Abstract Diagnosis of First Order Functional Logic Programs

Giovanni Bacci and Marco Comini

Dipartimento di Matematica e Informatica
University of Udine

Abstract. We present a generic scheme for the abstract debugging of functional logic programs. We associate to programs a semantics based on a (continuous) immediate consequence operator, $\mathcal{P}[\![\mathcal{R}]\!]$, which models correctly the powerful features of modern functional logic languages (non-deterministic, non-strict functions defined by non-confluent programs and call-time choice behaviour). 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 provide error symptoms in advance and is applicable with partial specifications and even partial programs.

1 Introduction

Finding program bugs is a long-standing problem in software development, even for highly expressive declarative functional logic languages because of laziness. There has been a lot of work on debugging for functional logic languages, amongst all [1,9,6,7,17,8], mostly following the declarative debugging approach.

Declarative debugging 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 questions on correctness of solutions to 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 (as noted also by [1,18]).

Abstract diagnosis for Logic Programs [12,11,13] is a framework parametric w.r.t. an abstract program property which can be considered as an extension of declarative debugging since there are instances of the framework that deliver the same results. It is based on the use of an immediate consequence operator T_P to identify bugs in logic programs. 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 logic programs using the ideas of [12]. Since this technique is *inherently* based on the use of an immediate consequence operator, the (top-down) operational semantics for functional logic languages in the literature are not suited for this purpose. Thus we first introduce a (continuous) immediate consequence operator $\mathcal{P}[\![\mathcal{R}]\!]$

M. Alpuente (Ed.): LOPSTR 2010, LNCS 6564, pp. 215–233, 2011.

to a given functional logic program $\mathcal{R}$ which allows us to derive the concrete semantics for $\mathcal{R}$. Then, we formulate an efficacious debugging methodology based on abstract interpretation which proceeds by approximating the $\mathcal{P}[\![\mathcal{R}]\!]$ operator producing an "abstract immediate consequence operator" $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$. We show that, given the abstract intended specification $\mathcal{S}^\alpha$ of the semantics of a program $\mathcal{R}$, we can check the correctness of $\mathcal{R}$ by a single application of $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$ and thus, by a simple static test, we can determine all the rules which are wrong w.r.t. the considered abstract property.

The diagnosis is based on the detection of *incorrect rules* and *uncovered elements*, which both have been defined in terms of one application of $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$ to the abstract specification. It is worth noting that no fixpoint computation is required, since the abstract semantics does not need to be computed. The key issue of this approach is the goal-independence of the concrete semantics, meaning that the semantics is defined by collecting the observable properties about "most general" calls, while still providing a complete characterization of the program behavior.

Among other valuable facilities, this debugging approach supports the development of efficacious diagnostic tools that detect program errors without having to determine symptoms in advance. By using suitable abstract domains several details of the computation can be hidden and thus the information that is required to the user about the (abstract) intended behaviour can be dramatically reduced. Obviously if we use more abstract domains we can detect less errors: for example an erroneous integer value cannot be detected by looking at groundness information. The choice of an abstract domain is thus a tradeoff between the precision of errors that can be detected and the effort in providing the specification.

2 Notations and Assumptions

Let us briefly introduce the notations used in the paper. Since we are interested in modern first-order functional logic programs we will use the formalism of [15] because the class of GRSs that they consider is a good fit for our programs. In the following a program is a special case of Graph Rewriting System $(\Sigma, \mathcal{R})$, where the signature is partitioned into two disjoint sets $\Sigma := \mathcal{C} \uplus \mathcal{D}$, where term graphs (simply terms) are just labelled DAGs (instead of general admissible term graphs), without multiple occurrences of the same variable and where the left hand side of program rules $l \to r$ is a pattern.

A term π is a *pattern* if it is of the form $f(\overrightarrow{c_n})$ where $f/n \in \mathcal{D}$ and $\overrightarrow{c_n}$ is a n-tuple of constructor terms (not sharing subterms). $\overrightarrow{t_n}$ denotes a generic n-tuple of terms $t_1, \ldots, t_n$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions*. $\mathcal{T}(\Sigma, \mathcal{V})$ denotes the terms built over signature Σ and variables $\mathcal{V}$, $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are called constructor terms. $\mathcal{N}_g$ denotes the nodes of a term g. Each node in $\mathcal{N}_g$ is labelled with a symbol in Σ or a variable in $\mathcal{V}$. $\mathcal{L}_g \colon \mathcal{N}_g \to \Sigma \cup \mathcal{V}$, $\mathcal{S}_g$ and $\mathcal{R}oot_g$ denote the labeling, successor and root functions, respectively. $g_1 \oplus g_2$ denotes the sum of two terms. $g|_p$ is the subterm

at the position p of g. $g[s]_p$ denotes the replacement by a term s of the subterm rooted by a node p in a term g.

Throughout this paper, $\mathcal{V}$ denotes a (fixed) countably infinite set of variables and $\mathcal{N}$ a (fixed) countable set of nodes.

By $V(s)$ we denote the set of variables occurring in the syntactic object s. A *fresh* variable is a variable that appears nowhere else. $s \ll S$ indicates that the syntactic object s is a fresh variant of an object in S.

$\theta{\restriction_s}$ denotes the restriction of the substitution θ to the set of variables in the syntactic object s. The *empty substitution* is denoted by ε. By $h\colon g_1 \to g_2$ we denote a (rooted) homomorphism h from a term g_1 to a term g_2. If $h\colon g_1 \to g_2$ is an homomorphism and g is a term, $h[g]$ is the term built from g by replacing all the subterms shared between g and g_1 by their corresponding subterms in g_2. In the rest of the paper, with an abuse of notation, we use the same name to denote an homomorphism $h\colon g_1 \to g_2$ and its extension h' on g that preserves the nodes not belonging to $\mathcal{N}_{g_1}$, i.e., $h'(n) = n$ if $n \notin \mathcal{N}_{g_1}$, $h'(n) = h(n)$ otherwise. Given a homomorphism $h\colon g_1 \to g_2$, we indicate with σ_h the substitution such that for every $x \in V(g_1)$ which labels some node $p \in \mathcal{N}_{g_1}$, $x\sigma_h = g_2|_{h(p)}$.

$mgu(t, s)$ denotes "the" *most general unifier* homomorphism $h\colon t \oplus s \to g$ of terms t and s. By abuse of notation it denotes also its induced substitution. We will also denote by $mgu(\sigma_1, \ldots, \sigma_n)$ the most general unifier of substitutions $\sigma_1, \ldots, \sigma_n$ [19]. $t_0 \overset{\sigma}{\underset{\mathcal{R}}{\leadsto}}{}^* t_n$ denotes a *needed* narrowing derivation (see [15]) $t_0 \overset{\sigma_1}{\underset{\mathcal{R}}{\leadsto}} \ldots \overset{\sigma_n}{\underset{\mathcal{R}}{\leadsto}} t_n$ such that $\sigma = (\sigma_1 \ldots \sigma_n){\restriction_t}$. If $t \overset{\sigma}{\underset{\mathcal{R}}{\leadsto}}{}^* s$ the pair $\sigma \cdot s$ is said a *partial answer* for t and when $s \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ we call it a *computed answer*.

3 The Semantic Framework

In this section we introduce the concrete semantics of our framework, which is suitable to model correctly the typical cogent features of (the first-order fragment of) modern functional logic languages:

- non-deterministic, non-strict functions defined by non-confluent programs;
- the call-time choice behaviour (where the values of the arguments of an operation are determined before the operation is evaluated)

Actually in [4] we have obtained this semantics by optimal abstraction of a (much more) concrete semantics modeling needed narrowing trees, thus its correctness w.r.t. needed narrowing comes by construction (and standard abstract interpretation results [14]).

3.1 Incremental Answer Semantics

To deal with non-strict operations, as done by [8], we consider signatures $\Sigma_\perp$ that are extended by a special constructor symbol $\perp$ to represent undefined values, e.g. $S(S(\perp))$ denotes a number greater than 1 where the exact value is undefined. Such partial terms are considered as finite approximations of possibly infinite values. We denote by $\mathcal{T}(\mathcal{C}_\perp, \mathcal{V})$ the set of *partial constructor terms*.

Our fixpoint semantics is based on interpretations that consist of families of Incremental Answer Trees for the "most general calls" of a program. Intuitively an Incremental Answer Tree collects the "relevant history" of the computation of all computed answers of a goal, abstracting from function calls and focusing only on the way in which the answer is (incrementally) constructed.

Definition 1 (Incremental Answer Trees). *Given a term t over signature $\Sigma = \mathcal{C} \uplus \mathcal{D}$, an* incremental answer tree *for t is a tree of partial answers T s.t.*

1. *its root is $\varepsilon \cdot \tau_\bot(t)$, where $\tau_\bot(t)$ denotes the partial constructor term obtained by replacing all $\mathcal{D}$-rooted subterms of t with the $\bot$ symbol,*
2. *for any partial answer $\sigma \cdot s$ that is a parent of $\sigma' \cdot s'$ there exists a constructor substitution θ, a position p and $t \in \mathcal{T}(\mathcal{C}_\bot, \mathcal{V}) \setminus \{\bot\}$ such that $\sigma' = \sigma\theta$, $s|_p = \bot$ and $s' = (s\theta)[t]_p$,*
3. *there is no pair a, b of sibling nodes such that a and b are variants and*
4. *for any pair a, b of sibling nodes if a has a son (labelled) c which could be a son of b (in the sense of Item 2) then b has also a son (labelled) c.*

Given two incremental answer trees T and T' for t, $T \sqsubseteq T'$ if and only if every path in T starting from the root, is also a path of T'.

The semantic domain $\mathbb{T}$ is the set of all incremental answer trees ordered by $\sqsubseteq$. $\mathbb{T}$ is a complete lattice. We denote its bottom $\varepsilon \cdot \bot$ by $\bot_{\mathbb{T}}$.

The intuition is that for any parent node we choose an occurrence of a $\bot$ and we evaluate it until at least another constructor is added. All narrowing steps that do not introduce a constructor are collapsed together (see Example 1).

Item 3 ensures that there is no duplicated information (which will have to be collapsed by our semantics constructions).

Item 4 ensures that a certain evaluation performed on a subterm in a certain node is also performed in all other occurrences of the same subterm in all other nodes.

The idea behind our proposal is that of building a family of incremental answer trees "modulo variance", one tree for each most general call $f(x_1, \ldots, x_n)$ (which are a finite number). We quotient w.r.t. variance because the actual choice of variables names in $f(x_1, \ldots, x_n)$ has to be irrelevant. For notational convenience we represent this family as a function in the following way.

Definition 2 (Interpretations). *Let $\mathbb{MGC} := \{f(\overrightarrow{x_n}) \mid f \in \mathcal{D}, \overrightarrow{x_n}$ are distinct variables$\}$. An* interpretation *$\mathcal{I}$ is a function $\mathbb{MGC} \to \mathbb{T}$ modulo variance such that, for every $\pi \in \mathbb{MGC}$, $\mathcal{I}(\pi)$ is an incremental answer tree for π. Two functions I, J are* variants *if for each $\pi \in \mathbb{MGC}$ there exists an isomorphism ι (i.e., a renaming of variables and node names) such that $I(\iota[\pi]) = \iota[J(\pi)]$.*

The semantic domain $\mathbb{I}_\Sigma$ is the set of all interpretations ordered by the pointwise extension (modulo variance) of $\sqsubseteq$. $\mathbb{I}_\Sigma$ is a complete lattice. Its bottom is the constant function $\lambda f(\overrightarrow{x_n}).\bot_{\mathbb{T}}$.

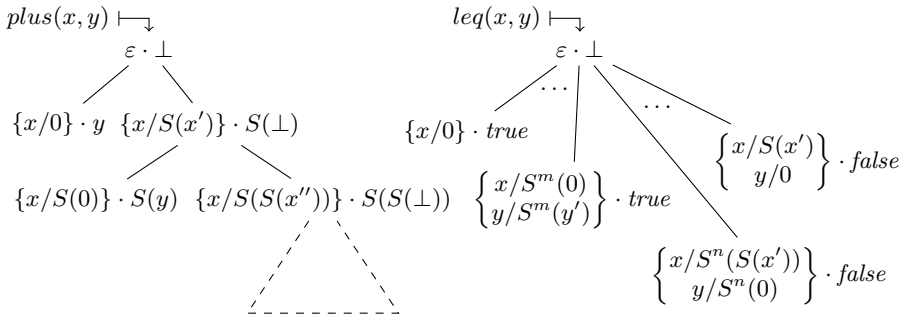


Fig. 1. Semantics for *plus* and *leq*

An interpretation $\mathcal{I}$ is implicitly extended to constructors as $\mathcal{I}(c(\overrightarrow{x_n})) := \varepsilon \cdot c(\overrightarrow{x_n})$[1].

Example 1. The incremental answer trees depicted in Figure 1 denote the semantics of the program

```
plus 0   y = y                    leq 0 _ = True
plus (S x) y = S (plus x y)       leq (S x) 0 = False
                                  leq (S x) (S y) = leq x y
```

Notice that both trees are infinite in different ways. The one associated to $plus(x, y)$ has infinite height, since *plus* builds the solution one S at a time. The one associated to $leq(x, y)$ has infinite width, since *leq* delivers just value true or false. □

While trees are easy to understand and to manage in implementations, they are quite big to show in a paper and technical definitions tends to be more cluttered. Thus, for the sake of compactness and comprehension, we prefer to use in the following an isomorphic representation where we collect in a set all the nodes of a tree but annotating with • the new root constructors introduced by expanding a $\bot$ in a parent node. This annotation conveys all the needed information about inner tree nodes. We call these sets *annotated-sets*.

For example, the annotated-set representation of the trees of Figure 1 is

$$\begin{cases} plus(x,y) \mapsto \{\, \varepsilon \cdot \bot, \{x/0\} \cdot \dot{y}, \{x/S(x')\} \cdot \dot{S}(\bot), \\ \qquad\qquad \{x/S(0)\} \cdot \dot{S}(\dot{y}), \{x/S(S(x'))\} \cdot \dot{S}(\dot{S}(\bot)), \ldots \} \\ leq(x,y) \mapsto \{\, \varepsilon \cdot \bot,\ \{x/0\} \cdot \dot{true}, \ldots, \{x/S^i(0), y/S^i(y')\} \cdot \dot{true}, \ldots \\ \qquad\qquad \{x/S(x'), y/0\} \cdot \dot{false}, \ldots, \{x/S^{i+1}(x'), y/S^i(0)\} \cdot \dot{false}, \ldots \} \end{cases}$$

[1] The evaluation of a constructor symbol can be considered as a degenerate case of a defined symbol that is just interpreted as itself. This technical assumption is useful in the following to have much compact definitions.

Solutions for *plus* have $\dot{S}$ everywhere, because S are introduced one at a time. For a rule like $doub(x) \to S(S(doub(x)))$, where two S are introduced in each step, solutions are instead $\{\bot, \dot{S}(S(\bot)), \dot{S}(S(\dot{S}(S(\bot)))), \ldots\}$.

Note that any annotated partial answer in an annotated-set corresponds (at least) to a path from the root to a given node of an incremental answer tree.

An important ingredient of the definition of our semantics is the evaluation of concrete terms occurring in the body of a program rule w.r.t. a current interpretation as defined next.

Definition 3 (Evaluation of terms). *Given an interpretation $\mathcal{I}$ we define by syntactic induction the* semantic evaluation *of a term t in $\mathcal{I}$ as*

$$\mathcal{E}[\![x]\!]_{\mathcal{I}} := \{\varepsilon \cdot x\}$$

$$\mathcal{E}[\![f(\overrightarrow{t_n})]\!]_{\mathcal{I}} := \left\{ (\vartheta\eta_h)\restriction_{\overrightarrow{t_n}} \cdot h[r] \;\middle|\; \begin{array}{l} \sigma_i \cdot s_i \in \mathcal{E}[\![t_i]\!]_{\mathcal{I}} \text{ for } i = 1, \ldots, n \\ \vartheta = mgu(\sigma_1, \ldots, \sigma_n), \\ \mu \cdot r \ll \mathcal{I}(f(\overrightarrow{x_n})) \\ \exists h = \mathring{mgu}_{V(r)}(f(\overrightarrow{x_n})\mu, f(\overrightarrow{s_n})\vartheta) \end{array} \right\} \tag{3.1}$$

where $\mathring{mgu}_V(t, s)$, for an annotation-free pattern t, a term s with no node in common with t and a set V of variables, is defined as the mgu homomorphism h for t and s, $h\colon t \oplus s \to r$, such that the following conditions hold:

1. *every $p \in \mathcal{N}_r$ is annotated if and only if $h^{-1}(p)$ has an annotated node*
2. *for every $x \in V(t) \setminus V$ if p is an annotated node of $\eta_h(x)$, then*
   - *p is shared with $\eta_h(y)$ for some $y \in V$, or*
   - *there exists $q \in \mathcal{N}_t \cap h^{-1}(p)$ such that $\mathcal{L}_t(q) \in \mathcal{C}$*

$\mathring{mgu}_V(t, s)$ is a most general unifier on terms which takes into account annotations and a target set of variables $V(t) \setminus V$. A variable in the target cannot be bound with an annotated term t', *unless* t' is unified with another non-variable term. Intuitively, when we perform $\mathring{mgu}_V(t, s)$, variables not in the target $V(t) \setminus V$ are those which will be bound to outermost-needed terms in subsequent narrowing steps, thus we have to evaluate them. On the contrary, variables in the target should not be evaluated to preserve laziness.

Equation (3.1) is the core of all the evaluation: first it "mounts" together all possible contributions of (the evaluations of) subterms $\overrightarrow{t_n}$, ensuring that their local instantiations are compatible with each other, then it takes a fresh partial answer[2] $\mu \cdot r$ from $\mathcal{I}(f(\overrightarrow{x_n}))$ and performs unification respecting annotations by $\mathring{mgu}_{V(r)}(f(\overrightarrow{x_n})\mu, f(\overrightarrow{s_n})\vartheta)$. This ensures that only the evaluations of terms $\overrightarrow{t_n}$ which are necessary to evaluate $f(\overrightarrow{t_n})$ are used. In other words, (3.1) uses the partial answer $\mu \cdot r \in \mathcal{I}(f(\overrightarrow{x_n}))$ as "big-step" rule for f, namely $f(\overrightarrow{x_n})\mu \to r$. Moreover the existence of the homomorphism $h = \mathring{mgu}_{V(r)}(f(\overrightarrow{x_n})\mu, f(\overrightarrow{s_n}))$ ensures that any annotated term in the co-domain of η_h corresponds to a term which is root-needed for $f(\overrightarrow{t_n})$.

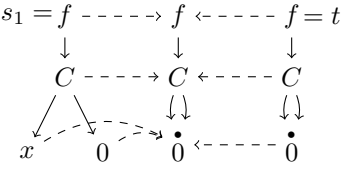
[2] Note that in the case the symbol f is a constructor c the partial answer is $\varepsilon \cdot c(\overrightarrow{x_n})$.

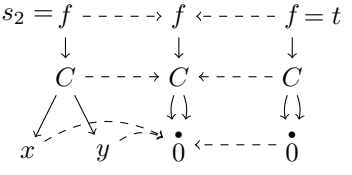
Example 2. Consider the following program and its interpretation $\mathcal{I}$:

```
coin = 0
coin = S 0
sub2 (S (S x)) = x
f (C x y) = 0
f (C x 0) = S x
pair(x) = C x x
```

$$I := \begin{cases} coin \mapsto \{\varepsilon \cdot \bot,\ \varepsilon \cdot \dot{0},\ \varepsilon \cdot \dot{S}(0)\} \\ sub2(x) \mapsto \{\varepsilon \cdot \bot,\ \{x/S(S(x'))\} \cdot \dot{x'}\} \\ f(z) \mapsto \{\varepsilon \cdot \bot,\ \{z/C(x,y)\} \cdot \dot{0}, \\ \qquad\qquad \{z/C(x,0)\} \cdot \dot{S}(x)\} \\ pair(x) \mapsto \{\varepsilon \cdot \bot,\ \varepsilon \cdot \dot{C}(x,x)\} \end{cases}$$

For the goal term $g := pair(sub2(s(coin)))$, we have $\varepsilon \cdot \dot{C}(\dot{0}, \dot{0}) \in \mathcal{E}[\![g]\!]_{\mathcal{I}}$. In order to evaluate $\mathcal{E}[\![f(g)]\!]_{\mathcal{I}}$ we have to decide if there exist $h_1 := m\mathring{g}u_{\{x\}}(s_1, t)$ and $h_2 := m\mathring{g}u_{\emptyset}(s_2, t)$ respectively, where





we see that h_1 exists and is equal to $\{x/\dot{0}\}$, while h_2 does not exists. Even if t and s_2 unify in the standard sense, x and y have to be bound to an annotated term, namely $\dot{0}$, but y is not in $V = \emptyset$. □

Now we can define our concrete semantics.

Definition 4 (Concrete Fixpoint Semantics). *Let $\mathcal{I}$ be an interpretation and $\mathcal{R}$ be a program. The immediate consequence operator is*

$$\mathcal{P}[\![\mathcal{R}]\!]_{\mathcal{I}} := \lambda f(\overrightarrow{x_n}).\ \{\varepsilon \cdot \bot\} \cup \left\{ (\{\overrightarrow{x_n}/\overrightarrow{t_n}\}\sigma)\restriction_{\overrightarrow{x_n}} \cdot \dot{s} \;\middle|\; \begin{array}{l} f(\overrightarrow{t_n}) \to r \ll \mathcal{R}, \\ \sigma \cdot s \in \mathcal{E}[\![r]\!]_{\mathcal{I}},\ s \neq \bot \end{array} \right\}$$

Since $\mathcal{P}[\![\mathcal{R}]\!]$ is continuous (as proved in [4]), we can define our fixpoint semantics as $\mathcal{F}[\![\mathcal{R}]\!] := lfp\ \mathcal{P}[\![\mathcal{R}]\!] = \mathcal{P}[\![\mathcal{R}]\!]\uparrow\omega$.

Example 3. Let $\mathcal{R}$ be the program

```
from n = n : from (S n)
take 0 _ = []
take (S n) (x:xs) = x : take n xs
m1 = take (S (S 0)) (from n) where n free
m2 = take (S (S 0)) (from coin)
```

The second iteration of the fixpoint computation $\mathcal{P}[\![\mathcal{R}]\!]\uparrow 2$ is

$$\begin{cases} \mathit{from}(n) \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot n{:}\bot,\ \varepsilon\cdot n{:}S(n){:}\bot\} \\ \mathit{coin} \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot 0,\ \varepsilon\cdot S(0)\} \\ \mathit{take}(n,xs) \mapsto \{\varepsilon\cdot\bot,\ \{n/0\}\cdot[\,],\ \{n/S(n'),xs/x'{:}xs'\}\cdot x'{:}\bot, \\ \qquad\qquad \{n/S(0),xs/x'{:}xs'\}\cdot x'{:}[\,], \\ \qquad\qquad \{n/S(S(n'')),xs/x'{:}x''{:}xs''\}\cdot x'{:}x''{:}\bot\} \\ \mathit{m1} \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot n{:}\bot\} \\ \mathit{m2} \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot\bot{:}\bot,\ \varepsilon\cdot 0{:}\bot,\ \varepsilon\cdot S(0){:}\bot)\} \end{cases}$$

while the resulting semantics $\mathcal{F}[\![\mathcal{R}]\!]$ is

$$\begin{cases} \mathit{from}(n) \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot n{:}\bot,\ \varepsilon\cdot n{:}S(n){:}\bot,\ \varepsilon\cdot n{:}S(n){:}S(S(n)){:}\bot,\dots\} \\ \mathit{coin} \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot 0,\ \varepsilon\cdot S(0)\} \\ \mathit{take}(n,xs) \mapsto \{\varepsilon\cdot\bot,\{n/0\}\cdot[\,],\ \{n/S(n'),xs/x'{:}xs'\}\cdot x'{:}\bot, \\ \qquad\qquad \{n/S(0),xs/x'{:}xs'\}\cdot x'{:}[\,], \\ \qquad\qquad \{n/S(S(n'')),xs/x'{:}x''{:}xs''\}\cdot x'{:}x''{:}\bot, \\ \qquad\qquad \{n/S(S(0)),xs/x'{:}x''{:}xs''\}\cdot x'{:}x''{:}[\,],\ \dots\} \\ \mathit{m1} \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot n{:}\bot,\ \varepsilon\cdot n{:}S(n){:}\bot,\ \varepsilon\cdot n{:}S(n){:}[\,]\} \\ \mathit{m2} \mapsto \{\varepsilon\cdot\bot,\ \varepsilon\cdot\bot{:}\bot,\ \varepsilon\cdot\bot{:}S(\bot){:}\bot,\ \varepsilon\cdot 0{:}\bot,\ \varepsilon\cdot 0{:}S(0){:}\bot, \\ \qquad \varepsilon\cdot 0{:}S(0){:}[\,],\ \varepsilon\cdot S(0){:}\bot,\ \varepsilon\cdot S(0){:}S(S(0)){:}\bot,\ \varepsilon\cdot S(0){:}S(S(0)){:}[\,]\} \end{cases}$$

It is worth noting that several fixpoint semantics proposed in the literature are defined on subclasses of programs (e.g. almost orthogonal, right linear, topmost, ...) while we require only left linearity and construction basedness (conditions that are satisfied by most functional logic languages). This is particularly important in view of application of the semantics in abstract diagnosis, where buggy programs cannot reasonably satisfy *a priori* any condition at all.

Our semantics is sound and complete w.r.t. needed narrowing in the sense of the following theorem.

Theorem 1 (Correctness and completeness [4]). *Let t be a term, and let $\mathcal{R}$ be a functional logic program. The following statements hold:*

1. *if $\sigma\cdot s_\bot \in \mathcal{E}[\![t]\!]_{\mathcal{F}[\![\mathcal{R}]\!]}$ then $\exists s \in \mathcal{T}(\Sigma,\mathcal{V})$ such that $t \overset{\sigma}{\underset{\mathcal{R}}{\leadsto}}{}^{*}\ s$ and $\tau_\bot(s) = s_\bot$*
2. *if $t \overset{\sigma}{\underset{\mathcal{R}}{\leadsto}}{}^{*}\ s$ then $\exists\vartheta \le \sigma$ and $s_\bot = \tau_\bot(s)\vartheta$ such that $\vartheta\cdot s_\bot \in \mathcal{E}[\![t]\!]_{\mathcal{F}[\![\mathcal{R}]\!]}$*

Each partial answer in the semantics corresponds to a term in a needed narrowing derivation and, viceversa, for each derivation there is a "more general" partial answer in the semantics.

3.2 Abstraction Scheme

In this section, starting from the fixpoint semantics in Section 3.1, we develop an abstract semantics which approximates the observable behavior of the program. Program properties which can be of interest are Galois Insertions between the concrete domain $\mathbb{T}$ and the abstract domain chosen to model the property. We assume familiarity with basic results of abstract interpretation [14].

We will focus our attention now on a special class of abstract interpretations which are obtained from what we call an *incremental answer abstraction* that is a Galois Insertion $(\mathbb{T}, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbb{A}, \leq)$. This abstraction can be systematically lifted to a Galois Insertion $\mathbb{I} \underset{\bar{\alpha}}{\overset{\bar{\gamma}}{\leftrightarrows}} [\mathbb{MGC} \to \mathbb{A}]$ by function composition (i.e., $\bar{\alpha}(f) = \alpha \circ f$).

Now we can derive the optimal abstract version of $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$ simply as $\mathcal{P}^\alpha[\![\mathcal{R}]\!] := \bar{\alpha} \circ \mathcal{P}[\![\mathcal{R}]\!] \circ \bar{\gamma}$. Abstract interpretation theory assures that $\mathcal{F}^\alpha[\![\mathcal{R}]\!] := \mathcal{P}^\alpha[\![\mathcal{R}]\!]\uparrow\omega$ is the best correct approximation of $\mathcal{F}[\![\mathcal{R}]\!]$. Correct means $\alpha(\mathcal{F}[\![\mathcal{R}]\!]) \leq \mathcal{F}^\alpha[\![\mathcal{R}]\!]$ and best means that it is the minimum (w.r.t. $\leq$) of all correct approximations. If $\mathbb{A}$ is Noetherian the abstract fixpoint is reached in a finite number of steps, that is, there exists a finite natural number h such that $\mathcal{P}^\alpha[\![\mathcal{R}]\!]\uparrow\omega = \mathcal{P}^\alpha[\![\mathcal{R}]\!]\uparrow h$.

A case study: The domain *depth*(k). An interesting finite abstraction of an infinite set of constructor terms are sets of terms up to a particular depth k, which has already been used in call-pattern analysis for functional logic programs [16], the abstract diagnosis of functional programs [3] and logic programs [12] or in the abstraction of term rewriting systems [5] (with $k = 1$).

Domains typical for analysis, like $\mathcal{POS}$, or domain of types, which lead to (very) compact specifications, could probably be better suited to show interesting instancies of our method. On the contrary specifications over *depth*(k) tend to be quite big and certainly do not scale well as k grows. However we decided to present this particular instance in order to relate our proposal with the ones in the literature: in Section 4 we can provide diagnosis examples that can be compared both with current abstract approaches (like [16,5]) and concrete ones (like [9,7,3]) when the depth of terms that show an error is less than the chosen k. Instead the comparison of the much abstract instancies with the more concrete one in the literature would obviously be unfair.

Now we show how to approximate an infinite set of incremental answer trees by means of a *depth*(k) cut $\lfloor\!\!\lfloor_k$ 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 $\widehat{\mathcal{V}}$ (disjoint from $\mathcal{V}$). *depth*(k) terms represent each term obtained by instantiating the variables of $\widehat{\mathcal{V}}$ with terms built over $\mathcal{V}$.

We extend $\lfloor\!\!\lfloor_k$ to partial answers $\{x_1/t_1, \ldots, x_n/t_n\} \cdot t_0$ essentially by cutting all t_i. However in case there is a shared term s between t_0 and another t_i and s is at depth greater than k in one of the two, in both terms we replace s with two (different) fresh variables from $\widehat{\mathcal{V}}$. Thus it can happen that variables from $\widehat{\mathcal{V}}$ appear at depth less than k.

For example, given $\alpha = \{x/S(x'), y/A(z, S(S(y)))\} \cdot \dot{B}(y, S(t), t, x')$ where $t = \dot{S}(S(\dot{z}))$, its cut $\alpha\!\downarrow_2$ is $\{x/S(x'), y/A(\hat{v}_1, S(\hat{v}_2))\} \cdot \dot{B}(\hat{v}_3, S(\dot{\hat{v}}_4), \dot{S}(\dot{\hat{v}}_5), x')$.

We define the order $\leq$ for this domain by successive lifting of the order s.t. $\hat{x} \leq t$ for every term t and variable $\hat{x} \in \widehat{\mathcal{V}}$. First we extend $\leq$ by structural induction on the structure over terms, then we extend it to substitutions by pointwise extension and then over partial answers. Finally given two $depth(k)$ partial answers trees T_1 and T_2, $T_1 \leq T_2$ iff for any $\alpha \in T_1$ exists a $\beta \in T_2$ such that $\alpha \leq \beta$. The set of all $depth(k)$ partial answers trees ordered by $\leq$ is a complete lattice. Let $\bigvee$ be its join.

The $depth(k)$ cut of a tree T is $\kappa(T) := \bigvee\{\beta\!\downarrow_k \mid \beta \in T\}$. For example, given $T = \{\varepsilon \cdot \bot,\ x/S^3(y) \cdot \dot{y},\ x/S^3(y) \cdot \dot{A}\}$, for $k = 2$, $\kappa(T) = \{\varepsilon \cdot \bot,\ x/S^3(y) \cdot \dot{y}\}$

The resulting (optimal) abstract immediate consequence operator is

$$\mathcal{P}^\kappa[\![\mathcal{R}]\!]_{I^\kappa} = \lambda f(\overrightarrow{x_n}).\ \{\varepsilon\cdot\bot\} \vee \bigvee \left\{ ((\{\overrightarrow{x_n}/\overrightarrow{t_n}\}\sigma)\restriction_{\overrightarrow{x_n}} \cdot \dot{s})\!\downarrow_k \;\middle|\; \begin{array}{l} f(\overrightarrow{t_n}) \to r \ll \mathcal{R}, \\ \sigma \cdot s \in \mathcal{E}^\kappa[\![r]\!]_{I^\kappa},\ s \neq \bot \end{array} \right\}$$

where $\mathcal{E}^\kappa[\![x]\!]_{I^\kappa} := \{\varepsilon \cdot x\}$ and

$$\mathcal{E}^\kappa[\![f(\overrightarrow{t_n})]\!]_{I^\kappa} := \left\{ (\vartheta\eta_h)\restriction_{\overrightarrow{t_n}} \cdot h[r] \;\middle|\; \begin{array}{l} \sigma_i \cdot s_i \in \mathcal{E}^\kappa[\![t_i]\!]_{I^\kappa} \text{ for } i = 1, \ldots, n \\ \vartheta = mgu(\sigma_1, \ldots, \sigma_n), \\ \mu \cdot r \ll I^\kappa(f(\overrightarrow{x_n})), \\ \exists h = \mathring{mgu}_{V(r)\cup\widehat{\mathcal{V}}}(f(\overrightarrow{x_n})\mu, f(\overrightarrow{s_n})\vartheta) \end{array} \right\}$$

Note that all examples that we show in the following are obtained by a proof of concept prototype.

Example 4. Consider the program in Example 3 and take $k = 3$. According to the previous definition, the abstract semantics of function *from* is:

$$\left\{ from(n) \mapsto \{\, \varepsilon \cdot \bot,\ \varepsilon \cdot n\dot{:}\bot,\ \varepsilon \cdot n\dot{:}S(\hat{x}_1)\dot{:}\bot,\ \varepsilon \cdot n\dot{:}S(\hat{x}_1)\dot{:}\hat{x}_2\dot{:}\hat{x}_3,\ \varepsilon \cdot n\dot{:}S(\hat{x}_1)\dot{:}\hat{x}_2\dot{:}\dot{\hat{x}}_3 \} \right.$$

Example 5. Consider the program $\mathcal{R}_{plus}$ of Example 1. For $k = 1$ $\mathcal{F}^\kappa[\![\mathcal{R}_{plus}]\!]$ is

$$\left\{ plus(x, y) \mapsto \{\, \varepsilon \cdot \bot,\ \{x/0\} \cdot \dot{y},\ \{x/S(\hat{x}_1)\} \cdot \dot{S}(\hat{x}_2),\ \{x/S(\hat{x}_1)\} \cdot \dot{S}(\dot{\hat{x}}_2) \} \right.$$

This is essentially the same result that can be obtained by [5], with the *head* upper-closure operator, that derives the following abstract TRS:

$$\begin{array}{ll} plus^a(0^a, 0^a) \to 0^a & plus^a(0^a, s^a(\top_{nat})) \to s^a(\top_{nat}) \\ plus^a(s^a(\top_{nat}), 0^a) \to s^a(\top_{nat}) & plus^a(s^a(\top_{nat}), s^a(\top_{nat})) \to s^a(\top_{nat}) \end{array}$$

Note that the first two rules of the abstract TRS are subsumed by the partial answer $\{x/0\} \cdot \dot{y}$ in our semantics. □

Example 6. Let $\mathcal{R}$ be the program obtained by adding to the one of Example 2 the following rules

```
main = disj m1
disj (C 0 (S x)) = True
m1 = pair (sub2 (S coin))
```

For $k = 2$, the abstract analysis of $\mathcal{R}$ reaches the fixpoint in 3 steps, giving

$$\begin{cases} sub2(x) \mapsto \{\varepsilon \cdot \bot,\ x/S(S(\hat{x}_1)) \cdot \dot{\hat{x}}_2\} \\ pair(x) \mapsto \{\varepsilon \cdot \bot,\ \varepsilon \cdot \dot{C}(x, x)\} \\ disj \mapsto \{\varepsilon \cdot \bot,\ \varepsilon \cdot \dot{C}(\bot, \bot),\ \varepsilon \cdot \dot{C}(\dot{\hat{x}}_2, \dot{\hat{x}}_2)\} \\ main \mapsto \{\varepsilon \cdot \bot\} \end{cases}$$

For the same program and same k [16] reaches the call pattern $disj(\top, \top) \doteq true$ causing $main \doteq true$ to be observed, which does not correspond to a concrete call pattern. However for $k \geq 3$ this false call pattern is not observed. □

It is worth noting that the resulting abstract semantics encompasses some limitations of previous works

- Since we use a "truly" goal-independent concrete semantics we obtain a much compact abstract semantics than [3]. This is exactly the reason why we are developing a compact semantics for generic TRS [2].
- For $k = 1$ if we consider TRS admissible to apply the technique of [5] we obtain the same results. However the abstract rewriting methodology of [5] requires canonicity, stratification, constructor discipline, and complete definedness for the analyses. This class of TRS is very restricted (even for functional programming) and certainly cannot cover functional logic programs. On the contrary we require only left linearity and construction basedness.
- Since we use a careful definition of the abstraction function that uses $depth(k)$ variables instead of just a $\top$ symbol like does [16] we have some slightly better results. For the same k we do not produce all false answers which are produced by [16]. These answers won't be generated by [16] for $k + 1$, but due to the quickly growing size of $depth(k)$ our improvement can be worthy.

4 Abstract Diagnosis of Functional Logic Programs

Now, following the approach of [12], we define abstract diagnosis of functional logic programs. The framework of abstract diagnosis [12] comes from the idea of considering the abstract versions of Park's Induction Principle[3]. It can be considered as an extension of declarative debugging since there are instances of the framework that deliver the same results. However, in the general case, diagnosing w.r.t. *abstract* program properties relieves the user from having to

[3] A concept of formal verification that is undecidable in general.

specify in excessive detail the program behavior (which could be more error-prone than the coding itself).

There have been several proposals about Declarative Diagnosis of functional logic languages, like [17,8,7,9]. The approach has revealed much more problematic in this paradigm than in the logic paradigm. As can be read from [8] "A more practical problem with existing debuggers for lazy functional (logic) languages is related to the presentation of the questions asked to the oracle". Actually the call-time choice model and the peculiarity of the needed narrowing strategy cannot be tackled by pure declarations about expected answers. Roughly speaking, the oracle must "understand" neededness.

As already noted by [8], the "simplification" proposed in [17] to adopt the generic approach to FLP is not well-suited. [8] aims at a debugging method that asks the oracle about computed answers that do not involve function calls, plus possible occurrences of an "undefined symbol". However this is exactly the kind of information we have in our concrete semantics, which then makes it a suitable starting point for our diagnosis methodology.

In the following, $\mathcal{S}^\alpha$ is the specification of the intended behavior of a program w.r.t. the property α.

Definition 5. *Let $\mathcal{R}$ be a program, α be a property over domain $\mathbb{A}$ and $\mathcal{S}^\alpha \in \mathbb{A}$.*

1. *$\mathcal{R}$ is (abstractly)* partially correct *w.r.t. $\mathcal{S}^\alpha$ if $\alpha(\mathcal{F}[\![\mathcal{R}]\!]) \leq \mathcal{S}^\alpha$.*
2. *$\mathcal{R}$ is (abstractly)* complete *w.r.t. $\mathcal{S}^\alpha$ if $\mathcal{S}^\alpha \leq \alpha(\mathcal{F}[\![\mathcal{R}]\!])$.*
3. *$\mathcal{R}$ is* totally correct *w.r.t. $\mathcal{S}^\alpha$, 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 $\alpha(\mathcal{F}[\![\mathcal{R}]\!])$ and not in terms of the (possibly less precise) abstract semantics $\mathcal{F}^\alpha[\![\mathcal{R}]\!]$. $\mathcal{S}^\alpha$ is the abstraction of the intended concrete semantics of $\mathcal{R}$. Thus, the user 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 (as well as finite failures). We cannot address termination issues here, since the concrete semantics we use is too abstract.

The *diagnosis* determines the "originating" 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 element*.

Definition 6. *Let $\mathcal{R}$ be a program, R a rule and $e, \mathcal{S}^\alpha \in \mathbb{A}$.*
R is abstractly incorrect *w.r.t. $\mathcal{S}^\alpha$ if $\mathcal{P}^\alpha[\![\{R\}]\!]_{\mathcal{S}^\alpha} \not\leq \mathcal{S}^\alpha$.*
e is an uncovered element *w.r.t. $\mathcal{S}^\alpha$ if $e \leq \mathcal{S}^\alpha$ and $e \wedge \mathcal{P}^\alpha[\![\mathcal{R}]\!]_{\mathcal{S}^\alpha} = \bot$*[4].

Informally, R is abstractly incorrect if it derives a wrong abstract element from the intended semantics. e is uncovered if there are no rules deriving it from the intended semantics. It is worth noting that checking these conditions requires one application of $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$ to $\mathcal{S}^\alpha$, while the standard detection based on symptoms would require the construction of $\alpha(\mathcal{F}[\![\mathcal{R}]\!])$ and therefore a fixpoint computation.

[4] Note that $e \wedge \mathcal{P}^\alpha[\![\mathcal{R}]\!]_{\mathcal{S}^\alpha} = \bot$ implies $e \not\leq \mathcal{P}^\alpha[\![\mathcal{R}]\!]_{\mathcal{S}^\alpha}$, but the converse is not true. Thus this definition is meant to detect "atomic" uncovered elements.

It is worth noting that correctness and completeness are defined in terms of $\alpha(\mathcal{F}^\alpha[\![\mathcal{R}]\!])$, i.e., in terms of abstraction of the concrete semantics. On the other hand, abstractly incorrect rules and abstract uncovered elements are defined directly in terms of abstract computations (the abstract immediate consequence operator $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$). In this section, we are left with the problem of formally 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. $\mathcal{S}^\alpha$.*

Proof. By hypothesis $\forall r \in \mathcal{R}.\ \mathcal{P}^\alpha[\![\{r\}]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$. Hence $\mathcal{P}^\alpha[\![\mathcal{R}]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$, i.e., $\mathcal{S}^\alpha$ is a pre-fixpoint of $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$. Since $\alpha(\mathcal{F}[\![\mathcal{R}]\!]) \leq \mathcal{F}^\alpha[\![\mathcal{R}]\!] = \mathit{lfp}\, \mathcal{P}^\alpha[\![\mathcal{R}]\!]$, by Knaster–Tarski's Theorem $\alpha(\mathcal{F}[\![\mathcal{R}]\!]) \leq \mathcal{F}^\alpha[\![\mathcal{R}]\!] \leq \mathcal{S}^\alpha$. The thesis follows by Point 1 of Definition 5. □

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

Proof. By construction $\alpha \circ \mathcal{P}[\![\mathcal{R}]\!] \circ \gamma \leq \mathcal{P}^\alpha[\![\mathcal{R}]\!]$, hence $\alpha \circ \mathcal{P}[\![\mathcal{R}]\!] \circ \gamma \circ \alpha \leq \mathcal{P}^\alpha[\![\mathcal{R}]\!] \circ \alpha$. Since $id \sqsubseteq \gamma \circ \alpha$, it holds that $\alpha \circ \mathcal{P}[\![\mathcal{R}]\!] \leq \alpha \circ \mathcal{P}[\![\mathcal{R}]\!] \circ \gamma \circ \alpha$ and $\alpha \circ \mathcal{P}[\![\mathcal{R}]\!] \leq \mathcal{P}^\alpha[\![\mathcal{R}]\!] \circ \alpha$. Hence,

$$
\begin{array}{ll}
\alpha(\mathcal{F}[\![\mathcal{R}]\!]) = & [\text{since } \mathcal{F}[\![\mathcal{R}]\!] \text{ is a fixpoint}] \\
\alpha(\mathcal{P}[\![\mathcal{R}]\!]_{\mathcal{F}[\![\mathcal{R}]\!]}) \leq & [\text{by } \alpha \circ \mathcal{P}[\![\mathcal{R}]\!] \leq \mathcal{P}^\alpha[\![\mathcal{R}]\!] \circ \alpha] \\
\mathcal{P}^\alpha[\![\mathcal{R}]\!]_{\alpha(\mathcal{F}[\![\mathcal{R}]\!])} \leq & [\text{since } \mathcal{P}^\alpha[\![\mathcal{R}]\!] \text{ is monotone and } \mathcal{R} \text{ is partially correct}] \\
\mathcal{P}^\alpha[\![\mathcal{R}]\!]_{\mathcal{S}^\alpha} &
\end{array}
$$

Now, if $\mathcal{R}$ has an abstract uncovered element e i.e., $e \leq \mathcal{S}^\alpha$ and $e \wedge \mathcal{P}^\alpha[\![\mathcal{R}]\!]_{\mathcal{S}^\alpha} = \bot$, then $e \wedge \alpha(\mathcal{F}[\![\mathcal{R}]\!]) = \bot$ and $\mathcal{S}^\alpha \not\leq \alpha(\mathcal{F}[\![\mathcal{R}]\!])$. The thesis follows from Point 2 of Definition 5. □

Abstract incorrect rules are in general just a warning about a possible source of errors. Because of the approximation, it can happen that a (concretely) correct rule is abstractly incorrect.

However, as shown by the following theorems, all concrete errors — that are "visible"[5] in the abstract domain — are detected as they lead to an abstract incorrectness or abstract uncovered.

Theorem 4. *Let r be a rule, $\mathcal{S}$ a concrete specification. If $\mathcal{P}[\![\{r\}]\!]_{\mathcal{S}} \not\sqsubseteq \mathcal{S}$ and $\alpha(\mathcal{P}[\![\{r\}]\!]_{\mathcal{S}}) \not\leq \alpha(\mathcal{S})$ then r is abstractly incorrect w.r.t. $\alpha(\mathcal{S})$.*

Proof. Since $\mathcal{S} \sqsubseteq \gamma \circ \alpha(\mathcal{S})$, by monotonicity of α and the correctness of $\mathcal{P}^\alpha[\![\{r\}]\!]$, it holds that $\alpha(\mathcal{P}[\![\{r\}]\!]_{\mathcal{S}}) \leq \alpha(\mathcal{P}[\![\{r\}]\!]_{\gamma\circ\alpha(\mathcal{S})}) \leq \mathcal{P}^\alpha[\![\{r\}]\!]_{\alpha(\mathcal{S})}$. By hypothesis $\alpha(\mathcal{P}[\![\{r\}]\!]_{\mathcal{S}}) \not\leq \alpha(\mathcal{S})$, therefore $\mathcal{P}^\alpha[\![\{r\}]\!]_{\alpha(\mathcal{S})} \not\leq \alpha(\mathcal{S})$, since $\alpha(\mathcal{P}[\![\{r\}]\!]_{\mathcal{S}}) \leq \mathcal{P}^\alpha[\![\{r\}]\!]_{\alpha(\mathcal{S})}$. The thesis holds by Definition 6. □

[5] A concrete symptom is visible if its abstraction is different from the abstraction of correct answers. For example if we abstract to the length of lists, an incorrect rule producing wrong lists of the same length of the correct ones is not visible.

Theorem 5. *Let $\mathcal{S}$ be a concrete specification. If there exists an abstract uncovered element a w.r.t. $\alpha(\mathcal{S})$, such that $\gamma(a) \sqsubseteq \mathcal{S}$ and $\gamma(\bot) = \bot$, then there exists a concrete uncovered element e w.r.t. $\mathcal{S}$ (i.e., $e \sqsubseteq \mathcal{S}$ and $e \sqcap \mathcal{P}[\![\mathcal{R}]\!]_{\mathcal{S}} = \bot$).*

Proof. By hypothesis $a \leq \alpha(\mathcal{S})$ and $a \wedge \mathcal{P}^{\alpha}[\![\mathcal{R}]\!]_{\alpha(\mathcal{S})} = \bot$. Hence, since $\gamma(\bot) = \bot$ and γ preserves greatest lower bounds, $\gamma(a) \sqcap \gamma(\mathcal{P}^{\alpha}[\![\mathcal{R}]\!]_{\alpha(\mathcal{S})}) = \bot$. By construction $\mathcal{P}^{\alpha}[\![\mathcal{R}]\!] = \alpha \circ \mathcal{P}[\![\mathcal{R}]\!] \circ \gamma$, thus $\gamma(a) \sqcap \gamma(\alpha(\mathcal{P}[\![\mathcal{R}]\!]_{\gamma(\alpha(\mathcal{S}))})) = \bot$. Since $id \sqsubseteq \gamma \circ \alpha$ and by monotonicity of $\mathcal{P}[\![\mathcal{R}]\!]$, $\gamma(a) \wedge \mathcal{P}[\![\mathcal{R}]\!]_{\mathcal{S}} = \bot$. By hypothesis $\gamma(a) \sqsubseteq \mathcal{S}$ hence $\gamma(a)$ is a concrete uncovered element. □

The diagnosis w.r.t. approximate properties over Noetherian domains is always effective, because the abstract specification is finite. However, 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.
- every uncovered is identified by an abstract uncovered. However an abstract uncovered does not always correspond to a bug.
- there exists no sufficient condition for completeness.

It is important to note that our method, since it has been derived by (properly) applying abstract interpretation techniques, is correct by construction.

Another property of our proposal, which is particularly useful for application, is that

- it can be used with partial specifications,
- it can be used with partial programs.

Obviously one cannot detect errors in rules involving functions which have not been specified. But for the rules that involve only functions that have a specification the check can be made, even if the whole program has not been written yet. This includes the possibility of applying our "local" method to all parts of a program not involving constructs which we cannot handle (yet). With other "global" approaches such programs could not be checked at all.

We now show the *depth*(k) instance of our methodology to provide a demonstrative application of our abstract diagnosis framework. It shows some encouraging results. Thus it will be interesting in the future to experiment also with other possible instances over more sophisticated domains.

Our case study: abstract diagnosis on *depth*(k). Now we show how we can derive an efficacious debugger by choosing suitable instances of the general framework described above. We consider the *depth*(k) abstraction κ presented in Section 3.2.

Note that *depth*(k) partial answers that do not contain variables belonging to $\widehat{\mathcal{V}}$ actually are concrete answers. Thus an additional benefit w.r.t. the general results is that errors exhibiting symptoms without variables belonging to $\widehat{\mathcal{V}}$ are actually concrete (real) errors (see Theorem 4).

Example 7. Consider the buggy program for *from* (proposed in [7]) with rule R: $from(n) \to n : from(n)$ and as intended specification $\mathcal{S}^\kappa$ the fixpoint from Example 4. We detect that rule R is abstractly incorrect since

$$\mathcal{P}^\kappa[\![\{R\}]\!]_{\mathcal{S}^\kappa} = \begin{cases} from(n) \mapsto \{\, \varepsilon \cdot \bot,\ \varepsilon \cdot n\dot{:}\bot,\ \varepsilon \cdot n\dot{:}n\dot{:}\bot, \\ \qquad\qquad \varepsilon \cdot n\dot{:}n\dot{:}\hat{x}_2\dot{:}\hat{x}_3,\ \varepsilon \cdot n\dot{:}n\dot{:}\hat{x}_2\dot{:}\hat{x}_3 \,\} \end{cases} \not\trianglelefteq \mathcal{S}^\kappa$$

Example 8. Consider the buggy program $\mathcal{R}_{bug}$

```
      main = C (h (f x)) x
      h (S x) = 0
R:    f (S x) = S 0
```

where rule R should have been f x = S (h x) to be correct w.r.t. the intended semantics on $depth(k)$, with $k > 2$,

$$\mathcal{S}^\kappa = \begin{cases} f(x) \mapsto \{\varepsilon \cdot \bot,\ \varepsilon \cdot \dot{S}(\bot),\ \{x/S(x')\} \cdot \dot{S}(\dot{0})\} \\ h(x) \mapsto \{\varepsilon \cdot \bot,\ \{x/S(x')\} \cdot \dot{0}\} \\ main \mapsto \{\varepsilon \cdot \bot,\ \varepsilon \cdot \dot{C}(\bot, x),\ \varepsilon \cdot \dot{C}(\dot{0}, x)\} \end{cases}$$

This error preserves the computed answer behavior both for h and f, but not for *main*. In fact, *main* evaluates to $\{x/S(x')\} \cdot C(0, S(x'))$. Rule R is abstractly incorrect.

Note that the diagnosis method of [9,7] does not report incorrectness errors (as there are no incorrectness symptoms) while it reports the missing answer $\varepsilon \cdot C(0, 0)$. □

Example 9. Consider the following buggy program for *double* w.r.t. the $depth(2)$ specification $\mathcal{S}^\kappa$.

```
R1: double 0   = s 0
R2: double (S x) = S (double x)
```

$$\mathcal{S}^\kappa := \begin{cases} double(x) \mapsto \{\, \varepsilon \cdot \bot,\ \{x/0\} \cdot \dot{0},\ \{x/S(x')\} \cdot \dot{S}(S(\hat{y})), \\ \qquad\qquad \{x/S(0)\} \cdot \dot{S}(S(\dot{\hat{y}})),\ \{x/S(S(\hat{x}_1)\} \cdot \dot{S}(S(\dot{\hat{y}})) \,\} \end{cases}$$

We can detect that both R_1 and R_2 are abstractly incorrect, since

$$\mathcal{P}^\kappa[\![\{R_1\}]\!]_{\mathcal{S}^\kappa}(double(x)) = \{\varepsilon \cdot \bot,\ \{x/0\} \cdot \dot{S}(0)\}$$

$$\mathcal{P}^\kappa[\![\{R_2\}]\!]_{\mathcal{S}^\kappa}(double(x)) = \{\, \varepsilon \cdot \bot,\ \varepsilon \cdot \dot{S}(\bot),\ \{x/S(0)\} \cdot \dot{S}(\dot{0}) \quad \{x/S(S(\hat{x}_1)\} \cdot \dot{S}(\dot{S}(\dot{\hat{y}}))\ \{x/S(S(\hat{x}_1)\} \cdot \dot{S}(\dot{S}(\dot{\hat{y}}))\}$$

However the debugger in [3] is not able to determine for $k = 2$ that rule R_2 is incorrect. This is because the errors in both the rules interfere with each other, making the goal $double(S(0))$ asymptomatic. □

We run our prototype on the possible benchmarks from the NOFIB-Buggy collection available at http://einstein.dsic.upv.es/nofib-buggy. Not all benchmarks can be checked yet since errors are in rules using higher-order or I/O or arithmetical features, which our semantic framework can not handle. However, since our methodology processes each rule independently, we can anyway find errors in the first-order rules of general programs, giving a partial specification. Indeed we discovered the incorrect rule in `boyer` at depth-4; we detected at depth-1 the incorrect rules of `clausify`, `knights` and `lift`[6]; we found some uncovered elements for `pretty`, `sorting` and `fluid`[7] at depth-8, depth-3 and depth-2, respectively.

The case of `boyer` is particularly interesting, because the error introduces nontermination and prevents declarative debuggers to be applied. Our methodology is not concerned by this kind of errors because it does not compute fixpoints.

The selection of the appropriate depth for the abstraction is a sensitive point of this instance, since the abstraction might not be precise enough. For example if we consider $k = 2$ in Example 7 we do not detect the incorrectness. The question of whether an optimal depth exists such that no additional errors are detected by considering deeper cuts is an interesting open problem which we plan to investigate as further work.

It is important to note that the resulting abstract diagnosis encompasses some limitations of other works on declarative diagnosis:

- Symptoms involving infinite answers cannot be directly tackled by [9,7], while, if the error manifests in the first part of the infinite answer we detect it (Example 7).
- If we just compare the actual and the intended semantics of a program some incorrectness bugs can be "hidden" because of an incompleteness bug. Our technique does not suffer of error interference (Examples 8 and 9), as each possible source of error is checked in isolation. Moreover we detect all errors simultaneously.
- It can be applied on partial programs.

5 Applicability of the Framework

Depending on the chosen abstract domain very different scenarios arise about the pragmatic feasibility of our proposal. The main difference depends on the size of the abstract domain.

Small domains. Specifications are small (and thus it is quite practical for the user to write them directly). However, due to the great amount in approximation,
  - the number of false positives (abstract errors not corresponding to concrete ones) increases;

[6] An adjustment for some rules was needed in order to simulate Haskell selection rule.
[7] Minor changes has been done to replace arithmetic operations with Peano's notation.

– the number of not visible concrete errors increases.

For the former issue it is possible to adapt a multi-domain approach that uses, in addition to the first, other more concrete abstract domains. For all abstract incorrect rules detected with the first abstract domain the user can be asked to write a (partial) more concrete specification limited to the functions involved in the abstract incorrect rules. The overhead of giving a more concrete specification is just *localized* by need.

We have made some experiments on a small domain, namely the domain $\mathcal{POS}$ for groundness dependencies analysis. This experiments showed that manually writing the specification is certainly affordable in this case. However, as expected, the resulting instance it is not very powerful in detecting errors.

Details about the abstraction over this domain can be found in [4].

Big domains. Due to the good precision of these domains, we can detect errors more precisely. However in these cases it is unreasonable to provide the whole specification manually. One possible solution to this issue is to compute the fixpoint of the buggy program and then present the user with the actual semantics so he can inspect it and just modify the results that he does not expect. All the explicit information given directly by the user can be saved (within the code itself) and then reused in successive sessions.

The pragmatical choice of an abstract domain is thus a tradeoff between the precision of errors that can be detected and the effort in providing the specification.

As a final remark about applicability of the current proposal beyond the first order fragment, let us note that we can tackle higher-order features and primitive operations of functional (logic) languages in the same way proposed by [16]: by using the technique known as "defunctionalization" and by approximating, very roughly, calls to primitive operations, that are not explicitly defined by rewrite rules, just with variables over $\widehat{\mathcal{V}}$.

6 Conclusions

We have presented a fixpoint semantics $\mathcal{P}[\![\mathcal{R}]\!]$ for functional programs. Our semantics allows us to model correctly non-determinism, non-strict functions defined by non-confluent programs and call-time choice behaviour.

Then, we formulated an efficacious generic scheme for the declarative debugging of functional logic programs based on approximating the $\mathcal{P}[\![\mathcal{R}]\!]$ operator by means of an abstract $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$ operator obtained by abstract interpretation. Our approach is based on the ideas of [3,12] which we apply to the diagnosis of functional logic programs.

We showed that, given the intended abstract specification $\mathcal{S}^\alpha$ of the semantics of a program $\mathcal{R}$, we can check the correctness of $\mathcal{R}$ by a single step of $\mathcal{P}^\alpha[\![\mathcal{R}]\!]$ and, by a simple static test, we can determine all the rules which are wrong w.r.t. the considered abstract property.

We presented a particular instance of our methodology in order to compare our proposal with [5,8,7,9,16] and showed several examples where we have better results.

For future work, on the abstract domains side, we intend to derive the instancies of the framework for the reduced products between $depth(k)$ and several domain typical of program analysis, like $\mathcal{POS}$, sharing, types, *etc.*. This could provide better tradeoffs between precision and size of specifications.

On the concrete semantics side, we intend to extend the base semantics to directly incorporate higher-order functions and residuation, in order to develop abstract diagnosis for full multi-paradigm languages like Curry and $\mathcal{TOY}$.

We also plan to develop a more concrete base semantics which can model "functional dependencies" (in order to tackle pre-post conditions) that could be employed to define abstract verification [10] for functional logic programs.

References

1. Alpuente, M., Ballis, D., Correa, F., Falaschi, M.: An Integrated Framework for the Diagnosis and Correction of Rule-Based Programs. Theoretical Computer Science 411(47), 4055–4101 (2010)
2. Alpuente, M., Comini, M., Escobar, S., Falaschi, M., Iborra, J.: A Compact Fixpoint Semantics for Term Rewriting Systems. Theoretical Computer Science 411(37), 3348–3371 (2010)
3. Alpuente, M., Comini, M., Escobar, S., Falaschi, M., Lucas, S.: Abstract Diagnosis of Functional Programs. In: Leuschel, M. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 1–16. Springer, Heidelberg (2003)
4. Bacci, G., Comini, M.: A Compact Goal-Independent Bottom-Up Fixpoint Modeling of the Behaviour of First Order Curry. Technical Report DIMI-UD/06/2010/RR, Dipartimento di Matematica e Informatica, Università di Udine (2010),
`http://www.dimi.uniud.it/comini/Papers/`
5. Bert, D., Echahed, R.: Abstraction of Conditional Term Rewriting Systems. In: Lloyd, J.W. (ed.) Proceedings of the 1995 Int'l Symposium on Logic Programming (ILPS 1995), pp. 162–176. The MIT Press, Cambridge (1995)
6. Braßel, B.: A Technique to build Debugging Tools for Lazy Functional Logic Languages. In: Falaschi, M. (ed.) Proceedings of the 17th Workshop on Functional and (Constraint) Logic Programming (WFLP 2008), pp. 63–76 (2008)
7. Caballero-Roldán, R.: A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In: WCFLP 2005: Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, pp. 8–13. ACM Press, New York (2005)
8. Caballero, R., López-Fraguas, F.J., Rodríguez-Artalejo, M.: Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In: Kuchen, H., Ueda, K. (eds.) FLOPS 2001. LNCS, vol. 2024, pp. 170–184. Springer, Heidelberg (2001)
9. Caballero-Roldán, R., Rodríquez-Artalejo, M., del Vado-Vírseda, R.: Declarative Diagnosis of Missing Answers in Constraint Functional-Logic Programming. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 305–321. Springer, Heidelberg (2008)
10. Comini, M., Gori, R., Levi, G., Volpe, P.: Abstract Interpretation based Verification of Logic Programs. Science of Computer Programming 49(1-3), 89–123 (2003)

11. Comini, M., Levi, G., Meo, M.C., Vitiello, G.: Proving properties of Logic Programs by Abstract Diagnosis. In: Dam, M. (ed.) LOMAPS-WS 1996. LNCS, vol. 1192, pp. 22–50. Springer, Heidelberg (1997)
12. Comini, M., Levi, G., Meo, M.C., Vitiello, G.: Abstract Diagnosis. Journal of Logic Programming 39(1-3), 43–93 (1999)
13. Comini, M., Levi, G., Vitiello, G.: Declarative Diagnosis Revisited. In: Lloyd, J.W. (ed.) Proceedings of the 1995 Int'l Symposium on Logic Programming (ILPS 1995), pp. 275–287. The MIT Press, Cambridge (1995)
14. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Antonio, Texas, January 29-31, pp. 269–282. ACM Press, New York (1979)
15. Echahed, R., Janodet, J.-C.: Admissible Graph Rewriting and Narrowing. In: Proceedings of the Joint International Conference and Symposium on Logic Programming, pp. 325–340. MIT Press, Cambridge (1998)
16. Hanus, M.: Call pattern analysis for functional logic programs. In: Proc. of the 10th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP 2008), pp. 67–78. ACM Press, New York (2008)
17. Naish, L., Barbour, T.: A Declarative Debugger for a Logical-Functional Language. In: Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, vol. 2, pp. 91–99 (1995)
18. Nilsson, H., Fritzson, P.: Algoritmic debugging for lazy functional languages. Journal of Functional Programming 4(1), 337–370 (1994)
19. Palamidessi, C.: Algebraic Properties of Idempotent Substitutions. In: Paterson, M.S. (ed.) ICALP 1990. LNCS, vol. 443, pp. 386–399. Springer, Heidelberg (1990)