A Semantic Framework for the Declarative Debugging of Wrong and Missing Answers in Declarative Constraint Programming

Rafael del Vado Vírseda and Fernando Pérez Morente
Universidad Complutense de Madrid
Spain

1. Introduction

Debugging tools 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. As a solution to this problem, and following a seminal idea by Shapiro (Shapiro, 1982), *declarative debugging* (a.k.a. *declarative diagnosis* or *algorithmic debugging*) uses *Computation Trees* (shortly, *CT*s) in place of traces. *CT*s 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. Each node in a *CT* represents the computation of some observable result, depending on the results of its 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 (Naish, 1997). Declarative diagnosis was first proposed in the field of *Logic Programming* (*LP*) (Ferrand, 1987; Lloyd, 1987; Shapiro, 1982), and it has been successfully extended to other declarative programming paradigms, including (lazy) *Functional Programming* (*FP*) (Nilsson, 2001; Nilsson & Sparud, 1997; Pope, 2006; Pope & Naish, 2003), *Constraint Logic Programming* (*CLP*) (Boye et al., 1997; Ferrand et al., 2003; Tessier & Ferrand, 2000), and *Functional-Logic Programming* (*FLP*) (Caballero & Rodríguez, 2004; Naish & Barbour, 1995). 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 debugging 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 most 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 recent approaches to error diagnosis using *abstract interpretation* (e.g., (Alpuente et al., 2003; Comini et al., 1999; Hermenegildo, 2002), and some of the approaches described in (Deransart et al., 2000)), 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 (Boye et al., 1997), safe inference of information from answers previously given by the user (Caballero & Rodríguez, 2004), or *CT*s tailored to the needs of a particular debugging problem over a particular computation domain (Ferrand et al., 2003). Another practical problem with declarative diagnosis is that the size of *CT*s 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 *CT*s have been considered; see (Pope, 2006) for a recent proposal in the *FP* field. However, current research in declarative diagnosis has still to face many challenges regarding both the foundations and the development of practical tools.

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. The aim of this chapter is to present a logical and semantic framework for diagnosing wrong and missing computed answers in $CFLP(\mathcal{D})$ (López et al., 2006), a newly proposed generic programming 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 (López et al., 2004). Moreover, useful instances of this scheme have been implemented in the $\mathcal{TOY}$ system (López & Sánchez, 1999) and tested in practical applications (Fernández et al., 2007). Borrowing ideas from $CFLP(\mathcal{D})$ declarative semantics we obtain a suitable notion of *intended interpretation*, as well as a kind of abridged *proof trees* with a sound logical meaning to play the role of *CT*s. Our aim is to achieve a natural combination of previous approaches that were independently developed for the $CLP(\mathcal{D})$ scheme (Tessier & Ferrand, 2000) and for lazy functional-logic languages (Caballero & Rodríguez, 2004). We give theoretical results showing that the proposed debugging method is logically correct for any sound $CFLP(\mathcal{D})$-system whose computed answers are logical consequences of the program in the sense of $CFLP(\mathcal{D})$ semantics. We also present a practical debugger called $\mathcal{DDT}$, developed as an extension of previously existing but less powerful tools (Caballero, 2005; Caballero & Rodríguez, 2004). $\mathcal{DDT}$ implements the proposed diagnosis method for $CFLP(\mathcal{R})$-programming in the $\mathcal{TOY}$ system (López & Sánchez, 1999) using the domain $\mathcal{R}$ of arithmetic constraints over the real numbers.

The rest of the chapter is organized as follows: Section 2 motivates our approach by presenting debugging examples which are used as illustration of the main features of our diagnosis method. Section 3 recalls the $CFLP(\mathcal{D})$ scheme from (López et al., 2006) to the extent needed for understanding the theoretical results in this chapter. Section 4 presents a correct method for the declarative diagnosis of wrong computed answers in any soundly implemented $CFLP(\mathcal{D})$-system. Section 5 describes the debugging tool $\mathcal{DDT}$ for wrong answers. Section 6 presents the abbreviated proof trees used as *CT*s in our method for debugging missing computed answers, as well as the results ensuring the logical correctness of the diagnosis. Section 7 presents a prototype debugger for diagnosing missing answers. Section 8 concludes and points to some plans for future work.

2. Motivating examples

As a motivation for our declarative debugging method of wrong answers in the *CFLP*$(\mathcal{D})$ scheme, we consider the following program fragment written in $\mathcal{TOY}$ (López & Sánchez, 1999), a programming system which supports several instances of the *CFLP*$(\mathcal{D})$ scheme:

Example 1. (Debugging Wrong Answers in $\mathcal{TOY}$)

```
infixr 40 &&
(&&) :: bool -> bool -> bool
false && Y = false
true && Y = Y

head :: [A] -> A
head [X|Xs] = X

type point = (real,real)
type figure = point -> bool

rect :: point -> real -> real -> figure
rect (X,Y) LX LY (X',Y') = (X' >= X)&&(X' <= X+LX)&&(Y' <= Y)&&(Y'<=Y+LY)

% This program rule is incorrect. It should be: (Y' >= Y)...

intersect :: figure -> figure -> figure
intersect F1 F2 P = F1 P && F2 P

ladder :: point -> real -> real -> [figure]
ladder (X,Y) LX LY = [rect (X,Y) LX LY | ladder (X+LX, Y+LY) LX LY]
```

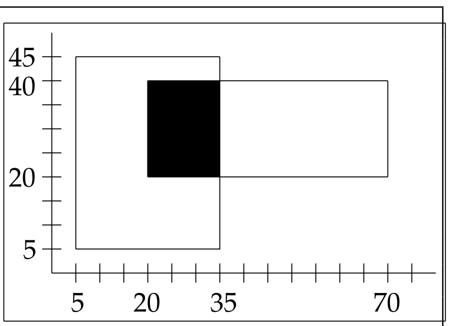


Fig. 1. Building ladders in $\mathcal{TOY}$

In this example (see Fig. 1), $\mathcal{TOY}$ is used to implement the instance *CFLP*$(\mathcal{R})$ of the *CFLP*$(\mathcal{D})$ scheme, with the parameter $\mathcal{D}$ replaced by the real numbers domain $\mathcal{R}$, which provides real numbers, arithmetic operations and various arithmetic constraints, including equalities, disequalities and inequalities. The type figure is intended to represent geometric figures as boolean functions, the function rect is intended to represent rectangles (more precisely, (rect (X,Y) LX LY) is intended to represent a rectangle with leftmost-bottom vertex (X,Y) and rightmost-upper vertex (X+LX,Y+LY)); and the function ladder is intended to build an infinite list of rectangles in the shape of a ladder. Although the text of the program seems to include no constraints, it uses arithmetic and comparison operators that give rise to constraint solving in execution time. More precisely, consider the following session in $\mathcal{TOY}$:

```
Toy> /run(examples/debug/ladder)                                      % compile ladder.toy
Toy> /cflpr                                                           % load CFLP(R)
Toy(R)> intersect (head (ladder (20,20) 50 20))
                         (head (ladder (5,5) 30 40)) (X,Y) == R       % goal
     { R -> true } { Y <= 5, X >= 2.0E+01, X <= 35 }                  % computed answer
```

The goal asks for the membership of a generic point (X,Y) to the intersection of the two rectangles (rect (20,20) 50 20) and (rect (5,5) 30 40), computed indirectly as the first steps of two particular ladders. The diagram included in Fig. 1 shows these two rectangles as well as the rectangle corresponding to their intersection (highlighted in black). The $\mathcal{TOY}$ system has solved the goal by a combination of lazy narrowing and constraint solving; the computed answer consists of the substitution R −> true and three constraints imposed on the variables X and Y[1]. The only constraint imposed on Y (namely Y <= 5) allows for arbitrarily small values of Y, which cannot correspond to points belonging to the rectangle expected as intersection. Therefore, the user will view the computed answer as *wrong* with respect to the intended meaning of the program. As we will see in Sections 4 and 5, the declarative debugging technique presented in this chapter leads to the diagnosis of the program rule for the function rect as responsible for the *wrong answer*. Indeed, this program rule is incorrect with respect to the intended program semantics; as shown in Fig. 1, the third inequality at the right hand side should be Y' >= Y instead of Y' <= Y.

After this correction, no more wrong computed answers will be observed for the goal discussed above. As any debugging technique, declarative diagnosis has limitations. A "corrected" program fragment can still include more subtle bugs that can be observed in the computed answers for other goals. In our case, we can consider the goal

```
Toy> /cflpr
Toy(R)> intersect (head (ladder (70,40) -50 -20))
                         (head (ladder (35,45) -30 -40)) (X,Y) == R
```

whose meaning with respect to the intended semantics is the same as for the previous goal, except that the rectangles playing the role of initial steps of the two ladders are represented differently. Since the boolean expression at the right hand side of the "corrected" program rule for function rect yields the result false whenever LX or LY is bound to a negative number, wrong answers including the substitution R −> false will be computed. Moreover, other answers including the substitution R −> true will be expected by the user but *missing* to occur among the computed answers.

The traditional approach to declarative debugging in the $CLP(\mathcal{D})$ scheme includes the diagnosis of both *wrong* and *missing* computed answers (Tessier & Ferrand, 2000). Now, we motivate our approach for the declarative debugging of *missing answers* in the $CFLP(\mathcal{D})$ scheme by means of the following example, intended to illustrate the main features of our diagnosis method.

[1] There are other five computed answers consisting of the substitution R −> false and various constraints imposed on X and Y.

Example 2. (Debugging Missing Answers in $\mathcal{TOY}$)

The following small *CFLP*$(\mathcal{H})$-program $\mathcal{P}_{\mathtt{fD}}$, written in $\mathcal{TOY}$ syntax over the *Herbrand domain* $\mathcal{H}$ with equality (==) and disequality (/=) constraints, 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 = instead 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 of such elements, and therefore `fDiff` is non-deterministic. Function `gen` is deterministic and returns a potentially infinite list of the form $[d_1, d_2, d_2, d_1, d_1, d_2, \ldots]$, 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 $G_{\mathtt{fD}}$: `even (fDiff (gen 1 2)) == true` for $\mathcal{P}_{\mathtt{fD}}$ 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 $\mathcal{P}_{\mathtt{fD}}$, the expression `(fDiff (gen 1 2))` would return only the numeric value `1`, and therefore the goal $G_{\mathtt{fD}}$ 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. As we will see in Sections 6 and 7, this particular *incompleteness symptom* could be mended by placing again the third rule for `fDiff` within the program.

3. The *CFLP*$(\mathcal{D})$ programming scheme

In this section we summarize the essentials of the *CFLP*$(\mathcal{D})$ scheme (López et al., 2006) for lazy Constraint Functional-Logic Programming over a parametrically given constraint domain $\mathcal{D}$, which serves as a logical and semantic framework for the declarative diagnosis method presented in the chapter.

3.1 Preliminary notions

We consider a *universal signature* $\Sigma = \langle DC, FS \rangle$, where $DC = \bigcup_{n \in \mathbb{N}} DC^n$ and $FS = \bigcup_{n \in \mathbb{N}} FS^n$ are countably infinite and mutually disjoint sets of *data constructors* resp. *evaluable function symbols*, indexed by arities. Evaluable functions are further classified into domain dependent *primitive functions* $PF^n \subseteq FS^n$ and user *defined functions* $DF^n = FS^n \setminus PF^n$ for each $n \in \mathbb{N}$. We write $\Sigma_\perp$ for the result of extending DC^0 with the special symbol $\perp$, intended to denote an undefined data value and we assume that DC includes the two constants *true* and *false* and the usual list constructors. We use the notations $c, d \in DC$, $f, g \in FS$, and $h \in DC \cup FS$. We also assume a countably infinite set $\mathcal{V}$ of *variables* $X, Y, \ldots$ and a set $\mathcal{U}$ of *primitive elements* $u, v, \ldots$ (as e.g. the set $\mathbb{R}$ of the real numbers) mutually disjoint and disjoint from $\Sigma_\perp$. *Expressions* $e \in Exp_\perp(\mathcal{U})$ have the following syntax:

$$e ::= \perp \mid u \mid X \mid h \mid (e\, e_1 \ldots e_m) \text{ \% shortly: } (e\, \overline{e}_m)$$

where $u \in \mathcal{U}$, $X \in \mathcal{V}$, $h \in DC \cup FS$. An important subclass of expressions is the set of *patterns* $s, t \in Pat_\perp(\mathcal{U})$, whose syntax is defined as follows:

$$t ::= \perp \mid u \mid X \mid (c\, \overline{t}_m) \mid (f\, \overline{t}_m)$$

where $u \in \mathcal{U}$, $X \in \mathcal{V}$, $c \in DC^n$ with $m \leq n$, and $f \in FS^n$ with $m < n$. Patterns are used as representations of possibly functional data values. For instance, the rectangle $(rect\ (5,5)\ 30\ 40)$ we met when discussing Example 1 is a functional data value represented as pattern[2].

As usual, we define *substitutions* $\sigma \in Sub_\perp(\mathcal{U})$ as mappings $\sigma : \mathcal{V} \to Pat_\perp(\mathcal{U})$ extended to $\sigma : Exp_\perp(\mathcal{U}) \to Exp_\perp(\mathcal{U})$ in the natural way. By convention, we write $e\sigma$ instead of $\sigma(e)$ for any $e \in Exp_\perp(\mathcal{U})$, and $\sigma\theta$ for the composition of σ and θ. A substitution σ such that $\sigma\sigma = \sigma$ is called *idempotent*.

3.2 Constraints over a constraint domain

Intuitively, a constraint domain provides a set of specific data elements, along with certain primitive functions operating upon them. Primitive predicates can be modelled as primitive functions returning boolean values. Formally, a *constraint domain* with primitive elements $\mathcal{U}$ and primitive functions $PF \subseteq FS$ is any structure $\mathcal{D} = \langle D_\mathcal{U}, \{p^\mathcal{D} \mid p \in PF\} \rangle$ with carrier set $D_\mathcal{U}$ the set of *ground* patterns (i.e., without variables) over $\mathcal{U}$ and interpretations $p^\mathcal{D} \subseteq D^n_\mathcal{U} \times D_\mathcal{U}$ of each $p \in PF^n$ satisfying the technical *monotonicity*, *antimonotonicity*, and *radicality* requirements given in (López et al., 2006). We use the notation $p^\mathcal{D}\, \overline{t}_n \to t$ to indicate that $(\overline{t}_n, t) \in p^\mathcal{D}$.

Constraints over a given constraint domain $\mathcal{D}$ are logical statements built from atomic constraints by means of logical conjunction $\wedge$ and existential quantification $\exists$. *Atomic constraints* can have the form $\Diamond$ (standing for *truth*), $\blacklozenge$ (standing for *falsity*), or $p\, \overline{e}_n \to! t$, meaning that the primitive function $p \in PF^n$ with parameters $\overline{e}_n \in Exp_\perp(\mathcal{U})$ returns a *total* result $t \in Pat_\perp(\mathcal{U})$ (i.e, with no occurrences of $\perp$). Constraints whose atomic parts have the

[2] Note that $(5,5)$ can be seen as syntactic sugar for $(pair\ 5\ 5)$, *pair* being a constructor for ordered pairs.

form $\Diamond$, $\blacklozenge$ or $p\,\bar{t}_n \rightarrow! \, t$ with $\bar{t}_n \in Pat_\perp(\mathcal{U})$ are called *primitive constraints*. In the sequel, we use the notation $PCon_\perp(\mathcal{D})$ for the set of primitive constraints over $\mathcal{D}$ and $DCon_\perp(\mathcal{D})$ for the set of user defined constraints over $\mathcal{D}$.

Example 3. (Constraint Domain $\mathcal{R}$) *The constraint domain $\mathcal{R}$ has the carrier set $D_\mathbb{R}$ of ground patters over $\mathbb{R}$ and the primitives defined below:*

1. *$eq_\mathbb{R}$, equality primitive for real numbers, such that: $eq_\mathbb{R}^\mathcal{R}\, u\, u \rightarrow true$ for all $u \in \mathbb{R}$; $eq_\mathbb{R}^\mathcal{R}\, u\, v \rightarrow false$ for all $u, v \in \mathbb{R}$, $u \neq v$; $eq_\mathbb{R}^\mathcal{R}\, t\, s \rightarrow \perp$ otherwise.*
2. *seq, strict equality primitive for ground patterns over the real numbers, such that: $seq^\mathcal{R}\, t\, t \rightarrow true$ for all total $t \in D_\mathbb{R}$; $seq^\mathcal{R}\, t\, s \rightarrow false$ for all $t, s \in D_\mathbb{R}$ such that t, s have no common upper bound with respect to the* information ordering *introduced in (López et al., 2006); $seq^\mathcal{R}\, t\, s \rightarrow \perp$ otherwise. In the sequel, $e_1 == e_2$ abbreviates $seq\, e_1\, e_2 \rightarrow! \, true$.*
3. *$+, -, *$, for addition, subtraction and multiplication, such that: $x +^\mathcal{R} y \rightarrow x +^\mathbb{R} y$ for all $x, y \in \mathbb{R}$; $t +^\mathcal{R} s \rightarrow \perp$ whenever $t \notin \mathbb{R}$ or $s \notin \mathbb{R}$; and analogously for $-^\mathcal{R}$ and $*^\mathcal{R}$.*
4. *$<, \leq, >, \geq$, for numeric comparisons, such that: $x <^\mathcal{R} y \rightarrow true$ for all $x, y \in \mathbb{R}$ with $x <^\mathbb{R} y$; $x <^\mathcal{R} y \rightarrow false$ for all $x, y \in \mathbb{R}$ with $x \geq^\mathbb{R} y$; $t <^\mathcal{R} s \rightarrow \perp$ whenever $t \notin \mathbb{R}$ or $s \notin \mathbb{R}$; and analogously for $\leq^\mathcal{R}, >^\mathcal{R}, \geq^\mathcal{R}$. In the sequel, $e_1 < e_2$ abbreviates $e_1 < e_2 \rightarrow! \, true$ and $e_1 \geq e_2$ abbreviates $e_1 < e_2 \rightarrow! \, false$ (analogously for other comparison primitives).*

The set of *valuations* over a constraint domain $\mathcal{D}$ is defined as the set $Val_\perp(\mathcal{D})$ of ground substitutions (i.e., mappings from variables to ground patterns). The semantics of constraints relies on the idea that a given valuation can satisfy or not a given constraint. Therefore, the set of *solutions* of $\pi \in PCon_\perp(\mathcal{D})$ can be defined in a natural way as a subset $Sol_\mathcal{D}(\pi) \subseteq Val_\perp(\mathcal{D})$; see (López et al., 2006) for details. Moreover, the set of solutions of $\Pi \subseteq PCon_\perp(\mathcal{D})$ is defined as $Sol_\mathcal{D}(\Pi) = \bigcap_{\pi \in \Pi} Sol_\mathcal{D}(\pi)$.

3.3 Constraint functional-logic programming

For any given constraint domain $\mathcal{D}$, a *CFLP*$(\mathcal{D})$-program $\mathcal{P}$ is presented as a set of constrained rewrite rules, called *program rules*, that define the behavior of user-defined functions. More precisely, a *constrained program rule R* for $f \in DF^n$ has the form $R : \; f\,\bar{t}_n \rightarrow r \Leftarrow \Delta$ (abbreviated as $f\,\bar{t}_n \rightarrow r$ if Δ is empty) and is required to satisfy the conditions listed below[3]:

1. The *left-hand side* $f\,\bar{t}_n$ is a *linear* expression (i.e, there is no variable having more than one occurrence), and for all $1 \leq i \leq n$, $t_i \in Pat_\perp(\mathcal{U})$ are total patterns. The *right-hand side* $r \in Exp_\perp(\mathcal{U})$ is also total.
2. $\Delta \subseteq DCon_\perp(\mathcal{D})$ is a finite set of total atomic constraints, intended to be interpreted as conjunction, and possibly including occurrences of user defined functions.

Program defined functions can be higher-order and/or non-deterministic. For instance, the $\mathcal{TOY}$ program presented in Example 1 can be viewed as an example of *CFLP*$(\mathcal{R})$-program written in $\mathcal{TOY}$'s syntax. The reader is referred to (López et al., 2006) for more explanations and examples in other constraint domains.

[3] In practice, $\mathcal{TOY}$ and similar languages require program rules to be well-typed in a polymorphic type system. However, the *CFLP*$(\mathcal{D})$ scheme can deal also with untyped programs. Well-typedness is viewed as an additional requirement, not as part of progam semantics.

The intended use of programs is to perform computations by solving goals proposed by the user. An *admissible goal* for a given $CFLP(\mathcal{D})$-program must have the form $G : \exists \overline{U}.\ (P \ \Box\ \Delta)$, where $\overline{U}$ is a finite set of so-called *existential variables* of the goal G (the rest of variables in G are called *free variables* and denoted by $fvar(G)$), P is a finite conjunction of so-called *productions* of the form $e \to s$ fulfilling the *admissibility conditions* given in (López et al., 2006), with $e \in Exp_\perp(\mathcal{U})$ and $s \in Pat_\perp(\mathcal{U})$ intended to mean that e can be evaluated to s, and $\Delta \subseteq DCon_\perp(\mathcal{D})$ is a finite conjunction of total user defined constraints. Two special kinds of admissible goals are interesting. *Initial goals*, where $\overline{U}$ and P are both empty (i.e., G has only a constrained part Δ without occurrences of existential variables), and *solved goals* (also called *solved forms*) of the form $S : \exists \overline{U}.\ (\sigma \ \Box\ \Pi)$, where σ is a finite set of productions $X \to t$ or $s \to Y$ interpreted as the variable bindings of an idempotent substitution and $\Pi \subseteq PCon_\perp(\mathcal{D})$ is a finite conjunction of total primitive constraints. Finally, a *goal solving system* for $CFLP(\mathcal{D})$ is expected to accept a program $\mathcal{P}$ and an initial goal G from the user, and to obtain one or more solved forms S_i as *computed answers*. As explained in Section 2, an initial goal G for the $CFLP(\mathcal{R})$-program shown in Example 1 can be $intersect\ (head\ (ladder\ (20,20)\ 50\ 20))\ (head\ (ladder\ (5,5)\ 30\ 40))\ (X,Y)$ $== R$ and a computed answer S for G is $R \to true\ \Box\ X \leq 35 \wedge X \geq 20 \wedge Y \leq 5$.

Goal solving systems can be implementations of $CFLP$ languages such as *Curry* (Hanus, 2003) or $\mathcal{TOY}$ (López & Sánchez, 1999), or formal *goal solving calculi* including recent proposals such as the $CDNC(\mathcal{D})$ calculus (López et al., 2004), which is sound and complete with respect to the declarative semantics discussed in the next subsection, and behaves as a faithful formal model for actual computations in the $\mathcal{TOY}$ system.

3.4 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 $f\ \bar{t}_n \to r \Leftarrow \Delta$ is replaced by a *standardized* form $f\ \overline{X}_n \to Y \Leftarrow R$, where $\overline{X}_n, Y$ are new variables, $R = \exists \overline{U}.\ R$ 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 $\forall \overline{X}_n, Y.\ (f\ \overline{X}_n \to Y \Rightarrow (\bigvee_{i \in I} R_i) \vee (\perp \to Y))$ for each function symbol f whose standardized program rules are $\{f\ \overline{X}_n \to Y \Leftarrow R_i\}_{i \in I}$. By convention, we may use the notation D_f for the disjunction $(\bigvee_{i \in I} R_i) \vee (\perp \to Y)$, 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 $(\perp \to Y)$ within D_f says that Y is bound to the undefined result $\perp$ 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*:

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

3.5 Answer collection assertions

In this work we propose to use computation trees for missing answers whose nodes have attached so-called *answer collection assertions*, briefly *aca*s. The *aca* at the root node has the form $G \Rightarrow \bigvee_{i \in I} S_i$, where G is the initial goal and $\bigvee_{i \in I} S_i$ (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 $G_{\mathtt{fD}}$ for program $\mathcal{P}_{\mathtt{fD}}$ is `even (fDiff (gen 1 2)) == true` $\Rightarrow$ ⧫ stating that this goal has (unexpectedly) failed. The *aca*s at internal nodes in our computation trees have the form $f\bar{t}_n \to t \,\square\, S \Rightarrow \bigvee_{i \in I} S_i$, asserting that the disjunction of computed answers $\bigvee_{i \in I} S_i$ covers all the solutions for the intermediate goal $G' : f\bar{t}_n \to t \,\square\, S$. Note that G' asks for the solutions of the production $f\bar{t}_n \to t$ 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. Due to *lazy evaluation*, the parameters $\bar{t}_n$ 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 $\mathcal{TOY}$.

3.6 Declarative semantics

In this subsection we recall some notions and results on the declarative semantics of *CFLP*($\mathcal{D}$)-programs which were developed in (López et al., 2006) and are needed for the rest of this work. Given a constraint domain $\mathcal{D}$ we consider two different kinds of constrained statements (briefly, *c-statements*) involving partial patterns t, $t_i \in Pat_\perp(\mathcal{U})$, partial expressions e, $e_i \in Exp_\perp(\mathcal{U})$, and a finite set $\Pi \subseteq PCon_\perp(\mathcal{D})$ of primitive constraints:

1. *c-productions* $e \to t \Leftarrow \Pi$, with $e \in Exp_\perp(\mathcal{U})$ and $t \in Pat_\perp(\mathcal{U})$, intended to mean that e can be evaluated to t if Π holds (if Π is empty they boil down to unconstrained productions written as $e \to t$). As a particular kind of c-productions useful for debugging we distinguish *c-facts* $f\,\bar{t}_n \to t \Leftarrow \Pi$ with $f \in DF^n$. A c-production is called *trivial* iff $t = \perp$ or $Sol_{\mathcal{D}}(\Pi) = \emptyset$.
2. *c-atoms* $p\,\bar{e}_n \to!\, t \Leftarrow \Pi$, with $p \in PF^n$ and t total (if Π is empty they boil down to unconstrained atoms written as $p\,\bar{e}_n \to!\, t$). A c-atom is called *trivial* iff $Sol_{\mathcal{D}}(\Pi) = \emptyset$.

In the sequel, we use φ to denote any c-statement. A *c-interpretation* over $\mathcal{D}$ is defined as any set $\mathcal{I}$ of c-facts including all the trivial c-facts and closed under *$\mathcal{D}$-entailment*, a generalization of the entailment notion introduced in (Caballero & Rodríguez, 2004) to arbitrary constraint domains. We write $\mathcal{I} \Vdash_{\mathcal{D}} \varphi$ to indicate that the c-statement φ (not necessarily a c-fact) is semantically valid in the c-interpretation $\mathcal{I}$. This notation relies on a formal definition given in (López et al., 2006). Now we are in a position to define various semantics notions which rely on a given c-interpretation $\mathcal{I}$ over $\mathcal{D}$.

Definition 1. (Interpretation-Dependent Semantic Notions)

1. *The set of* **solutions** *of* $\delta \in DCon_\perp(\mathcal{D})$ *is a subset* $Sol_{\mathcal{I}}(\delta) \subseteq Val_\perp(\mathcal{D})$ *defined as follows:*
   (a) $Sol_{\mathcal{I}}(\pi) = Sol_{\mathcal{D}}(\pi)$, *for any* $\pi \in PCon_\perp(\mathcal{D})$.

(b) $Sol_{\mathcal{I}}(\delta) = \{\eta \in Val_{\perp}(\mathcal{D}) \mid \mathcal{I} \Vdash_{\mathcal{D}} \delta\eta\}$, *for any* $\delta \in DCon_{\perp}(\mathcal{D}) \setminus PCon_{\perp}(\mathcal{D})$.

The set of solutions of a set of constraints $\Delta \subseteq DCon_{\perp}(\mathcal{D})$ *is defined as* $Sol_{\mathcal{I}}(\Delta) = \bigcap_{\delta\in\Delta} Sol_{\mathcal{I}}(\delta)$.

2. *The set of solutions of a production* $e \to t$ *is a subset* $Sol_{\mathcal{I}}(e \to t) \subseteq Val_{\perp}(\mathcal{D})$ *defined as* $Sol_{\mathcal{I}}(e \to t) = \{\eta \in Val_{\perp}(\mathcal{D}) \mid \mathcal{I} \Vdash_{\mathcal{D}} e\eta \to t\eta\}$. *The set of solutions of a set of productions P is defined as* $Sol_{\mathcal{I}}(P) = \bigcap_{(e\to t)\in P} Sol_{\mathcal{I}}(e \to t)$.
3. *The set of solutions of an admissible goal* $G : \exists\overline{U}.\ (P \ \square\ \Delta)$ *is a subset* $Sol_{\mathcal{I}}(G) \subseteq Val_{\perp}(\mathcal{D})$ *defined as follows:* $Sol_{\mathcal{I}}(G) = \{\eta \in Val_{\perp}(\mathcal{D}) \mid \eta' \in Sol_{\mathcal{I}}(P) \cap Sol_{\mathcal{I}}(\Delta)$ *for some* η' *such that* $\eta'(X) = \eta(X)$ *for all* $X \notin \overline{U}\}$.

For primitive constraints one can easily check that $Sol_{\mathcal{I}}(\Pi) = Sol_{\mathcal{D}}(\Pi)$. Moreover, we note that $Sol_{\mathcal{I}}(S) = Sol_{\mathcal{D}}(S)$ for every solved form S.

Definition 2. (Model-Theoretic Semantics) *Let* $\mathcal{P}$ *a CFLP*($\mathcal{D}$)*-program and* $\mathcal{I}$ *a c-interpretation.*

1. $\mathcal{I}$ *is a* **model** *of* $\mathcal{P}$ *(in symbols,* $\mathcal{I} \models_{\mathcal{D}} \mathcal{P}$*) iff every constrained program rule* $(f\bar{t}_n \to r \Leftarrow \Delta) \in \mathcal{P}$ *is* **valid** *in* $\mathcal{I}$*: for any ground substitution* $\eta \in Sub_{\perp}(\mathcal{U})$ *and* $t \in Pat_{\perp}(\mathcal{U})$ *ground such that* $(f\bar{t}_n \to r \Leftarrow \Delta)\eta$ *is ground,* $\mathcal{I} \Vdash_{\mathcal{D}} \Delta\eta$ *and* $\mathcal{I} \Vdash_{\mathcal{D}} r\eta \to t$ *one has* $\mathcal{I} \Vdash_{\mathcal{D}} (f\bar{t}_n)\eta \to t$ *(or equivalently,* $((f\bar{t}_n)\eta \to t) \in \mathcal{I}$*).*
2. *A solved form S is a* **semantically valid** *answer for a goal G with respect to a program* $\mathcal{P}$ *(in symbols,* $\mathcal{P} \models_{\mathcal{D}} G \Leftarrow S$*) iff* $Sol_{\mathcal{D}}(S) \subseteq Sol_{\mathcal{I}}(G)$ *for all* $\mathcal{I} \models_{\mathcal{D}} \mathcal{P}$.
3. $\mathcal{I}$ *is a* **model** *of* $\mathcal{P}^-$ *iff every axiom* $(f)^-_{\mathcal{P}} : (f\ \overline{X}_n \to Y \Rightarrow D_f) \in \mathcal{P}^-$ *satisfies* $Sol_{\mathcal{I}}(f\ \overline{X}_n \to Y) \subseteq Sol_{\mathcal{I}}(D_f)$. *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** *with respect to* $\mathcal{I}$.
4. *The aca* $G \Rightarrow \bigvee_{i\in I} S_i$ *is a* **logical consequence** *of* $\mathcal{P}^-$ *iff* $Sol_{\mathcal{I}}(G) \subseteq \bigcup_{i\in I} Sol_{\mathcal{D}}(S_i)$ *for any model* $\mathcal{I}$ *of* $\mathcal{P}^-$. *When this happens, we also say that the disjunction of answers* $\bigvee_{i\in I} S_i$ *is* **complete** *for G with respect to* $\mathcal{P}$.

4. Declarative debugging of wrong answers in *CFLP*($\mathcal{D}$)

In this section, we present the logical and semantic framework of the declarative diagnosis method of wrong answers for *CFLP*($\mathcal{D}$) and prove its logical correctness. In what follows, we assume that a constraint domain $\mathcal{D}$ and a *CFLP*($\mathcal{D}$)-program $\mathcal{P}$ are given.

4.1 Wrong answers and intended interpretations

Declarative diagnosis techniques rely on a declarative description of the intended program semantics. We will assume that the user knows (at least to the extent needed for answering queries during the debugging session) a so-called *intended model* $\mathcal{I}$, which is a c-interpretation expected to satisfy $\mathcal{I} \models_{\mathcal{D}} \mathcal{P}$, unless $\mathcal{P}$ is incorrect. For instance, $rect\,(X,Y)\,LX\,LY\,(A,B) \to false \Leftarrow A < X \wedge LX > 0 \wedge LY > 0$ could belong to the intended model $\mathcal{I}$ for the program fragment shown in Example 1. As explained in Subsection 3.6, the c-facts belonging to c-interpretations can be non-ground. Nevertheless, the model notion $\mathcal{I} \models_{\mathcal{D}} \mathcal{P}$ used here (see Definition 2 above) corresponds to the so-called *weak semantics* from (López et al., 2006), which depends just on the ground c-facts valid in $\mathcal{I}$. Therefore, different presentations of the intended model will be equivalent for the purposes of this work, as long as the ground c-facts valid in them are the same.

The aim of declarative diagnosis of wrong answers is to start with an observed *symptom* of erroneous program behavior, and detect some *error* in the program. The proper notions of symptom and error in our setting are as follows:

Definition 3. (Symptoms and Errors) *Assume $\mathcal{I}$ is the intended interpretation for program $\mathcal{P}$, and consider a solved form S produced as computed answer for the initial goal G by some goal solving system. We define:*

1. *S is a* **wrong answer** *w.r.t $\mathcal{I}$ (serving as* **symptom***) iff $Sol_{\mathcal{D}}(S) \not\subseteq Sol_{\mathcal{I}}(G)$.*
2. *$\mathcal{P}$ is* **incorrect** *with respect to $\mathcal{I}$ iff there exists some program rule $(f\bar{t}_n \to r \Leftarrow \Delta) \in \mathcal{P}$ (manifesting an* **error***) that is not valid in $\mathcal{I}$ (in the sense of Definition 2).*

For instance, the computed answer shown in Example 1 is wrong with respect to the intended model of the program assumed in that example, for the reasons already discussed in Section 2. As illustrated by that example, computed answers typically include constraints on the variables occurring in the initial goal. However, goal solving systems for *CFLP*($\mathcal{D}$) programs also maintain internal information on constraints related to variables used in intermediate computation steps, but not occurring in the initial goal. Such information is relevant for declarative debugging purposes. Therefore, in the rest of this section we will assume that computed answers *S* include also constraints related to intermediate variables.

4.2 A logical calculus for witnessing computed answers

Assuming that *S* is a computed answer for an initial goal *G* using a program $\mathcal{P}$, the declarative diagnosis of wrong answers needs a suitable *Computation Tree* (shortly, *CT*) representing the computation. In our setting we will obtain the *CT* from a logical proof $\mathcal{P} \vdash_{CPPC(\mathcal{D})} G \Leftarrow S$ which derives the statement $G \Leftarrow S$ from the program $\mathcal{P}$ in the *Constraint Positive Proof Calculus* (shortly $CPPC(\mathcal{D})$) given by the inference rules in Fig. 2. We will say that the $CPPC(\mathcal{D})$-proof *witnesses* the computed answer.

Most of these inference rules have been borrowed from the proof theory of *CRWL*($\mathcal{D}$), a *Constraint ReWriting Logic* which characterizes the semantics of *CFLP*($\mathcal{D}$) programs (López et al., 2006). The main novelties in $CPPC(\mathcal{D})$ are the addition of rule **EX** (to deal with existential quantifiers in computed answers) and a reformulation of rule $\mathbf{DF}_{\mathcal{P}}$, which is presented as the consecutive application of two inference steps named $\mathbf{AR}_f$ and $\mathbf{FA}_f$, which cannot be applied separately. The purpose of this composite inference is to introduce the c-facts $f\,\bar{t}_n \to t \Leftarrow \Pi$ at the conclusion of inference $\mathbf{FA}_f$, called *boxed c-facts* in the sequel. As we will see, only boxed c-facts will appear at the nodes of *CTs* obtained from $CPPC(\mathcal{D})$-proofs. Therefore, all the queries asked to the user during a declarative debugging session will be about the validity of c-facts in the intended model of the program, which is itself represented as a set of c-facts. We also agree that the premises $G\sigma \Leftarrow \Pi$ in rule **EX** (resp. $\Delta \Leftarrow \Pi$ in rule $\mathbf{DF}_{\mathcal{P}}$) must be understood as a shorthand for several premises $\alpha \Leftarrow \Pi$, one for each atomic φ in $G\sigma$ (resp. Δ). Moreover, rule **PF** depends on the side condition $Sol_{\mathcal{D}}(\Pi) \subseteq Sol_{\mathcal{D}}(p\bar{t}_n \to t)$ which is true iff $p^{\mathcal{D}}\,\bar{t}_n\eta \to t\eta$ holds for all $\eta \in Sol_{\mathcal{D}}(\Pi)$. Some other inference rules in Fig. 2 have similar conditions.

EX Existential $\dfrac{G\sigma \Leftarrow \Pi}{G \Leftarrow \exists \overline{U}.\,(\sigma \,\Box\, \Pi)}$ if $fvar(G) \cap \overline{U} = \emptyset$.

TI Trivial Inference $\dfrac{}{\varphi}$ if φ is a trivial c-statement.

RR Restricted Reflexivity $\dfrac{}{t \to t \Leftarrow \Pi}$ if $t \in \mathcal{U} \cup \mathcal{V}$.

SP Simple Production $\dfrac{}{s \to t \Leftarrow \Pi}$

if $s \in Pat_\perp(\mathcal{U})$, $s \in \mathcal{V}$ or $t \in \mathcal{V}$, and $Sol_\mathcal{D}(\Pi) \subseteq Sol_\mathcal{D}(s \to t)$.

DC Decomposition $\dfrac{e_1 \to t_1 \Leftarrow \Pi \;\ldots\; e_m \to t_m \Leftarrow \Pi}{h\overline{e}_m \to h\overline{t}_m \Leftarrow \Pi}$

IR Inner Reduction $\dfrac{e_1 \to t_1 \Leftarrow \Pi \;\ldots\; e_m \to t_m \Leftarrow \Pi}{h\overline{e}_m \to X \Leftarrow \Pi}$

if $h\overline{e}_m \notin Pat_\perp(\mathcal{U})$, $X \in \mathcal{V}$, and $Sol_\mathcal{D}(\Pi) \subseteq Sol_\mathcal{D}(h\overline{t}_m \to X)$.

PF Primitive Function $\dfrac{e_1 \to t_1 \Leftarrow \Pi \;\ldots\; e_n \to t_n \Leftarrow \Pi}{p\,\overline{e}_n \to t \Leftarrow \Pi}$

if $p \in PF^n$, $t_i \in Pat_\perp(\mathcal{U})$ $(1 \le i \le n)$, $Sol_\mathcal{D}(\Pi) \subseteq Sol_\mathcal{D}(p\overline{t}_n \to t)$.

DF$_\mathcal{P}$ $\mathcal{P}$-Defined Function

$$\dfrac{e_1 \to t_1 \Leftarrow \Pi \;\ldots\; e_n \to t_n \Leftarrow \Pi \qquad \dfrac{\Delta \Leftarrow \Pi \quad r \to t \Leftarrow \Pi}{\boxed{f\overline{t}_n \to t \Leftarrow \Pi}}\,\textbf{(FA}_f\textbf{)}}{f\,\overline{e}_n \to t \Leftarrow \Pi}\,\textbf{(AR}_f\textbf{)}$$

$$\dfrac{e_1 \to t_1 \Leftarrow \Pi \;\ldots\; e_n \to t_n \Leftarrow \Pi \qquad \dfrac{\Delta \Leftarrow \Pi \quad r \to s \Leftarrow \Pi}{\boxed{f\overline{t}_n \to s \Leftarrow \Pi}}\,\textbf{(FA}_f\textbf{)} \qquad s\,\overline{a}_k \to t \Leftarrow \Pi}{f\,\overline{e}_n\overline{a}_k \to t \Leftarrow \Pi}\,\textbf{(AR}_f\textbf{)}$$

if $f \in DF^n$ $(k > 0)$, $(f\,\overline{t}_n \to r \Leftarrow \Delta) \in [\mathcal{P}]_\perp \equiv \{R\theta \mid R \in \mathcal{P}, \theta \in Sub_\perp(\mathcal{U})\}$, and $s \in Pat_\perp(\mathcal{U})$.

AC Atomic Constraint $\dfrac{e_1 \to t_1 \Leftarrow \Pi \;\ldots\; e_n \to t_n \Leftarrow \Pi}{p\,\overline{e}_n \to!\, t \Leftarrow \Pi}$

if $p \in PF^n$, $t_i \in Pat_\perp(\mathcal{U})$ $(1 \le i \le n)$, $Sol_\mathcal{D}(\Pi) \subseteq Sol_\mathcal{D}(p\overline{t}_n \to!\, t)$.

Fig. 2. The Constraint Positive Proof Calculus $CPPC(\mathcal{D})$

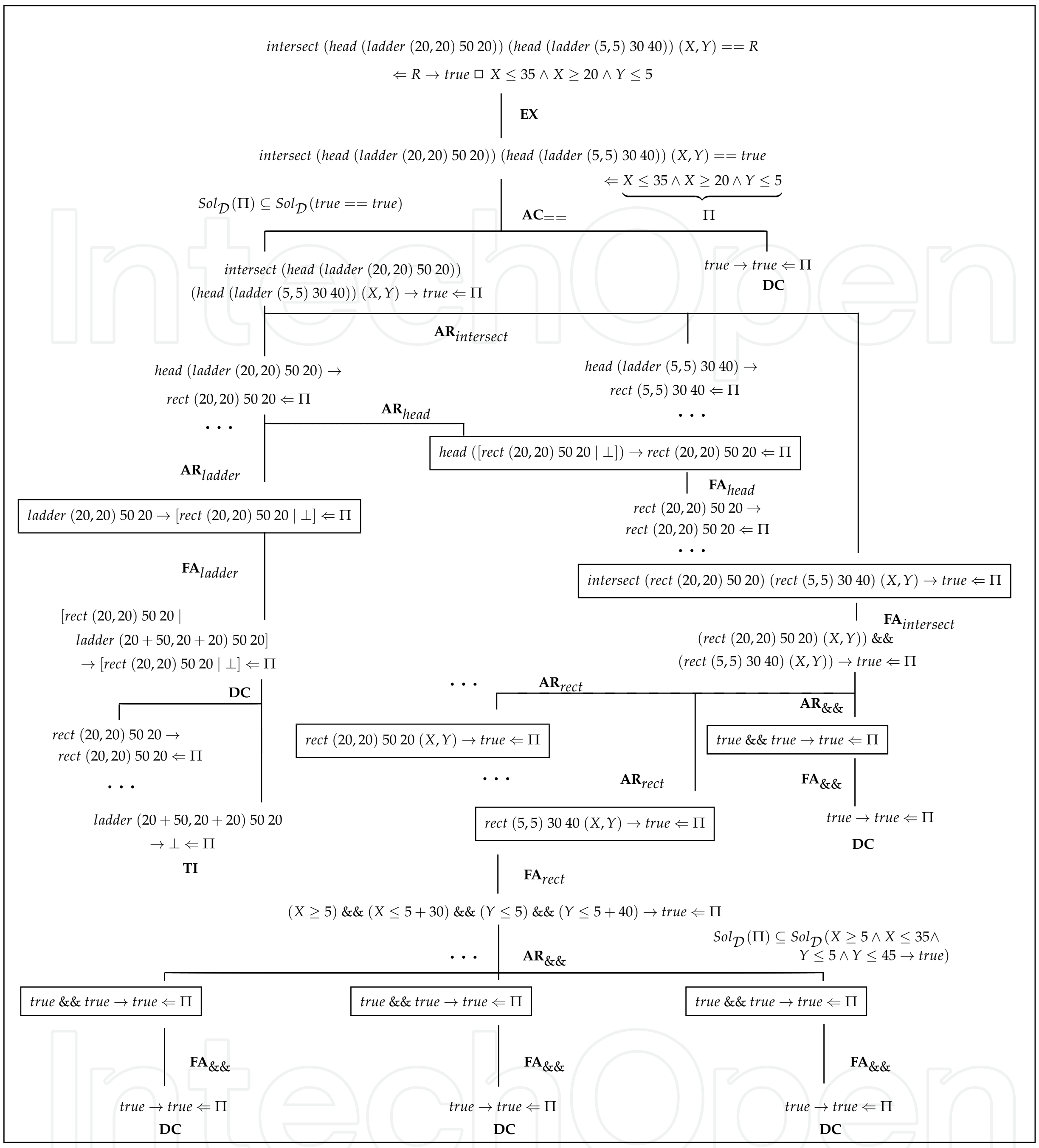

intersect (head (ladder (20,20) 50 20)) (head (ladder (5,5) 30 40)) (X,Y) == R ⇐ R → true □ X ≤ 35 ∧ X ≥ 20 ∧ Y ≤ 5
EX
intersect (head (ladder (20,20) 50 20)) (head (ladder (5,5) 30 40)) (X,Y) == true ⇐ X ≤ 35 ∧ X ≥ 20 ∧ Y ≤ 5
Π
Sol_D(Π) ⊆ Sol_D(true == true)
AC==
intersect (head (ladder (20,20) 50 20)) (head (ladder (5,5) 30 40)) (X,Y) → true ⇐ Π
true → true ⇐ Π
DC
AR_intersect
head (ladder (20,20) 50 20) → rect (20,20) 50 20 ⇐ Π
head (ladder (5,5) 30 40) → rect (5,5) 30 40 ⇐ Π
AR_head
head ([rect (20,20) 50 20 | ⊥]) → rect (20,20) 50 20 ⇐ Π
AR_ladder
FA_head
ladder (20,20) 50 20 → [rect (20,20) 50 20 | ⊥] ⇐ Π
rect (20,20) 50 20 → rect (20,20) 50 20 ⇐ Π
FA_ladder
intersect (rect (20,20) 50 20) (rect (5,5) 30 40) (X,Y) → true ⇐ Π
[rect (20,20) 50 20 | ladder (20 + 50, 20 + 20) 50 20] → [rect (20,20) 50 20 | ⊥] ⇐ Π
FA_intersect
(rect (20,20) 50 20) (X,Y)) && (rect (5,5) 30 40) (X,Y)) → true ⇐ Π
DC
AR_rect
AR_&&
rect (20,20) 50 20 → rect (20,20) 50 20 ⇐ Π
rect (20,20) 50 20 (X,Y) → true ⇐ Π
true && true → true ⇐ Π
AR_rect
FA_&&
ladder (20 + 50, 20 + 20) 50 20 → ⊥ ⇐ Π
TI
rect (5,5) 30 40 (X,Y) → true ⇐ Π
true → true ⇐ Π
DC
FA_rect
(X ≥ 5) && (X ≤ 5 + 30) && (Y ≤ 5) && (Y ≤ 5 + 40) → true ⇐ Π
Sol_D(Π) ⊆ Sol_D(X ≥ 5 ∧ X ≤ 35∧ Y ≤ 5 ∧ Y ≤ 45 → true)
AR_&&
true && true → true ⇐ Π
true && true → true ⇐ Π
true && true → true ⇐ Π
FA_&&
FA_&&
FA_&&
true → true ⇐ Π
DC
true → true ⇐ Π
DC
true → true ⇐ Π
DC

Fig. 3. A Positive Proof Tree in $CPPC(\mathcal{R})$

Any $CPPC(\mathcal{D})$-derivation $\mathcal{P} \vdash_{CPPC(\mathcal{D})} G \Leftarrow S$ can be depicted in the form of a *Positive Proof Tree* over $\mathcal{D}$ (shortly, $PPT(\mathcal{D})$) with $G \Leftarrow S$ at the root and c-statements at the internal nodes, and such that the statement at any node is inferred from the statements at its children using some $CPPC(\mathcal{D})$ inference rule. In particular, the statement at the root must be inferred using rule **EX**, which is then applied nowherelse in the proof tree. Fig. 3 shows a $PPT(\mathcal{R})$ representing a $CPPC(\mathcal{R})$-derivation which witnesses the computed answer from Example 1, which is wrong with respect to the intended model of the program. We say that a goal solving system is called $CPPC(\mathcal{D})$-*sound* iff for any computed answer S obtained for an initial goal G

using program $\mathcal{P}$ there is some witnessing $CPPC(\mathcal{D})$-proof $\mathcal{P} \vdash_{CPPC(\mathcal{D})} G \Leftarrow S$. The next result shows that $CPPC(\mathcal{D})$-sound goal solving systems exist:

Theorem 1. (Existence of $CPPC(\mathcal{D})$-Sound Goal Solving Systems) *The goal solving calculus $CDNC(\mathcal{D})$ given in (López et al., 2004) is $CPPC(\mathcal{D})$-sound.*

Proof. Straightforward adaptation of the soundness theorem for $CDNC(\mathcal{D})$ presented in (López et al., 2004). □

In addition to the goal solving calculus $CDNC(\mathcal{D})$, other formal goal solving calculi known for $CFLP(\mathcal{D})$ are also $CPPC(\mathcal{D})$-sound. Moreover, it is also reasonable to assume $CPPC(\mathcal{D})$-soundness for implemented goal solving systems such as *Curry* (Hanus, 2003) and $\mathcal{TOY}$ (López & Sánchez, 1999) whose computation model is based on constrained lazy narrowing. Moreover, any $CPPC(\mathcal{D})$-sound goal solving system is semantically sound in the sense of item 2 in Definition 2:

Theorem 2. (Semantic Correctness of the $CPPC(\mathcal{D})$ Calculus) *If G is an initial goal for $\mathcal{P}$ and S is a solved goal s.t. $\mathcal{P} \vdash_{CPPC(\mathcal{D})} G \Leftarrow S$ then $\mathcal{P} \models_{\mathcal{D}} G \Leftarrow S$.*

Proof. For each of the inference rules **EX**, $\mathbf{AR}_f$, and $\mathbf{FA}_f$, we prove that an arbitrary model $\mathcal{I} \models_{\mathcal{D}} \mathcal{P}$ such that the premises of the rule are valid in $\mathcal{I}$, also verifies that the conclusion of the rule is valid in $\mathcal{I}$. Similar proofs for the other inference rules in $CFLP(\mathcal{D})$ can be found in (López et al., 2006).

- The rule **EX** is semantically correct. Let $\mathcal{I}$ be an arbitrary model of $\mathcal{P}$ such that $\mathcal{I} \models_{\mathcal{D}} G\sigma \Leftarrow \Pi$, i.e., $Sol_{\mathcal{D}}(\Pi) \subseteq Sol_{\mathcal{I}}(G\sigma)$. We prove that $\mathcal{I} \models_{\mathcal{D}} G \Leftarrow \exists\overline{U}.\ (\sigma \,\square\, \Pi)$, i.e., $Sol_{\mathcal{D}}(\exists\overline{U}.\ (\sigma \,\square\, \Pi)) \subseteq Sol_{\mathcal{I}}(G)$. Let $\eta \in Sol_{\mathcal{D}}(\exists\overline{U}.\ (\sigma \,\square\, \Pi))$. By the syntactic form of solved goals, $\eta \in Sol_{\mathcal{D}}(\exists\overline{U}.\ (\overline{X_n \to t_n} \wedge \overline{s_m \to Y_m} \ \square\ \Pi))$ and $\eta \in Sol_{\mathcal{D}}(\exists\ \overline{U}.\ (\overline{X_n = t_n} \wedge \overline{Y_m = s_m} \,\square\, \Pi))$. By applying Definition 1, there exists $\eta' \in Val_{\perp}(\mathcal{D})$ such that $\eta' =_{\setminus\overline{U}} \eta$ y $\eta' \in Sol_{\mathcal{D}}(\overline{X_n = t_n} \wedge \overline{Y_m = s_m} \,\square\, \Pi)$, and therefore, $\eta' \in Sol_{\mathcal{D}}(\overline{X_n = t_n} \wedge \overline{Y_m = s_m})$ (i.e., $\eta' \in Sol_{\mathcal{D}}(\sigma)$) and $\eta' \in Sol_{\mathcal{D}}(\Pi)$. Since by *induction hypothesis* $Sol_{\mathcal{D}}(\Pi) \subseteq Sol_{\mathcal{I}}(G\sigma)$, it follows that $\eta' \in Sol_{\mathcal{I}}(G\sigma)$. Moreover, since $\eta' \in Sol_{\mathcal{D}}(\sigma)$, we obtain $\eta' \in Sol_{\mathcal{I}}(G)$. In consequence, there exists $\eta' \in Val_{\perp}(\mathcal{D})$ such that $\eta' =_{\setminus\overline{U}} \eta$ and $\eta' \in Sol_{\mathcal{I}}(G)$. Finally, using the condition of applicability $fvar(G) \cap \overline{U} = \emptyset$ associated to the rule **EX**, we can conclude that $\eta \in Sol_{\mathcal{I}}(G)$.
- The rule $\mathbf{AR}_f$ is semantically correct. Let $\mathcal{I}$ be an arbitrary model of $\mathcal{P}$ such that $\mathcal{I} \models_{\mathcal{D}} e_i \to t_i \Leftarrow \Pi$ for each $1 \leq i \leq n$ (i.e., $Sol_{\mathcal{D}}(\Pi) \subseteq Sol_{\mathcal{I}}(e_i \to t_i)$ for each $1 \leq i \leq n$), $\mathcal{I} \models_{\mathcal{D}} f\bar{t}_n \to s \Leftarrow \Pi$ (i.e., $Sol_{\mathcal{D}}(\Pi) \subseteq Sol_{\mathcal{D}}(f\bar{t}_n \to s)$) and $\mathcal{I} \models_{\mathcal{D}} s\bar{a}_k \to s \Leftarrow \Pi$ (i.e., $Sol_{\mathcal{D}}(\Pi) \subseteq Sol_{\mathcal{I}}(s\bar{a}_k \to t)$). We prove that $\mathcal{I} \models_{\mathcal{D}} f\bar{e}_n\bar{a}_k \to t \Leftarrow \Pi$, i.e., $Sol_{\mathcal{D}}(\Pi) \subseteq Sol_{\mathcal{I}}(f\bar{e}_n\bar{a}_k \to t)$. Let $\eta \in Sol_{\mathcal{D}}(\Pi)$. We have then $\eta \in Sol_{\mathcal{I}}(e_i \to t_i)$ for each $1 \leq i \leq n$, and by Definition 1, $\mathcal{I} \Vdash_{\mathcal{D}} e_i\eta \to t_i\eta$ for each $1 \leq i \leq n$. Analogously, $\eta \in Sol_{\mathcal{I}}(f\bar{t}_n \to s)$, by Definition 1, $\mathcal{I} \Vdash_{\mathcal{D}} f\bar{t}_n\eta \to s\eta$, and by the *Conservation Property* (see (López et al., 2006) for details), $(f\bar{t}_n\eta \to s\eta) \in \mathcal{I}$. Analogously, $\eta \in Sol_{\mathcal{I}}(s\bar{a}_k \to t)$ and by Definition 1, $\mathcal{I} \Vdash_{\mathcal{D}} (s\eta)(\bar{a}_k\eta) \to t\eta$. But then, by applying of the rule $\mathbf{DF}_{\mathcal{I}}$ (see (López et al., 2006) for details), we have that $\mathcal{I} \Vdash_{\mathcal{D}} f(\bar{e}_n\eta)(\bar{a}_k\eta) \to t\eta$. From Definition 1, we obtain finally $\eta \in Sol_{\mathcal{I}}(f\bar{e}_n\bar{a}_k \to t)$.

- The rule $\mathbf{FA}_f$ is semantically correct. By definition of $[\mathcal{P}]_\perp$, there are $(f\overline{t'}_n \to r' \Leftarrow \Delta') \in \mathcal{P}$ and $\theta \in Sub_\perp(\mathcal{U})$ such that $(f\overline{t'}_n \to r' \Leftarrow \Delta')\theta \equiv (f\bar{t}_n \to r \Leftarrow \Delta)$. Let $\mathcal{I}$ be an arbitrary model of $\mathcal{P}$ such that $\mathcal{I} \models_\mathcal{D} \Delta \Leftarrow \Pi$ (i.e., $Sol_\mathcal{D}(\Pi) \subseteq Sol_\mathcal{I}(\Delta)$) and $\mathcal{I} \models_\mathcal{D} r \to s \Leftarrow \Pi$ (i.e., $Sol_\mathcal{D}(\Pi) \subseteq Sol_\mathcal{I}(r \to s)$). We prove that $\mathcal{I} \models_\mathcal{D} f\bar{t}_n \to s \Leftarrow \Pi$, i.e., $Sol_\mathcal{D}(\Pi) \subseteq Sol_\mathcal{I}(f\bar{t}_n \to s)$. Let $\eta \in Sol_\mathcal{D}(\Pi)$. Then we have $\eta \in Sol_\mathcal{I}(\Delta)$, and by Definition 1, $\mathcal{I} \Vdash_\mathcal{D} \Delta\eta$, and also, $\mathcal{I} \Vdash_\mathcal{D} \Delta'\theta\eta$. Analogously, $\eta \in Sol_\mathcal{I}(r \to s)$, and by Definition 1, $\mathcal{I} \Vdash_\mathcal{D} r\eta \to s\eta$, and also, $\mathcal{I} \Vdash_\mathcal{D} r'\theta\eta \to s\eta$. We have then $(f\overline{t'}_n \to r' \Leftarrow \Delta') \in \mathcal{P}$, $\theta\eta \in Sub_\perp(\mathcal{U})$ ground substitution and $s\eta \in Pat_\perp(\mathcal{U})$ ground such that $(f\overline{t'}_n \to r' \Leftarrow \Delta')\theta\eta \equiv (f\bar{t}_n \to r \Leftarrow \Delta)\eta$ is ground, $\mathcal{I} \Vdash_\mathcal{D} \Delta'\theta\eta$ and $\mathcal{I} \Vdash_\mathcal{D} r'\theta\eta \to s\eta$. Since $\mathcal{I}$ is a model of $\mathcal{P}$, by applying Definition 2, we obtain $((f\overline{t'}_n)\theta\eta \to s\eta) \in \mathcal{I}$, i.e., $((f\bar{t}_n)\eta \to s\eta) \in \mathcal{I}$, or also, $(f\bar{t}_n \to s)\eta \in \mathcal{I}$. Finally, by applying the *Conservation Property* (see (López et al., 2006) for details), it is equivalent to $\mathcal{I} \Vdash_\mathcal{D} (f\bar{t}_n \to s)\eta$, and by Definition 1, we can conclude that $\eta \in Sol_\mathcal{I}(f\bar{t}_n \to s)$.

□

4.3 Declarative diagnosis using positive proof trees

Now we are ready to present a declarative diagnosis method of wrong answers and to prove its correctness. Our results apply to any $CPPC(\mathcal{D})$-sound goal solving system. First we prove that the observation of an error symptom implies the existence of some error in the program:

Theorem 3. (Wrong Answers Are Caused By Erroneous Program Rules) *We assume that a $CPPC(\mathcal{D})$-sound goal solving system computes S as an answer for the initial goal G using program $\mathcal{P}$. If S is wrong with respect to the user's intended interpretation $\mathcal{I}$ then some program rule belonging to $\mathcal{P}$ is incorrect with respect to $\mathcal{I}$.*

Proof. Because of $CPPC(\mathcal{D})$-soundness of the goal solving system, we know that $\mathcal{P} \vdash_{CPPC(\mathcal{D})} G \Leftarrow S$. Then, from Theorem 2 we obtain $\mathcal{P} \models_\mathcal{D} G \Leftarrow S$, i.e., $Sol_\mathcal{D}(S) \subseteq Sol_\mathcal{J}(G)$ for each model $\mathcal{J} \models_\mathcal{D} \mathcal{P}$. Since S is wrong with respect to the user's intended model $\mathcal{I}$, it must be the case that $Sol_\mathcal{D}(S) \nsubseteq Sol_\mathcal{I}(G)$ because of Definition 3. Therefore, we can conclude that the intended model $\mathcal{I}$ is not a model of $\mathcal{P}$. Then, by Definition 2, some program rule belonging to $\mathcal{P}$ is not valid in $\mathcal{I}$. □

The previous theorem does not yet provide a practical method for finding an erroneous program rule. As explained in the Introduction, a declarative diagnosis method is expected to find the erroneous program rule by inspecting a CT. We propose to use abbreviated $CPPC(\mathcal{D})$ proof trees as CTs. Since $\mathbf{DF}_\mathcal{P}$ is the only inference rule in the $CPPC(\mathcal{D})$ calculus that depends on the program, abbreviated proof trees will omit the inference steps related to all the other $CPPC(\mathcal{D})$ rules. More precisely, given a $PPT(\mathcal{D})$ $\mathcal{T}$, its associated *Abbreviated Positive Proof Tree* over $\mathcal{D}$ (shortly, $APPT(\mathcal{D})$) $\mathcal{AT}$ is defined as follows:

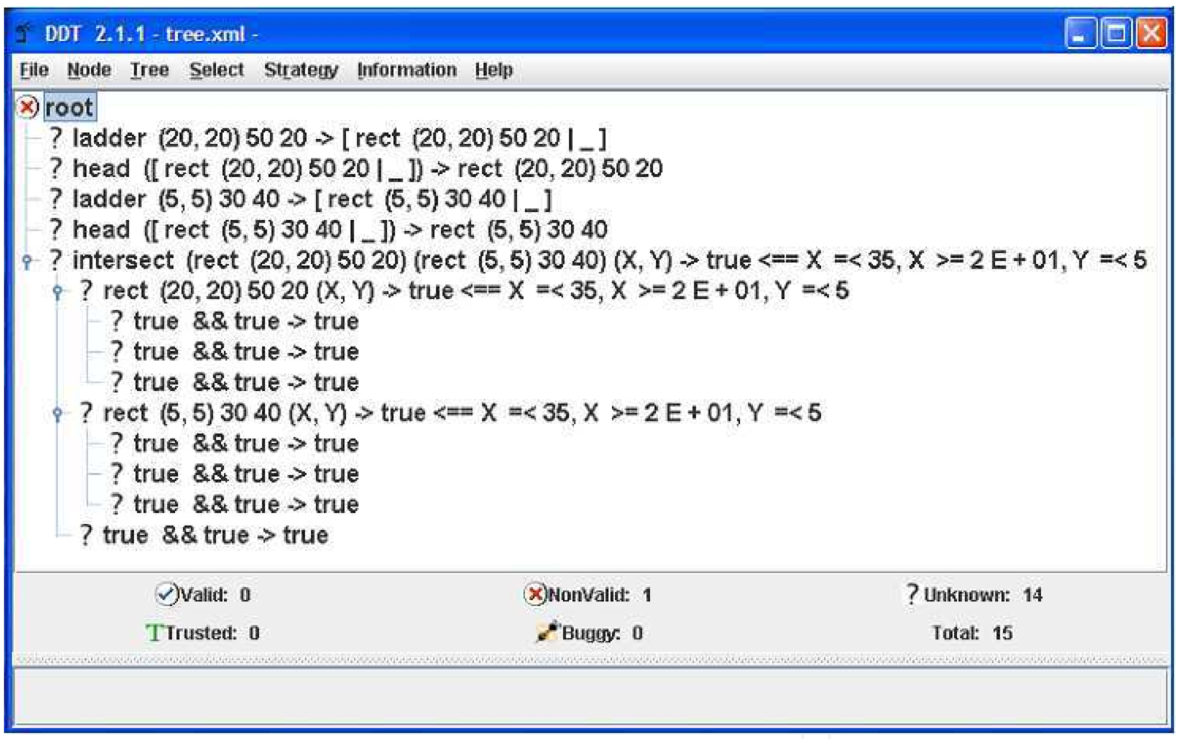
- The root of $\mathcal{AT}$ is the root of $\mathcal{T}$.
- The children of a node N in $\mathcal{AT}$ are the closest descendants of N in $\mathcal{T}$ corresponding to boxed c-facts introduced by $\mathbf{DF}_\mathcal{P}$ inference steps.

A node in an $APPT(\mathcal{D})$ is called a *buggy node* iff the c-statement at the node is not valid in the intended interpretation $\mathcal{I}$, while all the c-statements at the children nodes are valid in $\mathcal{I}$. Our last theorem guarantees that declarative diagnosis with $APPT(\mathcal{D})$s used as *CTs* leads to the correct detection of program errors.

Theorem 4 (Declarative Diagnosis of Wrong Answers). *Under the assumptions of Theorem 3, any $APPT(\mathcal{D})$ witnessing $\mathcal{P} \vdash_{CPPC(\mathcal{D})} G \Leftarrow S$ (which must exist due to $CPPC(\mathcal{D})$-soundness of the goal solving system) has some buggy node. Moreover, each buggy node points to a program rule belonging to $\mathcal{P}$ which is incorrect in the user's intended interpretation.*

5. A declarative debugging tool of wrong answers in $\mathcal{TOY}$

Fig. 4 shows the $APPT(\mathcal{R})$ associated to the $PPT(\mathcal{R})$ of Fig. 3 as displayed by $\mathcal{DDT}$, the debugger tool included in the system $\mathcal{TOY}$. Although in theory all the c-facts in a $PPT(\mathcal{R})$ should include the same constraint Π, in practice the tool simplifies Π at each c-fact $f\,\bar{t}_n \to t \Leftarrow \Pi$, keeping only those atomic constraints related to the variables occurring on $f\,\bar{t}_n \to t$. It can be checked that such a simplification does not affect the intended meaning of c-facts.



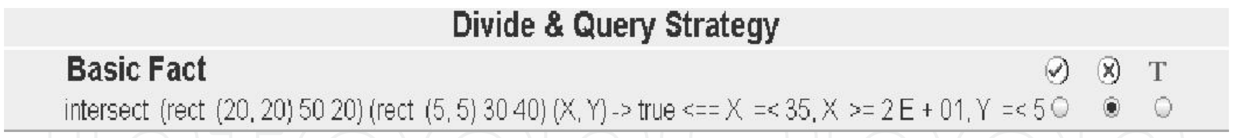
```
root
  ? ladder (20, 20) 50 20 -> [ rect (20, 20) 50 20 | _ ]
  ? head ([ rect (20, 20) 50 20 | _ ]) -> rect (20, 20) 50 20
  ? ladder (5, 5) 30 40 -> [ rect (5, 5) 30 40 | _ ]
  ? head ([ rect (5, 5) 30 40 | _ ]) -> rect (5, 5) 30 40
  ? intersect (rect (20, 20) 50 20) (rect (5, 5) 30 40) (X, Y) -> true <== X =< 35, X >= 2 E + 01, Y =< 5
    ? rect (20, 20) 50 20 (X, Y) -> true <== X =< 35, X >= 2 E + 01, Y =< 5
      ? true && true -> true
      ? true && true -> true
      ? true && true -> true
    ? rect (5, 5) 30 40 (X, Y) -> true <== X =< 35, X >= 2 E + 01, Y =< 5
      ? true && true -> true
      ? true && true -> true
      ? true && true -> true
    ? true && true -> true

Valid: 0        NonValid: 1        ? Unknown: 14
Trusted: 0      Buggy: 0           Total: 15
```

Fig. 4. The $APPT(\mathcal{R})$ corresponding to the $PPT(\mathcal{R})$ of Fig. 3

Before starting a *debugging session*, the user may inspect and simplify the tree using several facilities. For instance the user could mark any node corresponding to the infix function && as *trusted*, indicating that the definition of && is surely not erroneous. This makes all the nodes corresponding to && automatically valid. Valid nodes can be removed from the tree safely (the set of buggy nodes doesn't change) by using a suitable menu option.

Next, the user can start a debugging session by selecting one of the two possible strategies included in $\mathcal{DDT}$: the *top-down* or the *divide and query* strategy (see (Caballero & Rodríguez,

2004) for a comparative between both strategies in an older version of $\mathcal{DDT}$ which did not yet support constraints). After selecting the *divide and query* strategy, which usually leads to shorter sessions, $\mathcal{DDT}$ asks about the validity of the following node:

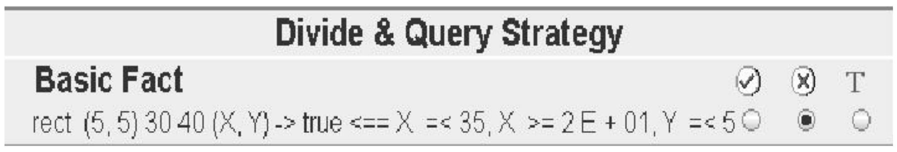


Divide & Query Strategy

Basic Fact ✓ ⊗ T

intersect (rect (20, 20) 50 20) (rect (5, 5) 30 40) (X, Y) -> true <== X =< 35, X >= 2 E + 01, Y =< 5 ○ ● ○

The intended program model corresponds to the intuitions explained in Section 2. Therefore, the question must be understood as: *Is* (X, Y) *a point in the intersection of the two rectangles for all possible values of* X, Y *satisfying* $X \leq 35, X \geq 20, Y \leq 5$ *is* (X, Y)? The answer is *no*, because with these constraints Y can take any value less than 5 and some of these values would yield a pair (X, Y) out of the intersection for every X. Therefore the user marks the cross meaning that the c-fact is non-valid. The next question is:

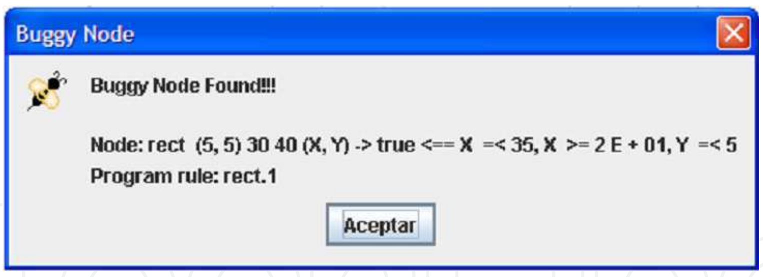


Divide & Query Strategy

Basic Fact ✓ ⊗ T

rect (5, 5) 30 40 (X, Y) -> true <== X =< 35, X >= 2 E + 01, Y =< 5 ○ ● ○

which is also reported as non-valid by the user. At this point a buggy node is found by the tool, pointing out to the incorrect program rule and ending the debugging session:



Buggy Node

Buggy Node Found!!!

Node: rect (5, 5) 30 40 (X, Y) -> true <== X =< 35, X >= 2 E + 01, Y =< 5
Program rule: rect.1

The current version of the debugger supports programs using the constraint domain $\mathcal{R}$, which provides arithmetic constraints over the real numbers as well as strict equality and disequality constraints over data values of any type; see Example 3 and (López et al., 2006) for details. The tool is as an extension of older versions which did not yet support constraints over the domain $\mathcal{R}$ (Caballero, 2005; Caballero & Rodríguez, 2004), and it is part of the public distribution of the functional-logic programming system $\mathcal{TOY}$, available at `http://toy.sourceforge.net`. The $APPT(\mathcal{R})$ associated to a wrong answer is constructed by means of a suitable program transformation. The yielded tree is then displayed through a graphical debugging interface implemented in Java. More detailed explanations on the practical use of $\mathcal{DDT}$ can be found in (Caballero, 2005; Caballero & Rodríguez, 2004).

SF Solved Form $$\frac{}{R \,\Box\, S \Rightarrow D}$$ if $Sol_{\mathcal{D}}(S) \subseteq Sol_{\mathcal{D}}(D)$.

CJ Conjunction

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

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

TS Trivial Statement $$\frac{}{\varphi : G \Rightarrow D}$$ if $Sol(G) \subseteq Sol_{\mathcal{D}}(D)$.

DC DeComposition $$\frac{\overline{e_m \to t_m} \,\Box\, S \Rightarrow D}{h\bar{e}_m \to h\bar{t}_m \,\Box\, S \Rightarrow D}$$ if $h\bar{e}_m$ is not a pattern.

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

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

(AR)$_p$ Argument Reduction for Primitive Functions

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

if $p \in PF^n, \overline{X}_n \notin var(p\bar{e}_n \to^{?} t \,\Box\, S)$, $\to^{?} \equiv \to$ (*production*)$\cup \to!$ (*constraint*).

(AR)$_f$ Argument Reduction for Defined Functions

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

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

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

if $f \in DF^n$, $\overline{X}_n, Y \notin var(f\bar{t}_n \to t \,\Box\, S)$, and $(f\overline{X}_n \to Y \Rightarrow \bigvee_{i\in I} R_i) \in \mathcal{P}^{-}$.

Fig. 5. The Constraint Negative Proof Calculus $CNPC(\mathcal{D})$

6. Declarative debugging of missing answers in $CFLP(\mathcal{D})$

The declarative debugging of *missing answers* also assumes an intended interpretation $\mathcal{I}_{\mathcal{P}}$ of the $CFLP(\mathcal{D})$-program $\mathcal{P}$, starts with the observation of an *incompleteness symptom* and ends with an *incompleteness diagnosis*. A more precise definition of this *debugging scenario of missing answers* is as follows:

Definition 4. (Debugging Scenario of Missing Answers) *For any given $CFLP(\mathcal{D})$-program $\mathcal{P}$:*

1. *An* **incompleteness symptom** *occurs if the goal solving system computes finitely many solved goals $\{S_i\}_{i\in I}$ as answers for an admissible initial goal G, and the programmer judges that $Sol_{\mathcal{I}_{\mathcal{P}}}(G) \nsubseteq \bigcup_{i\in I} Sol_{\mathcal{D}}(S_i)$, meaning that the aca $G \Rightarrow \bigvee_{i\in I} S_i$ is not valid in the intended interpretation $\mathcal{I}_{\mathcal{P}}$, so that some expected answers are* **missing**.
2. *An* **incompleteness diagnosis** *is given by pointing to some defined function symbol f such that the axiom $(f)^{-}_{\mathcal{P}} : (f\ \overline{X}_n \rightarrow Y \Rightarrow D_f)$ for f in $\mathcal{P}^-$ is not valid in $\mathcal{I}_{\mathcal{P}}$, which means $Sol_{\mathcal{I}_{\mathcal{P}}}(f\ \overline{X}_n \rightarrow Y) \nsubseteq Sol_{\mathcal{I}_{\mathcal{P}}}(D_f)$, 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. Assume now that an incompleteness symptom has been observed by the programmer. Since the goal solving system has computed the disjunction of answers $D = \bigvee_{i\in I} S_i$, the *aca* $G \Rightarrow D$ asserting that the computed answers cover all the solutions of G should be derivable from $\mathcal{P}^-$. The *$\underline{C}$onstraint $\underline{N}$egative $\underline{P}$roof $\underline{C}$alculus* $CNPC(\mathcal{D})$ consisting of the inference rules displayed in Fig. 5 has been designed with the aim of enabling logical proofs $\mathcal{P}^- \vdash_{CNPC(\mathcal{D})} G \Rightarrow D$ of *aca*s. We use a special operator & in order to express the result of attaching to a given goal G a solved goal S' resulting from a previous computation, so that computation can continue from the new goal G & S'.

Formally, assuming $G = \exists\overline{U}.\ (R \,\Box\, (\Pi \,\Box\, \sigma))$ and $S' = \exists\overline{U}'.\ (\Pi' \,\Box\, \sigma')$ 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 & S' is defined as $\exists\overline{U}'.\ (R\sigma' \,\Box\, (\Pi' \,\Box\, \sigma'))$. The inference rule **CJ** infers an *aca* for a goal with composed kernel $(R_1 \wedge R_2) \,\Box\, S$ from *aca*s for goals with kernels of the form $R_1 \,\Box\, S$ and $(R_2$ & $S_i)$, 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 *$\underline{N}$egative $\underline{P}$roof $\underline{T}$ree* 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 $\{S_i\}_{i\in I}$ are computed as answers for an admissible initial goal G, one has $\mathcal{P}^- \vdash_{CNPC(\mathcal{D})} G \Rightarrow \bigvee_{i\in I} S_i$ with some witnessing NPT. The next theorem is intended to provide some plausibility to the pragmatic assumption that actual $CFLP$ systems such as *Curry* (Hanus, 2003) or $\mathcal{TOY}$ (López & Sánchez, 1999) are admissible goal solving systems.

Theorem 5. (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: If $(\underline{R} \wedge R')$ & $S \Vdash^{p}_{\mathcal{P},GSC(\mathcal{D})} D$ (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 (López et al., 2004), and a certain selection strategy that only selects atoms descendants of the part R) then $\mathcal{P}^- \vdash_{CNPC(\mathcal{D})} R \;\&\; S \Rightarrow D$ with some witnessing NPT. The proof proceeds by induction on p, using an auxiliary lemma to deal with compound goals whose kernel is a conjunction. □

We have also proved 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 of missing answers.

Theorem 6. (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 2.*

6.1 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 7. (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 4, with intended interpretation $\mathcal{I}_\mathcal{P}$, admissible initial goal G, and finite disjunction of computed answers $D = \bigvee_{i\in I} S_i$. Assume also that the computation has been performed by an admissible goal solving system. Then there exists a 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 with respect to $\mathcal{I}_\mathcal{P}$.*

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 6. By Definition 2, 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 $Sol_{\mathcal{I}_\mathcal{P}}(G) \nsubseteq Sol_\mathcal{D}(D)$, because the disjunction D of computed answers is an incompleteness symptom with respect to $\mathcal{I}_\mathcal{P}$. 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 **(DF)**$_f$ is the only inference rule in the $CNPC(\mathcal{D})$ calculus that depends on the program, while all the other inference rules are correct with respect to arbitrary interpretations. For this reason, abbreviated proof trees will omit the inference steps related to the $CNPC(\mathcal{D})$ inference rules other than **(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 *$\underline{A}$bbreviated $\underline{N}$egative $\underline{P}$roof $\underline{T}$ree* (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 **(DF)**$_f$ inference steps.

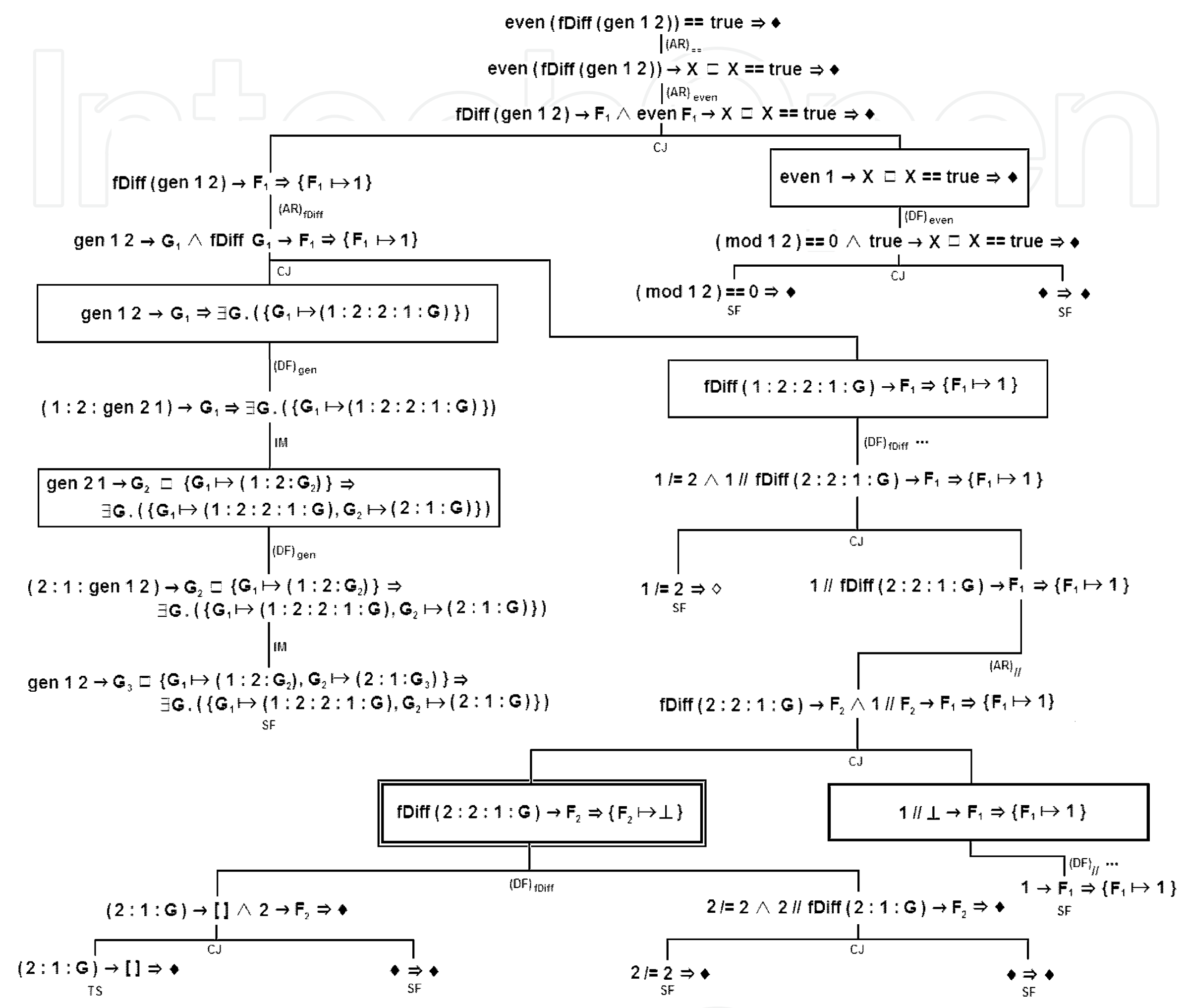

even (fDiff (gen 1 2)) == true ⇒ ♦
(AR)==
even (fDiff (gen 1 2)) → X ⊏ X == true ⇒ ♦
(AR)even
fDiff (gen 1 2) → F1 ∧ even F1 → X ⊏ X == true ⇒ ♦
CJ
fDiff (gen 1 2) → F1 ⇒ {F1 ↦ 1}
(AR)fDiff
gen 1 2 → G1 ∧ fDiff G1 → F1 ⇒ {F1 ↦ 1}
CJ
gen 1 2 → G1 ⇒ ∃G. ({G1 ↦ (1 : 2 : 2 : 1 : G)})
(DF)gen
(1 : 2 : gen 2 1) → G1 ⇒ ∃G. ({G1 ↦ (1 : 2 : 2 : 1 : G)})
IM
gen 2 1 → G2 ⊏ {G1 ↦ (1 : 2 : G2)} ⇒ ∃G. ({G1 ↦ (1 : 2 : 2 : 1 : G), G2 ↦ (2 : 1 : G)})
(DF)gen
(2 : 1 : gen 1 2) → G2 ⊏ {G1 ↦ (1 : 2 : G2)} ⇒ ∃G. ({G1 ↦ (1 : 2 : 2 : 1 : G), G2 ↦ (2 : 1 : G)})
IM
gen 1 2 → G3 ⊏ {G1 ↦ (1 : 2 : G2), G2 ↦ (2 : 1 : G3)} ⇒ ∃G. ({G1 ↦ (1 : 2 : 2 : 1 : G), G2 ↦ (2 : 1 : G)})
SF
even 1 → X ⊏ X == true ⇒ ♦
(DF)even
(mod 1 2) == 0 ∧ true → X ⊏ X == true ⇒ ♦
CJ
(mod 1 2) == 0 ⇒ ♦
SF
♦ ⇒ ♦
SF
fDiff (1 : 2 : 2 : 1 : G) → F1 ⇒ {F1 ↦ 1}
(DF)fDiff ...
1 /= 2 ∧ 1 // fDiff (2 : 2 : 1 : G) → F1 ⇒ {F1 ↦ 1}
CJ
1 /= 2 ⇒ ◇
SF
1 // fDiff (2 : 2 : 1 : G) → F1 ⇒ {F1 ↦ 1}
(AR)//
fDiff (2 : 2 : 1 : G) → F2 ∧ 1 // F2 → F1 ⇒ {F1 ↦ 1}
CJ
fDiff (2 : 2 : 1 : G) → F2 ⇒ {F2 ↦ ⊥}
1 // ⊥ → F1 ⇒ {F1 ↦ 1}
(DF)// ...
1 → F1 ⇒ {F1 ↦ 1}
SF
(DF)fDiff
(2 : 1 : G) → [] ∧ 2 → F2 ⇒ ♦
CJ
(2 : 1 : G) → [] ⇒ ♦
TS
♦ ⇒ ♦
SF
2 /= 2 ∧ 2 // fDiff (2 : 1 : G) → F2 ⇒ ♦
CJ
2 /= 2 ⇒ ♦
SF
♦ ⇒ ♦
SF

Fig. 6. *NPT* for the declarative diagnosis of missing answers

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 *CT*s 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. 6 displays a *NPT* which can be used for the diagnosis of missing answers in the Example 2. Buggy nodes are highlighted by encircling the *aca*s attached to them within double boxes. The *CT* shown in Fig. 7 is the *ANPT* constructed from this *NPT*. In this case, the programmer will judge the root *aca* as *invalid* because he did not expect finite failure. Moreover, from him knowledge of the intended interpretation, he 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. 7) 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. Our last result is a refinement of Theorem 7. It

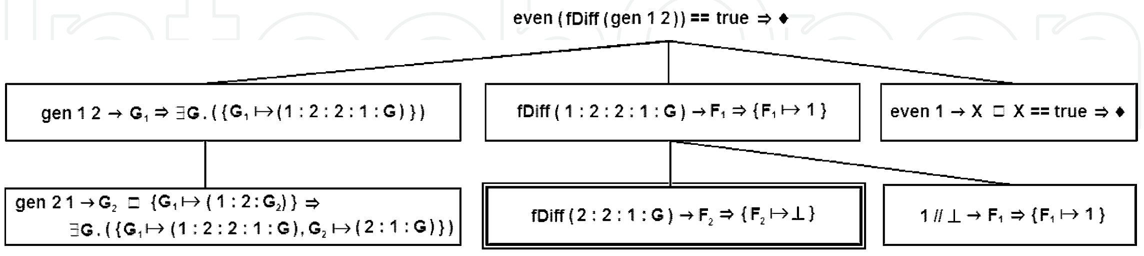

even (fDiff (gen 1 2)) == true ⇒ ♦
gen 1 2 → G₁ ⇒ ∃G.({G₁ ↦ (1:2:2:1:G)})
fDiff (1:2:2:1:G) → F₁ ⇒ {F₁ ↦ 1}
even 1 → X □ X == true ⇒ ♦
gen 2 1 → G₂ □ {G₁ ↦ (1:2:G₂)} ⇒ ∃G.({G₁ ↦ (1:2:2:1:G), G₂ ↦ (2:1:G)})
fDiff (2:2:1:G) → F₂ ⇒ {F₂ ↦ ⊥}
1 // ⊥ → F₁ ⇒ {F₁ ↦ 1}

Fig. 7. *CT* for the declarative diagnosis of missing answers

guarantees that declarative diagnosis with *ANPT*s used as *CT*s leads to the correct detection of incomplete program functions.

Theorem 8. (*ANPT*s Lead to the Diagnosis of Incomplete Functions) *As in Theorem 7, assume that an incompleteness symptom has been observed for a given CFLP($\mathcal{D}$)-program $\mathcal{P}$ as explained in Definition 4, with intended interpretation $\mathcal{I}_{\mathcal{P}}$, admissible initial goal G, and finite disjunction of answers $D = \bigvee_{i \in I} S_i$, 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 with respect to the user's intended interpretation $\mathcal{I}_{\mathcal{P}}$.*

7. A declarative debugging tool of missing answers in $\mathcal{TOY}$

In this section, we discuss the implementation in the $\mathcal{TOY}$ system of a tool based on the debugging method presented in the previous section. 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 with respect to $\mathcal{P}$. 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 with respect to $\mathcal{P}$*.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.

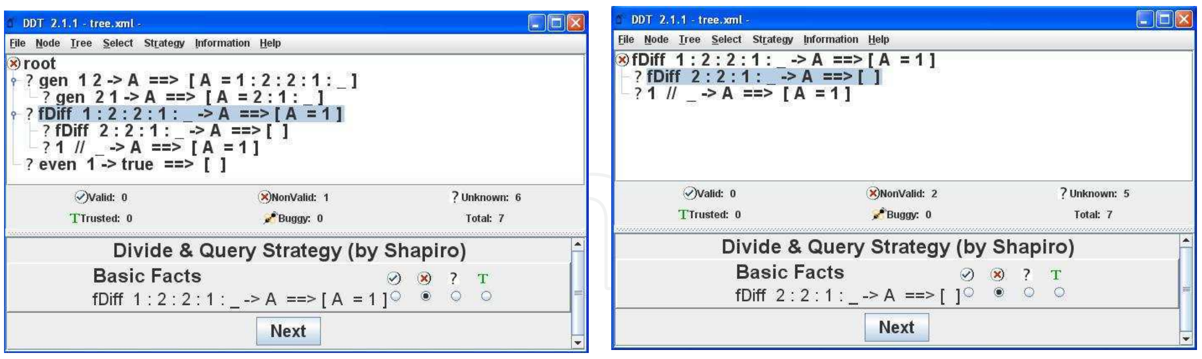
If the computation of answers for G finishes after having collected finitely many answers, the user may decide that there are some missing answers (*incompleteness symptom*, in the terminology of Definition 4) and type the command `/missing` at the system prompt in order to initiate a *debugging session*. The debugger proceeds carrying out the following steps:

1. The object program $\mathcal{P}.pl$ is transformed into a new Prolog program $\mathcal{P}^{\mathcal{T}}.\ pl$. 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 $\mathcal{P}^{\mathcal{T}}$ behaves almost identically to $\mathcal{P}$, being the only difference 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 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 invoking 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 the 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 outputs. 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 with respect to $\mathcal{P}^{\mathcal{T}}$. After each successful computation, the debugger enforces a `fail` in order to trigger the backtracking mechanism and produces 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 (Silva, 2006) 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 6, as responsible for the missing answers. The current implementation of the prototype is available at `http://gpd.sip.ucm.es/rafav/`.



```
DDT 2.1.1 - tree.xml -
File  Node  Tree  Select  Strategy  Information  Help
root
  ? gen 1 2 -> A ==> [ A = 1 : 2 : 2 : 1 : _ ]
    ? gen 2 1 -> A ==> [ A = 2 : 1 : _ ]
  ? fDiff 1 : 2 : 2 : 1 : _ -> A ==> [ A = 1 ]
    ? fDiff 2 : 2 : 1 : _ -> A ==> [ ]
    ? 1 // _ -> A ==> [ A = 1 ]
  ? even 1 -> true ==> [ ]
Valid: 0        NonValid: 1     ? Unknown: 6
Trusted: 0      Buggy: 0        Total: 7
Divide & Query Strategy (by Shapiro)
Basic Facts
fDiff 1 : 2 : 2 : 1 : _ -> A ==> [ A = 1 ]
Next
```

```
DDT 2.1.1 - tree.xml -
File  Node  Tree  Select  Strategy  Information  Help
fDiff 1 : 2 : 2 : 1 : _ -> A ==> [ A = 1 ]
  ? fDiff 2 : 2 : 1 : _ -> A ==> [ ]
  ? 1 // _ -> A ==> [ A = 1 ]
Valid: 0        NonValid: 2     ? Unknown: 5
Trusted: 0      Buggy: 0        Total: 7
Divide & Query Strategy (by Shapiro)
Basic Facts
fDiff 2 : 2 : 1 : _ -> A ==> [ ]
Next
```

Fig. 8. Snapshots of the prototype of missing answers

Fig. 8 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. 7, whose construction as *ANPT* has been explained in Section 6. When displaying an *aca* $f\,\bar{t}_n \rightarrow t \,\square\, S \Rightarrow \bigvee_{i \in I} S_i$, the tool uses list notation for representing the disjunction $\bigvee_{i \in I} S_i$ 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. 7; and if t happens to be a variable X, the case $\{X \mapsto \bot\}$ is omitted from the disjunction $\bigvee_{i \in I} S_i$, so that the user must interpret the *aca* as a collection of 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 with respect to the intended interpretation $\mathcal{I}_{\mathcal{P}_{\texttt{fD}}}$ of $\mathcal{P}_{\texttt{fD}}$, while the statement $\forall X. (\texttt{1 //}\ X \rightarrow A \Rightarrow \{A \mapsto 1\})$ has a different meaning and is not valid in $\mathcal{I}_{\mathcal{P}_{\texttt{fD}}}$.

In the debugging session shown in Fig. 8 the user has selected the *Divide & Query* strategy (Silva, 2006) 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 before). 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 for $\mathcal{P}^{\mathcal{T}}.\ pl$ requires almost the same system resources as for $\mathcal{P}.pl$ 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 *CT*s 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 well 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 *CT*s 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.

8. Conclusions and future work

We have presented a logical and semantic framework for the declarative diagnosis of wrong and missing computed answers in *CFLP*$(\mathcal{D})$, a generic scheme for Constraint Functional-Logic Programming over a given constraint domain $\mathcal{D}$ which combines the expressivity of lazy *FP* and *CLP* languages. The diagnosis technique of wrong answers represents the computation which has produced a wrong computed answer by means of an abridged proof tree whose inspection leads to the discovery of some erroneous program rule responsible for the wrong answer. The logical correctness of the method can be formally proved thanks to the connection between abbreviated proof trees and program semantics. The method for missing answers relies on computation trees 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. Following the *CLP* tradition, we have shown that our computation trees for missing answers 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* (Brassel et al., 2004). However, *aca*s in our setting are not a programming technique. Rather, they serve as logical statements whose falsity reveals incompleteness of computed answers with respect to 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*. 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 debugging tool called $\mathcal{DDT}$, which implements the proposed technique for wrong answers over the domain $\mathcal{R}$ of arithmetic constraints over the real numbers has been implemented as a non-trivial extension of previously existing debugging tools. $\mathcal{DDT}$ provides several practical facilities for reducing the number and the complexity of the questions that are presented to the user during a debugging session. Moreover, a prototype debugger for missing answers under development is available, which implements the method in $\mathcal{TOY}$.

As future work, we plan several improvements of $\mathcal{DDT}$, such as enabling the diagnosis supporting *finite domain constraints* (Estévez et al., 2009; Fernández et al., 2007), and providing new facilities for simplifying the presentation of queries to the user. In this sense, some important pragmatic problems well known for declarative diagnosis tools in *FP* and *CLP* languages also arise in our context: both the *CT*s 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 well 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.

9. References

Alpuente, M., Ballis, D., Correa, F.J. & Falaschi, M. (2003). Correction of Functional Logic Programs, *Proc. ESOP'03*, Springer LNCS, 2003.

Boye, J., Drabent, W. & Małuszyński, J. (1997). Declarative Diagnosis of Constraint Programs: an Assertion-based Approach. *DiSCiPl Delieverable D.WP2.2.M1.1-2*, 1997.

Brassel, B., Hanus, M. & Huch, F. (2004). Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004.

Caballero, R. (2005). A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. *Proc. WCFLP'05*, ACM, pp. 8–13, 2005.

Caballero, R. & Rodríguez-Artalejo, M. (2004). $\mathcal{DDT}$: A Declarative Debugging Tool for Functional Logic Languages. *Proc. FLOPS'04*, Springer LNCS 2998, pp. 70–84, 2004.

Comini, M., Levi, G., Meo, M.C. & Vitiello, G. (1999). Abstract diagnosis. *Journal of Logic Programming* 39 (1–3): 43–93, 1999.

Deransart, P., Hermenegildo, M. & Małuszyński, J. (2000). *Analysis and Visualization tools for Constraint Programming: Constraint Debugging*. Springer LNCS 1870, pp. 151–174, 2000.

Drabent, W., Nadjm-Tehrani, S. & Małuszyński, J. (1989). Algorithmic Debugging with Assertions. In *Harvey Abramson and M.H. Rogers, editors, Meta-programming in Logic Programming*, pp. 501-522. The MIT Press, 1989.

Estévez, S, Hortalá, T., Rodríguez, M. & del Vado, R. (2009). On the cooperation of the constraint domains $\mathcal{H}$, $\mathcal{R}$, and $\mathcal{FD}$ in *CFLP*. *Journal of Theory and Practice of Logic Programming*, vol. 9, pp. 415–527, 2009.

Fernández, A.J., Hortalá-González, M.T., Sáenz-Pérez, F. & del Vado-Vírseda, R. (2007). Constraint functional logic programming over finite domains. *Theory and Practice of Logic Programming*, 7(5):537–582, 2007.

Ferrand, G. (1987). Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method. *Journal of Logic Programming* 4(3), 177-198, 1987.

Ferrand, G., Lesaint, W. & Tessier, A. (2002). Theoretical foundations of value withdrawal explanations for domain reduction. *ENTCS 76*, 2002.

Ferrand, G., Lesaint, W. & Tessier, A. (2003). Towards declarative diagnosis of constraint programs over finite domains. *ArXiv Computer Science e-prints*, 2003.

Hanus, M. (2003). *Curry: an Integrated Functional Logic Language*, Version 0.8, April 15, 2003. Available at:
`http://www-ps.informatik.uni-kiel.de/currywiki/`.

Hermenegildo, M., Puebla, G., Bueno, F. & López-García, P. (2002). Abstract Verification and Debugging of Constraint Logic Programs. *Proc. CSCLP'02*, pp. 1–14, 2002.

Lloyd, J.W. (1987). Declarative Error Diagnosis. *New Generation Computing 5(2)*, 133–154, 1987.

López-Fraguas, F.J.; Rodríguez-Artalejo, M. & del Vado-Vírseda, R. (2006). A New Generic Scheme for Functional Logic Programming with Constraints. *Journal of Higher-Order and Symbolic Computation*, volume 20, numbers 1-2, pages 73-122, June 2007.

López-Fraguas, F.J., Rodríguez-Artalejo, M. & del Vado-Vírseda, R. (2005). Constraint Functional Logic Programming Revisited. *WRLA'04*, ENTCS 117, pp. 5–50, 2005.

López-Fraguas, F.J. & Sánchez-Hernández, J. (1999). $\mathcal{TOY}$: A Multiparadigm Declarative System. In *Proc. RTA'99*, Springer LNCS 1631, pp 244–247, 1999. System and documentation available at `http://toy.sourceforge.net`.

López-Fraguas, F.J., Rodríguez-Artalejo, M. & del Vado-Vírseda, R. (2004). A lazy narrowing calculus for declarative constraint programming. *6th International Conference on PPDP'04*, ACM Press, pp. 43–54, 2004.

Naish, L. (1997). A Declarative Debugging Scheme. *Journal of Functional and Logic Programming (3)*, 1997.

Naish, L. & Barbour, T. (1995). A declarative debugger for a logical-functional language. *DSTO General Document*, 5(2):91–99, 1995.

Nilsson, H. (2001). How to look busy while being as lazy as ever: the Implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.

Nilsson, H. & Fritzson, P. (1994). Algorithmic Debugging of Lazy Funcional Languages. *Journal of Functional Programming*, 4(3):337–370, 1994.

Nilsson, H. & Sparud, J. (1997). The Evaluation Dependence Tree as a basis for Lazy Functional Debugging. *Automated Software Engineering*, 4(2):121–150, 1997.

Pope, B. (2006). A Declarative Debugger for Haskell. *PhD thesis, Department of Computer Science and Software Engineering, University of Melbourne*, 2006.

Pope, B. & Naish, L. (2003). Practical aspects of declarative debugging in Haskell 98. *Proc. PPDP'03*, ACM Press, pp. 230–240, 2003.

Shapiro, E.Y. (1982). *Algorithmic Program Debugging*. The MIT Press, Cambridge, 1982.

Silva, J. (2006). A comparative study of algorithmic debugging strategies. In *LOPSTR'06*, volume 4407 of LNCS, pages 143–159. Springer, 2006.

Tessier, A. & Ferrand, G. (2000). Declarative Diagnosis in the CLP Scheme. In P. Deransart, M. Hermenegildo, J. Małuszyński (eds.), *Analysis and Visualization Tools for Constraint Programming*, Chapter 5, pp. 151–174. Springer LNCS 1870, 2000.