Declarative Diagnosis of Missing Answers in Constraint Functional-Logic Programming

Rafael Caballero, Mario Rodríguez Artalejo,
and Rafael del Vado Vírseda*

Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
{rafa,mario,rdelvado}@sip.ucm.es

Abstract. We present a declarative method for diagnosing *missing computed answers* in $CFLP(\mathcal{D})$, a generic scheme for lazy *Constraint Functional-Logic Programming* which can be instantiated by any constraint domain $\mathcal{D}$ given as parameter. As far as we know, declarative diagnosis of missing answers in such an expressive framework has not been tackled before. Our approach combines and extends previous work done separately for constraint logic programming and lazy functional programming languages. Diagnosis can be started whenever a user finds that the set of computed answers for a given goal with finite search space misses some expected solution w.r.t. an *intended interpretation* of the program, that provides a declarative description of its expected behavior. Diagnosis proceeds by exploring a *proof tree*, that provides a declarative view of the *answer-collection* process performed by the computation, and it ends up with the detection of some function definition in the program that is incomplete w.r.t. the intended interpretation. We can prove the *logical correctness* of the diagnosis method under the assumption that the recollection of computed answers performed by the goal solving system can be represented as a proof tree. We argue the plausibility of this assumption, and we describe the prototype of a tool which implements the diagnosis method.

1 Introduction

Debuggers are a practical need for helping programmers to understand why their programs do not work as intended. Declarative programming paradigms involving complex operational details, such as constraint solving and lazy evaluation, do not fit well to traditional debugging techniques relying on the inspection of low-level computation traces. For this reason, the design of usable debugging tools becomes a difficult task. As a solution to this problem, and following a seminal idea by Shapiro [28], *declarative diagnosis* (a.k.a. *declarative debugging* or *algorithmic debugging*) proposes to use *Computation Trees* (shortly, *CT*s) in

* The authors have been partially supported by the Spanish National Projects MERIT-FORMS (TIN2005-09027-C03-03) and PROMESAS-CAM (S-0505/TIC/0407).

J. Garrigue and M. Hermenegildo (Eds.): FLOPS 2008, LNCS 4989, pp. 305–321, 2008.

place of traces. CTs are built *a posteriori* to represent the structure of a computation whose top-level outcome is regarded as a symptom of the unexpected behavior by the user, with results attached to their nodes representing the computation of some observable result, and such that the result at any internal node follows from the results at the children nodes, using a program fragment also attached to the node. Declarative diagnosis explores a CT looking for a so-called *buggy node* which computes an unexpected result from children whose results are all expected. Each buggy node points to a program fragment responsible for the unexpected behavior. The search for a buggy node can be implemented with the help of an external *oracle* (usually the user with some semiautomatic support) who has a reliable declarative knowledge of the expected program semantics, the so-called *intended interpretation*.

The generic description of declarative diagnosis in the previous paragraph follows [22]. Declarative diagnosis was first proposed in the field of *Logic Programming* (LP) [28,14,18], and it has been successfully extended to other declarative programming paradigms, including (lazy) *Functional Programming* (FP) [25,24,27,26], *Constraint Logic Programming* (CLP) [1,30,15] and *Functional Logic Programming* (FLP) [23,6,7]. The nature of unexpected results differs according to the programming paradigm. Unexpected results in FP are mainly *incorrect values*, while in CLP and FLP an unexpected result can be either a single computed answer regarded as *incorrect*, or a set of computed answers (for one and the same goal with a finite search space) regarded as *incomplete*. These two possibilities give rise to the declarative diagnosis of *wrong* and *missing* computed answers, respectively. The case of unexpected *finite failure* of a goal is a particular symptom of missing answers with special relevance. However, diagnosis methods must consider the more general case, since finite failure of a goal is often caused by non-failing subgoals that do not compute all the expected answers.

In contrast to alternative approaches to error diagnosis based on *abstract interpretation* techniques [17], declarative diagnosis often involves complex queries to the user. This problem has been tackled by means of various techniques, such as user-given partial specifications of the program's semantics [1,7], safe inference of information from answers previously given by the user [6], or CTs tailored to the needs of a particular debugging problem over a particular computation domain [15]. Another practical problem with declarative diagnosis is that the size of CTs can cause excessive overhead in the case of computations that demand a big amount of computer storage. As a remedy, techniques for piecemeal construction of CTs have been considered; see [26] for a recent proposal in the FP field.

In spite of the above mentioned difficulties, we are confident that declarative diagnosis methods can be useful for detecting programming bugs by observing computations whose demand of computer storage is modest. In this paper, we present a declarative method for diagnosing *missing computed answers* in $CFLP(\mathcal{D})$ [20], a generic scheme for lazy *Constraint Functional-Logic Programming* which can be instantiated by any constraint domain $\mathcal{D}$ given as parameter, and supports a powerful combination of functional and constraint logic programming over $\mathcal{D}$. Sound and complete goal solving procedures for the $CFLP(\mathcal{D})$

scheme have been obtained [19,11,12]. Moreover, useful instances of this scheme have been implemented in the $\mathcal{TOY}$ system [21] and tested in practical applications [13].

The rest of the paper is organized as follows: Section 2 motivates our approach and presents a debugging example, intended to illustrate the main features of our diagnosis method. Section 3 presents the abbreviated proof trees used as CTs in our method, as well as the results ensuring the logical correctness of the diagnosis. Section 4 presents a prototype debugger under development, and Section 5 concludes and gives an overview of planned future work. Full proofs of the main results given in Section 3 are available in [10].

2 Motivation

While methods and tools for the declarative diagnosis of *wrong answers* are known for FLP [23,6,7] and $CFLP$ [4,8] languages, we are not aware of any research concerning the declarative diagnosis of *missing answers* in $CFLP$ languages, except our poster presentation [9]. However, missing answers are a common problem which can arise even in the absence of wrong answers.

We are interested in the declarative diagnosis of missing answers in $CFLP(\mathcal{D})$ [20], a very expressive generic scheme for Functional and Constraint Logic Programming over a constraint domain $\mathcal{D}$ given as parameter. Each constraint domain provides basic values and primitive operations for building domain specific constraints to be used in programs and goals. Useful constraint domains include the Herbrand domain $\mathcal{H}$ for equality (==) and disequality (/=) constraints over constructed data values; the domain $\mathcal{R}$ for arithmetic constraints over real numbers; and the domain $\mathcal{FD}$ for finite domain constraints over integer values.

The $CFLP(\mathcal{D})$ scheme supports programming with lazy functions that may be non-deterministic and/or higher-order. *Programs* $\mathcal{P}$ include *program rules* of the form $f\, t_1 \ldots t_n \rightarrow r \Leftarrow \Delta$, abbreviated as $f\, \overline{t}_n \rightarrow r \Leftarrow \Delta$, with Δ omitted if empty. Such a rule specifies that f when acting over parameters matching the patterns $\overline{t}_n$ at the left hand side, will return the values resulting from the right hand side expression r, provided that the constraints in Δ can be satisfied. *Goals* G for a given program have the general form $\exists\overline{U}.\ (R \,\square\, S)$, where $\exists\overline{U}$ is an existentially quantified prefix of local variables, $R = (P \,\square\, \Delta)$ is the yet *unsolved part*, including *productions* $e \rightarrow s$ in P and *constraints* in Δ, and $S = (\Pi \,\square\, \sigma)$ is the *constraint store*, consisting of *primitive constraints* Π and an *idempotent substitution* σ. Productions $e \rightarrow s$ are solved by *lazy narrowing*, a combination of unification and lazy evaluation; the expression e must be narrowed to match the pattern s. *Initial goals* have neither productions nor local variables, and *solved goals* have the form $\exists\overline{U}.\ S$. Solved goals are also called *computed answers* and abbreviated as $\hat{S}$.

In this paper we focus mainly in $CFLP(\mathcal{D})$ programming as implemented in $\mathcal{TOY}$ [21]. The interested reader is referred to [20,19,11] for formal details on the declarative and operational semantics of the $CFLP(\mathcal{D})$ scheme.

The following small $CFLP(\mathcal{H})$-program $\mathcal{P}_{\texttt{fD}}$, written in $\mathcal{TOY}$ syntax, includes program rules for the non-deterministic functions `(//)` and `fDiff`, and the deterministic functions `gen` and `even`. Note the infix syntax used for `(//)`, as well as the use of the equality symbol `=` in place of the rewrite arrow `-->` for the program rules of those functions viewed as deterministic by the user. This is just meant as user given information, not checked by the $\mathcal{TOY}$ system, which treats all the program defined functions as possibly non-deterministic.

```
infixr 40 //                % non-deterministic choice operator

(//) :: A -> A -> A
X // _ --> X
_ // Y --> Y

fDiff :: [A] -> A
fDiff [X]      --> X
fDiff (X:Y:Zs) --> X // fDiff (Y:Zs) <== X /= Y
fDiff (X:Y:Zs) --> X                 <== X == Y

gen :: A -> A -> [A]        even :: int -> bool
gen X Y = X : Y : gen Y X   even N = true <== (mod N 2) == 0
```

Function `fDiff` is intended to return any element belonging to the longest prefix `Xs` of the list given as parameter such that `Xs` does not include two identical elements in consecutive positions. In general, there will be several such elements, and therefore `fDiff` is non-deterministic. Function `gen` is deterministic and returns a potentially infinite list of the form $[d_1, d_2, d_2, d_1, d_1, d_2, ...]$, where the elements d_1 and d_2 are the given parameters. Therefore, the lazy evaluation of `(fDiff (gen 1 2))` is expected to yield the two possible results `1` and `2` in alternative computations, and the initial goal $G_{\texttt{fD}}$: `even (fDiff (gen 1 2)) == true` for $\mathcal{P}_{\texttt{fD}}$ is expected to succeed, since `(fDiff (gen 1 2))` is expected to return the even number `2`. However, if the third program rule for function `fDiff` were missing in program $\mathcal{P}_{\texttt{fD}}$, the expression `(fDiff (gen 1 2))` would return only the numeric value `1`, and therefore the goal $G_{\texttt{fD}}$ would fail unexpectedly. At this point, a diagnosis for missing answers could take place, looking for a *buggy node* in a suitable CT in order to detect some incomplete function definition (that of function `fDiff`, in this case) to be blamed for the missing answers.

We propose to use CTs whose nodes have attached so-called *answer collection assertions*, briefly *aca*s. The *aca* at the root node has the form $G_0 \Rightarrow \bigvee_{i \in I} \hat{S}_i$, where G_0 is the initial goal and $\bigvee_{i \in I} \hat{S}_i$ (written as the *failure symbol* ⧫ if $I = \emptyset$) is the disjunction of computed answers observed by the user. This root *aca* asserts that the computed answers cover all the solutions of the initial goal, and will be regarded as a false statement in case that the user misses computed answers. For example, the root *aca* corresponding to the initial goal $G_{\texttt{fD}}$ for program $\mathcal{P}_{\texttt{fD}}$ is `even (fDiff (gen 1 2)) == true` $\Rightarrow$ ⧫ stating that this goal has (unexpectedly) failed. The *aca*s at internal nodes in our CTs have the form $f\overline{t}_n \rightarrow t \,\square\, S \Rightarrow \bigvee_{i \in I} \hat{S}_i$, asserting that the disjunction of computed answers

$\bigvee_{i \in I} \hat{S}_i$ covers all the solutions for the intermediate goal $G' : f\overline{t}_n \rightarrow t \,\square\, S$. Note that G' asks for the solutions of the production $f\overline{t}_n \rightarrow t$ which satisfy the constraint store S. The *aca*s of this form correspond to the intermediate calls to program defined functions f needed for collecting all the answers computed for the initial goal G_0. Due to *lazy evaluation*, the parameters $\overline{t}_n$ and the result t will appear in the most evaluated form demanded by the topmost computation. When these values are functions, they are represented in terms of partial applications of top-level function names. This is satisfactory under the assumption that no local function definitions are allowed in programs, as it happens in $\mathcal{TOY}$.

We build our *CT*s as abbreviated *proof trees* w.r.t. a logically sound inference system for deriving *aca*s. For this reason, our *CT*s are such that the validity of the *aca* at each node follows from the validity of the *aca*s at their children, under the assumption that the function definition relating the parent node to the children nodes is complete w.r.t. the *intended interpretation* of the program. Any *CT* whose root *aca* is invalid must include at least one *buggy node* labeled with an invalid *aca* and whose children are all labeled with valid *aca*s. Each buggy node N is related to some particular function f whose program rules are responsible for the computation of the *aca* at N from the *aca*s at N's children. Therefore, the program rules for f can be diagnosed as incomplete. The search for a buggy node can be implemented with the help of an external *oracle* who has a reliable declarative knowledge of the valid *aca*s w.r.t. the intended program interpretation. Since the oracle is usually the programmer, she can even experiment with different choices of the intended interpretation in order to obtain different diagnosis of possibly incomplete functions.

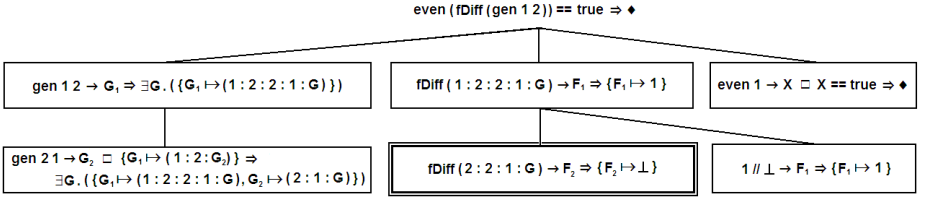

Fig. 1. *CT* for the declarative diagnosis of missing answers

A *CT* corresponding to the goal $G_{\texttt{fD}}$ for program $\mathcal{P}_{\texttt{fD}}$ (with the third program rule for function `fDiff` omitted) is displayed in Fig. 1. More on its structure and construction will be explained in Section 3. In this case, the programmer will judge the root *aca* as *invalid* because she did not expect finite failure. Moreover, from her knowledge of the intended interpretation, she will decide to consider the *aca*s for the functions `gen`, `even` and `(//)` as valid. However, the *aca* `fDiff (2:2:1:G)` $\rightarrow F_2 \Rightarrow (F_2 \mapsto \bot)$ asserts that the *undefined value* $\bot$ is the only possible result for the function call `fDiff (2:2:1:G)`, while the user expects also the result `2`. Therefore, the user will judge this *aca* as invalid. The node where it sits (enclosed within a double box in Fig. 1) has no children and thus

becomes buggy, leading to the diagnosis of `fDiff` as incomplete. This particular incompleteness symptom could be mended by placing the third rule for `fDiff` within the program.

3 Declarative Diagnosis of Missing Answers

As explained in the previous sections, the declarative diagnosis method proposed in this paper relies on building CTs as *abbreviated proof trees* w.r.t. a logically sound inference system for deriving *aca*s. In this section, we present such an inference system, whose *negative proof trees* represent the deduction of *aca*s from the *negative theory* $\mathcal{P}^-$ associated to a given $CFLP(\mathcal{D})$-program $\mathcal{P}$. We also present results ensuring the *logical correctness* of the declarative diagnosis method whose CTs are abbreviated representations of negative proof trees.

3.1 Standardized Programs and Negative Theories

Let $\mathcal{P}$ be a $CFLP(\mathcal{D})$-program. Its associated *Negative Theory* $\mathcal{P}^-$ is obtained in two steps. First, each program rule $f\ \overline{t}_n \to r \Leftarrow \Delta$ is replaced by a *standardized* form $f\ \overline{X}_n \to Y \Leftarrow \hat{R}$, where $\overline{X}_n, Y$ are new variables, $\hat{R} = \exists \overline{U}.\ R$ with $\overline{U} = var(R) \setminus \{\overline{X}_n, Y\}$, and the condition R is $X_1 \to t_1, \ldots, X_n \to t_n, \Delta, r \to Y$. Next, $\mathcal{P}^-$ is built by taking one *axiom* $(f)^-_{\mathcal{P}}$ of the form $\forall \overline{X}_n, Y.\ (f\ \overline{X}_n \to Y \Rightarrow (\bigvee_{i \in I} \hat{R}_i) \vee (\bot \to Y))$ for each function symbol f whose standardized program rules are $\{f\ \overline{X}_n \to Y \Leftarrow \hat{R}_i\}_{i \in I}$. By convention, we may use the notation D_f for the disjunction $(\bigvee_{i \in I} \hat{R}_i) \vee (\bot \to Y)$, and we may leave the universal quantification of the variables $\overline{X}_n, Y$ implicit. Intuitively, the axiom $(f)^-_{\mathcal{P}}$ says that any result computed for f must be obtained by means of some of the rules for f in the program. The last alternative $(\bot \to Y)$ within D_f says that Y is bound to the undefined result $\bot$ in case that no program rule for f succeeds to compute a more defined result. For example, let $\mathcal{P}_{\mathtt{fD}}$ be the $CFLP(\mathcal{H})$-program given in Section 2, with the third program rule for `fDiff` omitted. Then $\mathcal{P}^-_{\mathtt{fD}}$ includes (among others) the following axiom for the function symbol *fDiff*:

$$
\begin{aligned}
(\mathit{fDiff})^-_{\mathcal{P}_{\mathtt{fD}}} : \forall L, F.\ (\mathit{fDiff}\ L \to F \Rightarrow\ & \\
& \exists X.\ (L \to [X] \wedge X \to F) \vee \\
& \exists X, Y, Zs.\ (L \to (X : Y : Zs) \wedge X\ \mathtt{/=}\ Y \wedge X\ /\!/\ \mathit{fDiff}\,(Y : Zs) \to F) \vee \\
& (\bot \to F))
\end{aligned}
$$

Interpretations $\mathcal{I}$ are formally defined in [20]. Each interpretation represents a certain behavior of the program defined functions. We write $\mathcal{I} \Vdash_{\mathcal{D}} f\overline{t}_n \to t$ to indicate that the statement $f\overline{t}_n \to t$ is *valid* in $\mathcal{I}$. Here, f is a program defined function, $\overline{t}_n$ stand for possibly partially evaluated arguments, and t stands for a possibly partially evaluated result. Knowing the valid assertions $\mathcal{I} \Vdash_{\mathcal{D}} f\overline{t}_n \to t$ suffices for defining the *solution set* $Sol_{\mathcal{I}}(G)$ whose elements are all the *valuations* (i.e., substitutions of domain values for variables) that satisfy the goal G w.r.t. $\mathcal{I}$. We will use similar notations for other solution sets in the rest of the

paper, writing $Sol_{\mathcal{D}}$ instead of $Sol_{\mathcal{I}}$ whenever the solutions do not depend on the interpretation $\mathcal{I}$ of program defined functions. The following definition helps to understand the semantics of missing answers:

Definition 1 (Interpretation-Dependent Semantics). *Let $\mathcal{P}$ a $CFLP(\mathcal{D})$-program and $\mathcal{I}$ an interpretation over $\mathcal{D}$.*

1. *$\mathcal{I}$ is a* **model** *of $\mathcal{P}^-$ iff every axiom $(f)^-_{\mathcal{P}} : (f\ \overline{X}_n \rightarrow Y \Rightarrow D_f) \in \mathcal{P}^-$ satisfies $Sol_{\mathcal{I}}(f\ \overline{X}_n \rightarrow Y) \subseteq Sol_{\mathcal{I}}(D_f)$. When this inclusion holds, we say that $(f)^-_{\mathcal{P}}$ is* **valid** *in $\mathcal{I}$, or also that f's definition as given in $\mathcal{P}$ is* **complete** *w.r.t. $\mathcal{I}$.*
2. *The aca $G \Rightarrow \bigvee_{i\in I} \hat{S}_i$ is a* **logical consequence** *of $\mathcal{P}^-$ iff $Sol_{\mathcal{I}}(G) \subseteq \bigcup_{i\in I} Sol_{\mathcal{D}}(\hat{S}_i)$ for any model $\mathcal{I}$ of $\mathcal{P}^-$. When this happens, we also say that the disjunction of answers $\bigvee_{i\in I} \hat{S}_i$ is* **complete** *for G w.r.t. $\mathcal{P}$.*

3.2 Negative Proof Trees for Answer Collection Assertions

The declarative debugging of missing answers presupposes an *intended interpretation* of the program, starts with the observation of an *incompleteness symptom* and ends with an *incompleteness diagnosis*. A more precise definition of this *debugging scenario* is as follows:

Definition 2 (Debugging Scenario). *For any given $CFLP(\mathcal{D})$-program $\mathcal{P}$:*

1. *The* **intended interpretation** *is some interpretation $\mathcal{I}_{\mathcal{P}}$ over $\mathcal{D}$ which represents the behavior of the functions defined in $\mathcal{P}$ as expected by the programmer.*
2. *An* **incompleteness symptom** *occurs if the goal solving system computes finitely many solved goals $\{\hat{S}_i\}_{i\in I}$ as answers for an admissible initial goal G, and the programmer judges that $Sol_{\mathcal{I}_{\mathcal{P}}}(G) \not\subseteq \bigcup_{i\in I} Sol_{\mathcal{D}}(\hat{S}_i)$, meaning that the aca $G \Rightarrow \bigvee_{i\in I} \hat{S}_i$ is not valid in the intended interpretation $\mathcal{I}_{\mathcal{P}}$, so that some expected answers are missing.*
3. *An* **incompleteness diagnosis** *is given by pointing to some defined function symbol f such that the axiom $(f)^-_{\mathcal{P}}$ for f in $\mathcal{P}^-$ is not valid in $\mathcal{I}_{\mathcal{P}}$, which means $Sol_{\mathcal{I}_{\mathcal{P}}}(f\ \overline{X}_n \rightarrow Y) \not\subseteq Sol_{\mathcal{I}_{\mathcal{P}}}(D_f)$, showing that f's definition as given in $\mathcal{P}$ is incomplete w.r.t. $\mathcal{I}_{\mathcal{P}}$.*

Some concrete debugging scenarios have been discussed in Section 2 and [9]. Assume now that an incompleteness symptom has been observed by the programmer. Since the goal solving system has computed the disjunction of answers $D = \bigvee_{i\in I} \hat{S}_i$, the *aca* $G \Rightarrow D$ asserting that the computed answers cover all the solutions of G should be derivable from $\mathcal{P}^-$. The *Constraint Negative Proof Calculus* $CNPC(\mathcal{D})$ consisting of the inference rules displayed in Fig. 2 has been designed with the aim of enabling logical proofs $\mathcal{P}^- \vdash_{CNPC(\mathcal{D})} G \Rightarrow D$ of *acas*. We use a special operator & in order to express the result of attaching to a given goal G a solved goal $\hat{S}'$ resulting from a previous computation, so that computation can continue from the new goal $G\ \&\ \hat{S}'$.

SF Solved Form $\dfrac{}{R \;\square\; S \Rightarrow D}$ if $Sol_{\mathcal{D}}(S) \subseteq Sol_{\mathcal{D}}(D)$.

CJ Conjunction

$$\frac{R_1 \;\square\; S \Rightarrow \bigvee_{i\in I} \exists \overline{Z}_i.\; S_i \quad \ldots\; (\hat{R}_2 \;\&\; \hat{S}_i) \Rightarrow \bigvee_{j\in J_i} \exists \overline{Z}_{ij}.\; S_{ij} \;\ldots\; (i \in I)}{(R_1 \wedge R_2) \;\square\; S \Rightarrow \bigvee_{i\in I} \bigvee_{j\in J_i} \exists \overline{Z}_i, \overline{Z}_{ij}.\; S_{ij}}$$

if $\overline{Z}_i \notin var((R_1 \wedge R_2) \;\square\; S)$, $\overline{Z}_{ij} \notin var((R_1 \wedge R_2) \;\square\; S) \cup \overline{Z}_i$, for all $i \in I, j \in J_i$.

TS Trivial Statement $\dfrac{}{\varphi : G \Rightarrow D}$

if φ is a trivial *aca* s.t. $Sol(G) \subseteq Sol_{\mathcal{D}}(D)$.

DC DeComposition $\dfrac{\overline{e_m \to t_m} \;\square\; S \Rightarrow D}{h\overline{e}_m \to h\overline{t}_m \;\square\; S \Rightarrow D}$ if $h\overline{e}_m$ is not a pattern.

IM IMitation $\dfrac{\overline{e_m \to X_m} \;\square\; (S \wedge h\overline{X}_m \to X) \Rightarrow \bigvee_{i\in I} \exists \overline{Z}_i.\; S_i}{h\overline{e}_m \to X \;\square\; S \Rightarrow \bigvee_{i\in I} \exists \overline{X}_m, \overline{Z}_i.\; S_i}$

if $h\overline{e}_m$ is not a pattern, $X \in \mathcal{V}$, and $\overline{X}_m \notin var(h\overline{e}_m \to X \;\square\; S)$.

$(\mathbf{AR})_p$ Argument Reduction for Primitive Functions

$$\frac{\overline{e_n \to X_n} \;\square\; (S \wedge p\overline{X}_n \to!\; t) \Rightarrow \bigvee_{i\in I} \exists \overline{Z}_i.\; S_i}{p\overline{e}_n \to^{?} t \;\square\; S \Rightarrow (S \wedge \bot \to t) \vee (\bigvee_{i\in I} \exists \overline{X}_n, \exists \overline{Z}_i.\; S_i)}$$

if $p \in PF^n, \overline{X}_n \notin var(p\overline{e}_n \to^{?} t \;\square\; S)$, and $\to^{?} \equiv \to$ (*production*) $\cup \to!$ (*constraint*). For instance, *equality constraints* e_1 `==` e_2 (resp., *disequality constraints* e_1 `/=` e_2). are abbreviations of e_1 `==` $e_2 \to!\; true$ (resp., e_1 `==` $e_2 \to!\; false$).

$(\mathbf{AR})_f$ Argument Reduction for Defined Functions

$$\frac{(\overline{e_n \to X_n} \wedge f\overline{X}_n \to t) \;\square\; S \Rightarrow \bigvee_{i\in I} \exists \overline{Z}_i.\; S_i}{f\overline{e}_n \to t \;\square\; S \Rightarrow \bigvee_{i\in I} \exists \overline{X}_n, \overline{Z}_i.\; S_i}$$

if $f \in DF^n$, and $\overline{X}_n \notin var(f\overline{e}_n \to t \;\square\; S)$.

$$\frac{(\overline{e_n \to X_n} \wedge f\overline{X}_n \to Y \wedge Y\overline{a}_k \to t) \;\square\; S \Rightarrow \bigvee_{i\in I} \exists \overline{Z}_i.\; S_i}{f\overline{e}_n\overline{a}_k \to t \;\square\; S \Rightarrow \bigvee_{i\in I} \exists \overline{X}_n, Y, \overline{Z}_i.\; S_i}$$

if $f \in DF^n$ $(k > 0)$, and $\overline{X}_n, Y \notin var(f\overline{e}_n\overline{a}_k \to t \;\square\; S)$.

$(\mathbf{DF})_f$ Defined Function $\dfrac{\ldots\; R_i[\overline{X}_n \mapsto \overline{t}_n, Y \mapsto t] \;\square\; S \Rightarrow D_i \;\ldots\; (i \in I)}{\boxed{f\overline{t}_n \to t \;\square\; S \Rightarrow (S \wedge \bot \to t) \vee (\bigvee_{i\in I} D_i)}}$

if $f \in DF^n$, $\overline{X}_n, Y \notin var(f\overline{t}_n \to t \;\square\; S)$, and $(f\overline{X}_n \to Y \Rightarrow \bigvee_{i\in I} \hat{R}_i) \in \mathcal{P}^-$.

Fig. 2. The Constraint Negative Proof Calculus $CNPC(\mathcal{D})$

Formally, assuming $G = \exists\overline{U}.\ (R \sqcap (\Pi \sqcap \sigma))$ and $\hat{S}' = \exists\overline{U}'.\ (\Pi' \sqcap \sigma')$ a solved goal such that $\overline{U} \setminus dom(\sigma') \subseteq \overline{U}'$, $\sigma\sigma' = \sigma'$ and $Sol_{\mathcal{D}}(\Pi') \subseteq Sol_{\mathcal{D}}(\Pi\sigma')$, the operation $G \ \&\ \hat{S}'$ is defined as $\exists\overline{U}'.\ (R\sigma' \sqcap (\Pi' \sqcap \sigma'))$. The inference rule **CJ** infers an *aca* for a goal with composed kernel $(R_1 \wedge R_2) \sqcap S$ from *aca*s for goals with kernels of the form $R_1 \sqcap S$ and $(\hat{R}_2 \ \&\ \hat{S}_i)$, respectively; while other inferences deal with different kinds of atomic goal kernels.

Any $CNPC(\mathcal{D})$-derivation $\mathcal{P}^- \vdash_{CNPC(\mathcal{D})} G \Rightarrow D$ can be depicted in the form of a *Negative Proof Tree* over $\mathcal{D}$ (shortly, NPT) with *aca*s at its nodes, such that the *aca* at any node is inferred from the *acas* at its children using some $CNPC(\mathcal{D})$ inference rule. We say that a goal solving system for $CFLP(\mathcal{D})$ is *admissible* iff whenever finitely many solved goals $\{\hat{S}_i\}_{i\in I}$ are computed as answers for an admissible initial goal G, one has $\mathcal{P}^- \vdash_{CNPC(\mathcal{D})} G \Rightarrow \bigvee_{i\in I} \hat{S}_i$ with some witnessing NPT. The next theorem is intended to provide some plausibility to the pragmatic assumption that actual $CFLP$ systems such as *Curry* [16] or $\mathcal{TOY}$ [21] are admissible goal solving systems.

Theorem 1 (Existence of Admissible Goal Solving Calculi). *There is an admissible Goal Solving Calculus $GSC(\mathcal{D})$ which formalizes the goal solving methods underlying actual $CFLP$ systems such as Curry or $\mathcal{TOY}$.*

Proof. A more general result can be proved, namely: If $(\widehat{\underline{R} \wedge R'})\ \&\ \hat{S} \not\sim^{p}_{\mathcal{P},GSC(\mathcal{D})} D$ (with a partially developed search space of finite size p built using the program $\mathcal{P}$, a *Goal Solving Calculus* $GSC(\mathcal{D})$ inspired in [19,11], and a certain selection strategy that only selects atoms descendants of the part R) then $\mathcal{P}^- \vdash_{CNPC(\mathcal{D})} \hat{R}\ \&\ \hat{S} \Rightarrow D$ with some witnessing NPT. The proof proceeds by induction of p, using an auxiliary lemma to deal with compound goals whose kernel is a conjunction. Details are given in [10]. □

We have also proved in [10] the following theorem, showing that any *aca* which has been derived by means of a NPT is a logical consequence of the negative theory associated to the corresponding program. This result will be used below for proving the correctness of our diagnosis method.

Theorem 2 (Semantic Correctness of the $CNPC(\mathcal{D})$ Calculus). *Let $G \Rightarrow D$ be any aca for a given $CFLP(\mathcal{D})$-program $\mathcal{P}$. If $\mathcal{P}^- \vdash_{CNPC(\mathcal{D})} G \Rightarrow D$ then $G \Rightarrow D$ is a logical consequence of $\mathcal{P}^-$ in the sense of Definition 1.*

3.3 Declarative Diagnosis of Missing Answers Using Negative Proof Trees

We are now prepared to present a declarative diagnosis method for missing answers which is based on NPTs and leads to correct diagnosis for any admissible goal solving system. First, we show that incompleteness symptoms are caused by incomplete program rules. This is guaranteed by the following theorem:

Theorem 3 (Missing Answers are Caused by Incomplete Program Rules). *Assume that an incompleteness symptom has been observed for a given $CFLP(\mathcal{D})$-program $\mathcal{P}$ as explained in Definition 2, with intended interpretation $\mathcal{I}_{\mathcal{P}}$, admissible initial goal G, and finite disjunction of computed answers $D = \bigvee_{i \in I} \hat{S}_i$. Assume also that the computation has been performed by an admissible goal solving system. Then there exists some defined function symbol f such that the axiom $(f)^{-}_{\mathcal{P}}$ for f in $\mathcal{P}^{-}$ is not valid in $\mathcal{I}_{\mathcal{P}}$, so that f's definition as given in $\mathcal{P}$ is incomplete w.r.t. $\mathcal{I}_{\mathcal{P}}$.*

Proof. Because of the admissibility of the goal solving system, we can assume $\mathcal{P}^{-} \vdash_{CNPC(\mathcal{D})} G \Rightarrow D$. Then the *aca* $G \Rightarrow D$ is a logical consequence of $\mathcal{P}^{-}$ because of Theorem 2. By Definition 1, we conclude that $Sol_{\mathcal{I}}(G) \subseteq Sol_{\mathcal{D}}(D)$ holds for any model $\mathcal{I}$ of $\mathcal{P}^{-}$. However, we also know that $Sol_{\mathcal{I}_{\mathcal{P}}}(G) \not\subseteq Sol_{\mathcal{D}}(D)$, because the disjunction D of computed answers is an incompleteness symptom w.r.t. $\mathcal{I}_{\mathcal{P}}$. Therefore, we can conclude that $\mathcal{I}_{\mathcal{P}}$ is not a model of $\mathcal{P}^{-}$, and therefore the completeness axiom $(f)^{-}_{\mathcal{P}}$ of some defined function symbol f must be invalid in $\mathcal{I}_{\mathcal{P}}$. □

The previous theorem does not yet provide a practical method for finding an incomplete function definition. As explained in Section 2, a declarative diagnosis method is expected to find the incomplete function definition by inspecting a CT. We propose to use abbreviated NPTs as CTs. Note that $\mathbf{(DF)}_f$ is the only inference rule in the $CNPC(\mathcal{D})$ calculus that depends on the program, and all the other inference rules are correct w.r.t. arbitrary interpretations. For this reason, abbreviated proof trees will omit the inference steps related to the $CNPC(\mathcal{D})$ inference rules other than $\mathbf{(DF)}_f$. More precisely, given a NPT $\mathcal{T}$ witnessing a $CNPC(\mathcal{D})$ proof $\mathcal{P}^{-} \vdash_{CNPC(\mathcal{D})} G \Rightarrow D$, its associated *Abbreviated Negative Proof Tree* (shortly, $ANPT$) $\mathcal{AT}$ is constructed as follows:

(1) The root of $\mathcal{AT}$ is the root of $\mathcal{T}$.
(2) The children of any node N in $\mathcal{AT}$ are the closest descendants of N in $\mathcal{T}$ corresponding to *boxed acas* introduced by $\mathbf{(DF)}_f$ inference steps.

As already explained, declarative diagnosis methods search a given CT looking for a *buggy node* whose result is unexpected but whose children's results are all expected. In our present setting, the CTs are $ANPT$s, the "results" attached to nodes are *aca*s, and a given node N is *buggy* iff the *aca* at N is *invalid* (i.e., it represents an incomplete recollection of computed answers in the intended interpretation $\mathcal{I}_{\mathcal{P}}$) while the *aca* at each children node N_i is *valid* (i.e., it represents a complete recollection of computed answers in the intended interpretation $\mathcal{I}_{\mathcal{P}}$).

As a concrete example, Fig. 3 displays a NPT which can be used for the diagnosis of missing answers in the example presented in Section 2. Buggy nodes are highlighted by encircling the *aca*s attached to them within double boxes. The CT shown in Fig. 1 is the $ANPT$ constructed from this NPT.

Our last result is a refinement of Theorem 3. It guarantees that declarative diagnosis with $ANPT$s used as CTs leads to the correct detection of incomplete program functions. A proof can be found in [10].

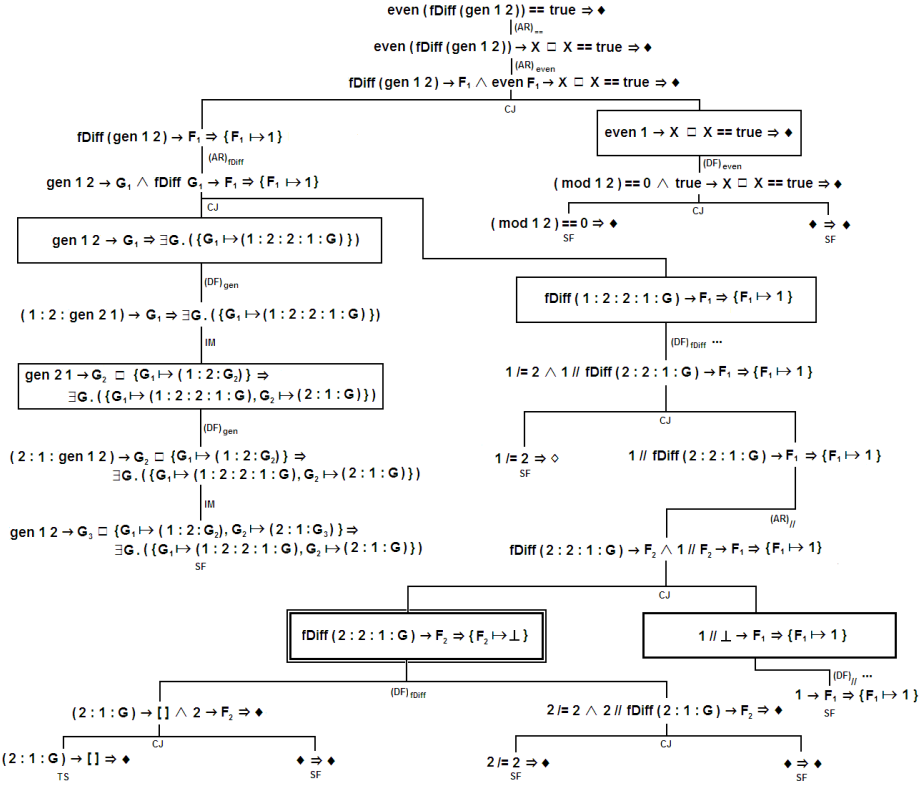

Fig. 3. *NPT* for the declarative diagnosis of missing answers

Theorem 4 (***ANPT*s Lead to the Diagnosis of Incomplete Functions**). *As in Theorem 3, assume that an incompleteness symptom has been observed for a given CFLP($\mathcal{D}$)-program $\mathcal{P}$ as explained in Definition 2, with intended interpretation $\mathcal{I}_{\mathcal{P}}$, admissible initial goal G, and finite disjunction of answers $D = \bigvee_{i \in I} \hat{S}_i$, computed by an admissible goal solving system. Then $\mathcal{P}^- \vdash_{CNPC(\mathcal{D})} G \Rightarrow D$, and the ANPT constructed from any NPT witnessing this derivation, has some buggy node. Moreover, each such buggy node points to an axiom $(f)^-_{\mathcal{P}}$ which is incomplete w.r.t. the user's intended interpretation $\mathcal{I}_{\mathcal{P}}$.*

4 Implementation in the *TOY* System

In this section, we discuss the implementation in the $\mathcal{TOY}$ system of a tool based on the debugging method presented in the previous sections. The current prototype only supports the Herbrand constraint domain $\mathcal{H}$, although the same principles can be applied to other constraint domains $\mathcal{D}$.

We summarize first the normal process followed by the $\mathcal{TOY}$ system when compiling a source program $\mathcal{P}.toy$ and solving an initial goal G w.r.t. $\mathcal{P}$. During

the compilation process the system translates a source program $\mathcal{P}.toy$ into a Prolog program $\mathcal{P}.pl$ including a predicate for each function in $\mathcal{P}$. For instance the function `even` of our running example is transformed into a predicate

```
even(N,R,IC,OC):- ... code for even ... .
```

where the variable `N` corresponds to the input parameter of the function, `R` to the function result, and `IC, OC` represent, respectively, the input and output constraint store. Moreover, each goal G of $\mathcal{P}$ is also translated into a Prolog goal and solved w.r.t. $\mathcal{P}.pl$ by the underlying Prolog system. The result is a collection of answers which are presented to the user in a certain sequence, as a result of Prolog's backtracking.

If the computation of answers for G finishes after having collected finitely many answers, the user may decide that there are some missing answers (*incompleteness symptom*, in the terminology of Definition 2) and type the command `/missing` at the system prompt in order to initiate a *debugging session*. The debugger proceeds carrying out the following steps:

1. The object program $\mathcal{P}.pl$ is transformed into a new Prolog program $\mathcal{P}^{\mathcal{I}}.pl$. The debugger can safely assume that $\mathcal{P}.pl$ already exists because the tool is always initiated *after* some missing answer has been detected by the user. The transformed program $\mathcal{P}^{\mathcal{I}}$ behaves almost identically to $\mathcal{P}$, the only difference being that it produces a suitable *trace* of the computation in a text file. For instance here is a fragment of the code for the function `even` of our running example in the transformed program:

```
 1 % this clause wraps the original predicate
 2 even(N,R,IC,OC):-
 3     % display the input values for even
 4     write(' begin('), write(' even,'), writeq(N), write(','),
 5     write(R), write(', '), writeq(IC), write(').'), nl,
 6     % evenBis corresponds to the original predicate for even
 7     evenBis(N,R,IC,OC),
 8     % display an output result
 9     write(' output('), write(' even,'), writeq(N), write(','),
10     write(R), write(', '), writeq(OC), write(').'), nl.

11 % when all the possible outputs of the function have been produced
12 even(N,R,IC,OC):-
13     nl, write(' end(even).'), nl,
14     !,
15     fail.
16 evenBis(N,R,IC,OC) :- ... original code for even ... .
```

   As the example shows, the code for each function now displays information about the values of the arguments and the contents of the constraint store at the moment of using any user defined function (lines 4-5). Then the predicate corresponding to the original function, now renamed with the `Bis` suffix, is called (line 7). After any successful function call the trace displays again

the values of the arguments and result, which may have changed, and the contents of the output constraint store (lines 9, 10). A second clause (lines 12-15) displays the value `end` when the function has exhausted its possible output. The clause fails in order to ensure that the program flow is not changed. The original code for each function is kept unaltered in the transformed program except for the renaming (`evenBis` instead of `even` in the example, line 16). This ensures that the program will behave equivalently to the original program, except for the trace produced as a side-effect.

2. In order to obtain the trace file, the debugger repeats the computation of all the answers for the goal G w.r.t. $\mathcal{P}^{\mathcal{T}}$. After each successful computation the debugger enforces a `fail` in order to trigger the backtracking mechanism and produce the next solution for the goal. The program output is redirected to a file, where the trace is stored.
3. The trace file is then analyzed by the *CT builder* module of the tool. The result is the *Computation Tree* (an *ANPT*), which is displayed by a *Java graphical interface*.
4. The tree can be navigated by the user either manually, providing information about the validity of the *aca*s contained in the tree, or using any of the automatic strategies included in the tool which try to minimize the number of nodes that the user must examine (see [29] for a description of some strategies and their efficiency). The process ends when a buggy node is found and the tool points to an incomplete function definition, as explained in Section 3, as responsible for the missing answers. The current implementation of the prototype is available at `http://toy.sourceforge.net`. The generation of trace files works satisfactorily, while the *CT* builder module and the Java graphical interface do still need more improvements.

Fig. 4 shows how the tool displays the *CT* corresponding to the debugging scenario discussed in Section 2. The initial goal is not displayed, but the rest of the *CT* corresponds to Fig. 1, whose construction as *ANPT* has been explained in Section 3. When displaying an *aca* $f\,\overline{t}_n \rightarrow t \,\square\, S \Rightarrow \bigvee_{i\in I} \hat{S}_i$, the tool uses list notation for representing the disjunction $\bigvee_{i\in I} \hat{S}_i$ and performs some simplifications: useless variable bindings within the stores S and S_i are dropped, as in the *aca* displayed as `gen 2 1 -> A ==> [A = 2:1:_]` in Fig. 4; and if t happens to be a variable X, the case $\{X \mapsto \bot\}$ is omitted from the disjunction $\bigvee_{i\in I} \hat{S}_i$, so that the user must interpret the *aca* as collecting the possible results for X other than the undefined value $\bot$. The tool also displays the underscore symbol `_` at some places. Within any *aca*, the occurrences of `_` at the right hand side of the implication $\Rightarrow$ must be understood as different existentially quantified variables, while each occurrence of `_` at the left hand side of $\Rightarrow$ must be understood as $\bot$. For instance, `1 // _ -> A ==> [A = 1]` is the *aca* `1 //` $\bot \rightarrow A \Rightarrow \{A \mapsto 1\}$ as displayed by the tool. Understanding the occurrences of `_` at the left hand side of $\Rightarrow$ as different universally quantified variables would be incorrect. For instance, the *aca* `1 //` $\bot \rightarrow A \Rightarrow \{A \mapsto 1\}$ is valid w.r.t. the intended interpretation $\mathcal{I}_{\mathcal{P}_{\mathtt{fD}}}$ of $\mathcal{P}_{\mathtt{fD}}$, while the statement $\forall X.\ (\mathtt{1\ //}\ X \rightarrow A \Rightarrow \{A \mapsto 1\})$ has a different meaning and is not valid in $\mathcal{I}_{\mathcal{P}_{\mathtt{fD}}}$.

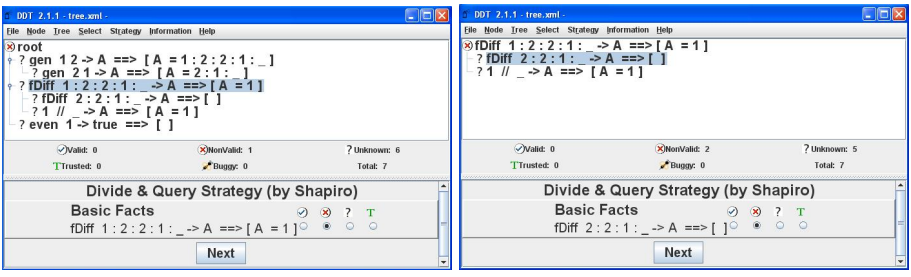

Fig. 4. Snapshots of the prototype

In the debugging session shown in Fig. 4 the user has selected the *Divide & Query* strategy [29] in order to find a buggy node. The lower part of the left-hand side snapshot shows the first question asked by the tool after selecting this strategy, namely the *aca* `fDiff 1:2:2:1:_ -> A ==> [A=1]`. According to her knowledge of $\mathcal{I}_{\mathcal{P}_{\texttt{fD}}}$ the user marks this *aca* as invalid. The strategy now prunes the CT keeping only the subtree rooted by the invalid *aca* at the previous step (every CT with an invalid root must contain at least one buggy node). The second question, which can be seen at the right-hand side snapshot, asks about the validity of the *aca* `fDiff 2:2:1:_ -> A ==> []` (which in fact represents `fDiff 2:2:1:`$\bot \rightarrow A \Rightarrow \{A \mapsto \bot\}$, as explained above). Again, her knowledge of $\mathcal{I}_{\mathcal{P}_{\texttt{fD}}}$ leads the user to expect that `fDiff 2:2:1:`$\bot$ can return some defined result, and the *aca* is marked as invalid. After this question the debugger points out at `fDiff` as an incomplete function, and the debugging session ends. Regarding the efficiency of this debugging method our preliminary experimental results show that:

1. Producing the transformed $\mathcal{P}^{\mathcal{T}}$. *pl* from $\mathcal{P}$.*pl* is proportional in time to the number of functions of the program, and does require an insignificant amount of system memory since each predicate is transformed separately.
2. The computation of the goal w.r.t. $\mathcal{P}^{\mathcal{T}}$. *pl* requires almost the same system resources as w.r.t. $\mathcal{P}$.*pl* because writing the trace causes no significant overhead in our experiments.
3. Producing the CT from the trace is not straightforward and requires several traverses of the trace. Although more time-consuming due to the algorithmic difficulty, this process only keeps portions of the trace in memory at each moment.
4. The most inefficient phase in our current implementation is the graphical interface. Although it would be possible to keep in memory only the portion of the tree displayed at each moment, our graphical interface loads the whole CT in main memory. We plan to improve this limitation in the future. However the current prototype can cope with CTs containing thousands of nodes, which is enough for medium size computations.
5. As usual in declarative debugging, the efficiency of the tool depends on the computation tree size, which in turn usually depends on the size of the data structures required and not on the program size.

A different issue is the difficulty of answering the questions by the user. Indeed in complicated programs involving constraints the *aca*s can be large and intricate, as it is also the case with other debugging tools for CLP languages. Nevertheless, our prototype works reasonably in cases where the goal's search space is relatively small, and we believe that working with such goals can be useful for detecting many programming bugs in practice. Techniques for simplifying CTs should be worked out in future improvements of the prototype. For instance, asking the user for a concrete missing instance of the initial goal and starting a diagnosis session for the instantiated goal might be helpful.

5 Conclusions and Future Work

We have presented a novel method for the declarative diagnosis of *missing computed answers* in $CFLP(\mathcal{D})$, a declarative programming scheme which combines the expressivity of lazy FP and CLP languages. The method relies on *Computation Trees* (CTs) whose nodes are labeled with *answer collection assertions* (*aca*s). As in declarative diagnosis for FP languages, the values displayed at *aca*s are shown in the most evaluated form demanded by the topmost computation. On the other hand, and following the CLP tradition, we have shown that our CTs are abbreviated proof trees in a suitable inference system, the so-called *constraint negative proof calculus*. Thanks to this fact, we can prove the correctness of our diagnosis method for any admissible goal solving system whose recollection of computed answers can be represented by means of a proof tree in the constraint negative proof calculus. As far as we know, no comparable result was previously available for such an expressive framework as $CFLP$.

Intuitively, the notion of *aca* bears some loose relationship to programming techniques related to answer recollection, as e.g., *encapsulated search* [2]. However, *aca*s in our setting are not a programming technique. Rather, they serve as logical statements whose falsity reveals incompleteness of computed answers w.r.t. expected answers. In principle, one could also think of a kind of logical statements somewhat similar to *aca*s, but asserting the *equality* of the observed and expected sets of computed answers for one and the same goal with a finite search space. We have not developed this idea, which could support the declarative diagnosis of a third kind of unexpected results, namely *incorrect answer sets* as done for *Datalog* [5]. In fact, we think that a separate diagnosis of wrong and missing answers is pragmatically more convenient for users of $CFLP$ languages.

On the practical side, our method can be applied to actual $CFLP$ systems such as *Curry* or $\mathcal{TOY}$, leading to correct diagnosis under the pragmatic assumption that they behave as admissible goal solving systems. This assumption is plausible in so far as the systems are based on formal goal solving procedures that can be argued to be admissible. A prototype debugger under development is available, which implements the method in $\mathcal{TOY}$. Although our implementation is based on the ad-hoc trace generated by the transformed program $\mathcal{P}^{\mathcal{T}}$, we think that it could be possible to obtain the CTs from the *redex trail* for functional-logic

programming described in [3]. This would allow reasoning about the correctness of the implementation by using the declarative semantics supporting this structure.

Some important pragmatic problems well known for declarative diagnosis tools in FP and CLP languages also arise in our context: both the CTs and the *aca*s at their nodes may be very big in general, causing computation overhead and difficulties for the user in answering the questions posed by the debugging tool. In spite of these difficulties, the prototype works reasonably in cases where the goal's search space is relatively small, and we believe that working with such goals can be useful for detecting many programming bugs in practice. Techniques for simplifying CTs should be worked out in future improvements of the prototype.

Acknowledgments

The authors are grateful to the referees of previous versions of this paper for their constructive comments and suggestions.

References

1. Boye, J., Drabent, W., Maluszynski, J.: Declarative diagnosis of contraint programs: An assertion-based approach. In: Automated and Algorithmic Debugging, pp. 123–140 (1997)
2. Brassel, B., Hanus, M., Huch, F.: Encapsulating non-determinism in functional logic computations. Journal of Functional and Logic Programming (2004)
3. Brassel, B., Hanus, M., Huch, F., Vidal, G.: A semantics for tracing declarative multi-paradigm programs. In: PPDP 2004, pp. 179–190. ACM Press, New York (2004)
4. Caballero, R.: A declarative debugger of incorrect answers for constraint functional-logic programs. In: WCFLP 2005, pp. 8–13. ACM Press, New York (2005)
5. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A new proposal for debugging datalog programs. In: WFLP 2007 (2007)
6. Caballero, R., Rodríguez-Artalejo, M.: A declarative debugging system for lazy functional logic programs. Electr. Notes Theor. Comput. Sci. 64 (2002)
7. Rodríguez-Artalejo, M., Caballero, R.: $\mathcal{DDT}$: A declarative debugging tool for functional-logic languages. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS 2004. LNCS, vol. 2998, pp. 70–84. Springer, Heidelberg (2004)
8. Caballero, R., Rodríguez-Artalejo, M., del Vado-Vírseda, R.: Declarative diagnosis of wrong answers in constraint functional-logic programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 421–422. Springer, Heidelberg (2006)
9. Caballero, R., Rodríguez-Artalejo, M., del Vado-Vírseda, R.: Declarative debugging of missing answers in constraint functional-logic programming. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 425–427. Springer, Heidelberg (2007)
10. Caballero, R., Rodríguez-Artalejo, M., del Vado-Vírseda, R.: Algorithmic debugging of missing answers in constraint functional-logic programming. Technical Report DSIC 2/08, Universidad Complutense de Madrid (2008), `http://gpd.sip.ucm.es/papers.html`

11. del Vado-Vírseda, R.: Declarative constraint programming with definitional trees. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 184–199. Springer, Heidelberg (2005)
12. Estévez, S., del Vado-Vírseda, R.: Designing an efficient computation strategy in $CFLP(\mathcal{FD})$ using definitional trees. In: WCFLP 2005, pp. 23–31. ACM Press, New York (2005)
13. Fernández, A.J., Hortalá-González, M.T., Sáenz-Pérez, F., del Vado-Vírseda, R.: Constraint functional logic programming over finite domains. Theory and Practice of Logic Programming 7(5), 537–582 (2007)
14. Ferrand, G.: Error diagnosis in logic programming, an adaption of E. Y. Shapiro's method. J. Log. Program. 4(3), 177–198 (1987)
15. Ferrand, G., Lesaint, W., Tessier, A.: Towards declarative diagnosis of constraint programs over finite domains. ArXiv Computer Science e-prints (2003)
16. Hanus, M.: Curry: An integrated functional logic language (version 0.8.2 of march 28, 2006) (2006), `http://www.informatik.uni-kiel.de/~curry`
17. Hermenegildo, M., Puebla, G., Bueno, F., López-García, P.: Abstract verification and debugging of constraint logic programs. In: O'Sullivan, B. (ed.) CologNet 2002. LNCS (LNAI), vol. 2627, pp. 1–14. Springer, Heidelberg (2003)
18. Lloyd, J.W.: Declarative error diagnosis. New Gen. Comput. 5(2), 133–154 (1987)
19. López-Fraguas, F.J., Rodríguez-Artalejo, M., del Vado-Vírseda, R.: A lazy narrowing calculus for declarative constraint programming. In: PPDP 2004, pp. 43–54. ACM Press, New York (2004)
20. López-Fraguas, F.J., Rodríguez-Artalejo, M., Vado-Vírseda, R.d.: A new generic scheme for functional logic programming with constraints. Higher-Order and Symbolic Computation 20(1-2), 73–122 (2007)
21. López-Fraguas, F.J., Sánchez-Hernández, J.: $\mathcal{TOY}$: A multiparadigm declarative system. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
22. Naish, L.: A declarative debugging scheme. Journal of Functional and Logic Programming 1997(3) (1997)
23. Naish, L., Barbour, T.: A declarative debugger for a logical-functional language. DSTO General Document 5(2), 91–99 (1995)
24. Nilsson, H.: How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. J. Funct. Program. 11(6), 629–671 (2001)
25. Nilsson, H., Sparud, J.: The evaluation dependence tree as a basis for lazy functional debugging. Autom. Softw. Eng. 4(2), 121–150 (1997)
26. B. Pope. *A Declarative Debugger for Haskell.* PhD thesis, Department of Computer Science and Software Engineering, University of Melbourne (2006)
27. Pope, B., Naish, L.: Practical aspects of declarative debugging in haskell 98. In: PPDP 2003, pp. 230–240. ACM Press, New York (2003)
28. Shapiro, E.Y.: Algorithmic Program Debugging. MIT Press, Cambridge, MA, USA (1983)
29. Silva, J.: A comparative study of algorithmic debugging strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)
30. Tessier, A., Ferrand, G.: Declarative diagnosis in the *CLP* scheme. In: Deransart, P., Małuszyński, J. (eds.) DiSCiPl 1999. LNCS, vol. 1870, pp. 151–174. Springer, Heidelberg (2000)