towards a discrimination between \( E^+ \) and \( E^- \).

### 4.2 Deductive and Inductive Learners

The automatic search for a new rule in an induction process can be performed either bottom-up (i.e. from an overly specific rule to a more general) or top-down (i.e. from an overly general rule to a more specific). There are some reasons to prefer the top-down or backward reasoning process to the bottom-up or forward reasoning process [11]. On the one hand, it eliminates the need for navigating through all possible logical consequences of the program. On the other hand, it integrates inductive reasoning with the deductive process, so that the derived program is guaranteed to be correct. Unfortunately, it is known that the deductive process alone (i.e. unfolding) does not generally suffice for coming up with the corrected program (unless the program is “overly general”, i.e. it covers all the positive examples) and inductive generalization techniques are necessary [11,25,26].

In [1], we presented a general bottom-up inductive generalization methodology along with a top-down inductive correction method. The former transforms a given program into a program which is “overly general” w.r.t. an example set, so that the latter can be applied. Therefore, in the sequel we only formalize an efficient top-down correction methodology, which we prove to be correct for a wider class of TRSs than the method which could be naïvely obtained by particularizing [1] to the functional setting.

### 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 \( \text{Sem}_{val}(I) \), since all equations in \( \tilde{F}_{val}(I) \) not containing variables in \( \check{V} \) belong to the concrete evaluation semantics \( \text{Sem}_{val}(I) \), as stated in the following lemma.

**Lemma 5.1** Let \( I \) be a TRS and \( E_P := \{ e | e \in \tilde{F}_{val}(I) \land \text{Var}(e) \cap \check{V} = \emptyset \} \). Then, \( E_P \subseteq \text{Sem}_{val}(I) \).
Now, by exploiting the information in $\tilde{\mathcal{F}}_{\text{val}}(\mathcal{I})$ and $\tilde{\mathcal{F}}_{\text{val}}(\mathcal{R})$, we can also generate a set of “negative” equations which belong to the concrete evaluation semantics $\text{Sem}_{\text{val}}(\mathcal{R})$ of the wrong program $\mathcal{R}$ but not to the concrete evaluation semantics $\text{Sem}_{\text{val}}(\mathcal{I})$ of the specification $\mathcal{I}$.

**Lemma 5.2** Let $\mathcal{R}$ be a TRS, $\mathcal{I}$ be a specification of the intended semantics and $E_N := \{ e | e \in \tilde{\mathcal{F}}_{\text{val}}(\mathcal{R}) \land \text{Var}(e) \cap \mathcal{V} = \emptyset \land \tilde{\mathcal{F}}_{\text{val}}(\mathcal{I}) \not\leq_S \{ e \} \}$. Then, $E_N \subseteq (\text{Sem}_{\text{val}}(\mathcal{R}) \setminus \text{Sem}_{\text{val}}(\mathcal{I}))$.

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. By considering these “data” examples, the inductive process becomes independent of 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.

$$E^+ = \{ l = c \mid f(t_1, \ldots, t_n) = c \in E_P \land f(t_1, \ldots, t_n) \equiv l \text{ is a pattern } \land \land c \in T(C) \land f \in \Sigma_{\mathcal{R}} \}$$

$$E^- = \{ l = c \mid l = c \in E_N \land l \text{ is a pattern } \land c \in T(C) \}$$

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 $\text{ExGen}(\mathcal{R}, \mathcal{I})$.

# 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 [6], 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, $E^+ \subseteq \text{Sem}_{\text{val}}(\mathcal{R})$). 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 6.1** Let $\mathcal{R}$ be a TRS and $E^+$ be a set of positive examples. If,
for each $e \in E^+$, there exists $e' \in \overline{F}_\text{val}(R)$ s.t.

(i) $e' \leq e$;
(ii) $\text{Var}(e') \cap \mathcal{V} = \emptyset$;

then, $R$ is overly general w.r.t. $E^+$.

Now, by exploiting Proposition 6.1, it is not difficult to figure out a procedure $\text{OVERLYGENERAL}(R,E)$ testing this condition w.r.t. a program $R$ and a set of examples $E$, e.g. a boolean function returning true if program $R$ is overly general w.r.t. $E$ and false otherwise.

6.1 The unfolding operator

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

Definition 6.2 Given two rules $r_1 \equiv \lambda_1 \rightarrow \rho_1$ and $r_2$, we define the rule unfolding of $r_1$ w.r.t. $r_2$ as $U_{r_2}(r_1) = \{ \lambda_1 \sigma \rightarrow \rho' \mid \rho_1 \sigma \Downarrow_{r_2} \rho \}$.

Definition 6.3 Given a TRS $R$ and a rule $r \ll R$, we define the program unfolding of $r$ w.r.t. $R$ as follows $U_R(r) = (R \cup \bigcup_{r' \in R} U_r(r)) \setminus \{r\}$.

Note that, by Definition 6.3, for any TRS $R$ and rule $r \ll R$, $r$ is never in $U_R(r)$.

Definition 6.4 Let $R$ be a TRS, $r$ be a rule in $R$. The rule $r$ is unfoldable w.r.t. $R$ if $U_R(r) \neq R \setminus \{r\}$.

Now, we are ready to prove that the “transformed” semantics, obtained after applying the unfolding operator to a given program $R$, still contains the semantics of $R$. In symbols, $\text{Sem}_\text{val}(R) \subseteq \text{Sem}_\text{val}(U_R(r))$, where $r$ is an unfoldable rule. We call this property unfolding correctness.

The following definition is auxiliary.

Definition 6.5 Let $t \in \tau(\Sigma \cup \mathcal{V})$ and $\perp$ be a symbol not in $\Sigma$. The shell of $t$, in symbols $\text{shell}(t)$, is defined as follows

$$\text{shell}(t) = \begin{cases} f(\text{shell}(t_1), \ldots, \text{shell}(t_n)) & \text{if } t \equiv f(t_1, \ldots, t_n), \text{ where } f \in \mathcal{D} \\ \perp & \text{otherwise} \end{cases}$$

A program $R$ is well-framed, if for each $\lambda \rightarrow \rho \in R$, $O_D(\text{shell}(\rho)) = O_D(\rho)$. The following theorem establishes the correctness of unfolding even for non-confluent programs, provided that they are well-framed. Note that well-framedness is much less demanding and easy to check.

Theorem 6.6 (unfolding correctness) Let $R$ be a well-framed left-linear CS, $r \ll R$ be an unfoldable rule and $R' = U_R(r)$. 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 $e \in \text{Sem}_\text{val}(R)$, then $e \in \ldots$

115
Finally, some important program properties such as well-framedness and program termination are preserved through unfolding.

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 6.3. 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 6.6). 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 $I$, also expressed as a program. First, TDCorrector computes the example sets $E^+$ and $E^-$ by means of ExGen, following the method presented in Section 5. Then, it checks whether program $\mathcal{R}$ is overly general following the scheme of Proposition 6.1, and finally it enters the main correction process.

In order to ensure correctness of the algorithm, we require $k$ to be greater than or equal to the maximum depth of the terms occurring in $E^+$. 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 easy to check, 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 ($\mu$-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 by using a fair selection strategy. More precisely, we non-deterministically choose rules which appear first in proofs of positive examples. Those rules can be easily computed retrieving them from the fixpoint semantics $\tilde{\mathcal{T}}^*_{\text{val}}$ by a further step of the $\tilde{\mathcal{T}}^*_\text{val}$ operator by means of the auxiliary function first, which
procedure TDCorrector($R, I$)
\[ (E^+, E^-) \leftarrow \text{ExGen}(R, I) \]

if not OverlyGeneral($R, E^+$) then Halt

\[ k \leftarrow 0; R_k \leftarrow R \]

while \( \exists e^- \in E^- : \tilde{F}_e^k(R_k) \leq_S \{ e^- \} \) do

\[ R \leftarrow \{ r \in R_k \mid r \text{ is unfoldable} \land r \in \text{first}(E^+) \} \]

if \( R \neq \emptyset \) then

\[ \text{Select}(r, R) \]

\[ R_{k+1} \leftarrow U_{R_k}(r); k \leftarrow k + 1 \]

end if

for each \( r \in R_k \) do

if OverlyGeneral($R_k \{ r \}, E^+$) then

\[ R_k \leftarrow R_k \{ r \} \]

end if

end for

end while

\[ R_c \leftarrow R_k \]
end procedure

Fig. 1. The top-down correction algorithm.

is formally defined as follows.

**Definition 6.7** Let $R$ be a TRS and $E$ be an example set. Then, we define

\[ \text{first}(E) := \bigcup_{e \in E} \{ r \mid e \in \tilde{T}_e^r(\tilde{F}_e^R) \}. \]

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 $R_k \{ r \}$ 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 an example.

**Example 6.8** The following OBJ program $R$ is wrong w.r.t. the specification $I$ of Example 3.4.

\begin{verbatim}
obj GAME is
sorts Nat Reward .
op o : -> Nat .
op s : Nat -> Nat .
op prize : -> Reward .
op sorry-no-prize : -> Reward .
op playdice : Nat -> Reward .
var X : Nat .
eq playdice(s(X)) = playdice(X) . (1)
eq playdice(0) = prize . (2)
eq playdice(0) = sorry-no-prize . (3)
\end{verbatim}
Note that program \( R \) is non-confluent and computes both values \( \text{prize} \) and \( \text{sorry-no-prize} \) for any natural \( s^n(0), n > 0 \). By fixing \( \kappa = 3 \), we compute the following least fixpoint abstract semantics for \( R \).

\[
\tilde{F}^3_{\text{val}}(R) = \{ \text{prize} = \text{prize}, \text{sorry-no-prize} = \text{sorry-no-prize},
\text{true} = \text{true}, \text{false} = \text{false}, 0 = 0 \} \cup
\{ s^i(0) = s^i(0) \mid i = 1, \ldots, 3 \} \cup \{ s^4(\hat{x}) = s^4(\hat{x}) \} \cup
\{ \text{playdice}(0) = \text{prize}, \text{playdice}(0) = \text{sorry-no-prize},
\text{playdice}(s(0)) = \text{prize}, \text{playdice}(s(0)) = \text{sorry-no-prize},
\text{playdice}(s^2(0)) = \text{prize}, \text{playdice}(s^2(0)) = \text{sorry-no-prize},
\text{playdice}(s^3(\hat{x})) = \text{prize}, \text{playdice}(s^3(\hat{x})) = \text{sorry-no-prize} \}.
\]

Considering the abstract fixpoint semantics \( \tilde{F}^3_{\text{val}}(I) \) computed in Example 3.9 and following the methodology of Section 5, we obtain the example sets below:

\[
E^+ = \{ \text{playdice}(s(0)) = \text{prize}, \text{playdice}(0) = \text{sorry-no-prize} \}
\]

\[
E^- = \{ \text{playdice}(s(0)) = \text{sorry-no-prize}, \text{playdice}(0) = \text{prize} \}.
\]

Now, since program \( R \) fulfills the condition for overgenerality expressed by Proposition 6.1, the algorithm proceeds and enters the main loop. Here, program rule (1) is unfolded, because (1) is unfoldable and belongs to \( \text{first}(E^+) \). So, the transformed program is

\[
\begin{align*}
\text{eq playdice}(s(s(X))) &= \text{playdice}(X). \quad (4) \\
\text{eq playdice}(s(0)) &= \text{prize}. \quad (5) \\
\text{eq playdice}(s(0)) &= \text{sorry-no-prize}. \quad (6) \\
\text{eq playdice}(0) &= \text{prize}. \quad (7) \\
\text{eq playdice}(0) &= \text{sorry-no-prize}. \quad (8)
\end{align*}
\]

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 (6), (7) are not necessary, and therefore are removed producing the correct program which consists of rules (4), (5) and (8).

Note that the above example cannot be repaired by using the correction method of [1].

### 6.3 Correctness of algorithm TDCorrector

In this section, we prove the correctness of the top-down correction algorithm TDCorrector, i.e., we show that it produces a specialized version of \( R \) which is a correct program w.r.t. \( E^+ \) and \( E^- \), provided that \( R \) is overly general w.r.t. \( E^+ \). A condition is necessary for establishing this result: no negative/positive
couple of the considered examples must be proven by using the same sequence of rules.

Let us start by giving an auxiliary definition and some technical lemmata.

**Definition 6.9** Let $\mathcal{R}$ be a TRS and $E$ be a set of examples. The unfolding succession $\mathcal{US}(\mathcal{R}) \equiv \mathcal{R}_0, \mathcal{R}_1, \ldots$ of program $\mathcal{R}$ w.r.t. $E$ is defined as follows:

$$\mathcal{R}_0 = \mathcal{R}, \quad \mathcal{R}_{i+1} = \begin{cases} \mathcal{U}_\mathcal{R}(r) & \text{where } r \in \mathcal{R}_i \text{ is unfoldable and } r \in \text{first}(E) \\ \mathcal{R}_i & \text{otherwise} \end{cases}$$

The next results state that we are always able to transform a program $\mathcal{R}$ into a program $\mathcal{R}'$ by a suitable number of unfolding steps, in such a way that any given proof of an example in $\mathcal{R}$ can be mimicked by a one-step proof in $\mathcal{R}'$. In the following we denote the length of a rewrite sequence $\mathcal{S}$ by $|\mathcal{S}|$.

**Lemma 6.10** Let $\mathcal{R}$ be a well-framed left-linear CS, $E$ be a set of examples and $r \in \text{first}(E)$ be an unfoldable rule such that $\mathcal{R}' = \mathcal{U}_\mathcal{R}(r)$. Let $t = c \in E$, where $t$ is a pattern and $c$ is a value. Then,

(i) if $\mathcal{S}$ is a rewrite sequence from $t$ to $c$ in $\mathcal{R}$, then there exists a rewrite sequence $\mathcal{S}'$ from $t$ to $c$ in $\mathcal{R}'$;

(ii) if $r$ occurs in $\mathcal{S}$, then $|\mathcal{S}'| < |\mathcal{S}|$.

**Lemma 6.11** Let $\mathcal{R}$ be a well-framed left-linear CS, $E$ be an example set and $t = c \in E$, where $t$ is a pattern and $c$ is a value. Let $t \rightarrow r_1 \ldots \rightarrow r_n c$, $n \geq 1$. Then, for each unfolding succession $\mathcal{US}(\mathcal{R})$ w.r.t. $E$, there exists $\mathcal{R}_k$ occurring in $\mathcal{US}(\mathcal{R})$ such that $t \rightarrow r^* c$, $r^* \in \mathcal{R}_k$.

The following result immediately derives from Claim (i) of Proposition 3.8.

**Lemma 6.12** Let $\mathcal{R}$ be a TRS and $e$ an equation. Then, if $\tilde{\mathcal{F}}_{\text{val}}(\mathcal{R}) \not\preceq_S \{e\}$, then $e \not\in \text{Sem}_{\text{val}}(\mathcal{R})$.

The following definition is auxiliary. We say that the pair of positive an negative example sets ($E^+, E^-$) is discriminable in $\mathcal{R}$ [6], if there are no $e^+ \in E^+$ and $e^- \in E^-$ which can be proven by using the same sequence of rules of $\mathcal{R}$. This property can be checked by using standard tools for proving program termination.

Now we are ready to prove the partial correctness and the termination of the algorithm.

**Theorem 6.13** Let $\mathcal{R}$ be a well-framed left-linear CS and $\mathcal{I}$ be a specification of the intended semantics of $\mathcal{R}$. Let $E^+$ and $E^-$ be the example sets generated by $\text{EXGEN}(\mathcal{R}, \mathcal{I})$.

(i) If ($E^+, E^-$) is discriminable in $\mathcal{R}$, then the algorithm $\text{TDCorrector}(\mathcal{R}, \mathcal{I})$ terminates.

(ii) If $\mathcal{R}$ is overly general w.r.t. $E^+$ and $\text{TDCorrector}(\mathcal{R}, \mathcal{I})$ terminates,
then the computed program \( \mathcal{R}^c \) is correct w.r.t. \( E^+ \) and \( E^- \).

We note that well-framedness can be dispensed in exchange for requiring complete definedness (CD), i.e. the property that functions are defined on all possible values of their arguments, as in [4] by simply delaying the deletion phase and performing a virtual deletion instead. This suffices to ensure that the CD property is preserved during the transformation.

7 Conclusions

In this paper, we have proposed a 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. For example, in QuickCheck [8], formal specifications are used to describe properties of Haskell programs (written as Haskell programs too) which are automatically tested.

Our methodology is based on a combination, in a single framework, of a diagnoser which identifies those parts of the code containing errors, together with a deductive program learner which, once the bug has been located in the program, tries to repair it starting from evidence examples (uncovered as well as incorrect equations) which are essentially obtained as an outcome of the diagnoser.

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 well-framed non-confluent programs, while [1] describes a methodology which only corrects inductively sequential (respectively, canonical and completely defined) programs according to the chosen lazy (respectively, eager) narrowing strategy. 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 the programmer’s intention).

We are currently extending the implementation of the diagnosis system Debussy [2] (available at \texttt{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


