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, CTs) in place of traces. CTs are built a posteriori to represent the structure of a computation whose top-level outcome is regarded as a symptom of the unexpected behavior by the user. 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 (Deransard 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 CTs 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 CTs can cause excessive overhead in the case of computations that demand a big amount of computer storage. As a remedy, techniques for piecemeal construction of CTs have been considered; see (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(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 D given as parameter, and supports a powerful combination of functional and constraint logic programming over D. Sound and complete goal solving procedures for the CFLP(D) scheme have been obtained (López et al., 2004). Moreover, useful instances of this scheme have been implemented in the TOY system (López & Sánchez, 1999) and tested in practical applications (Fernández et al., 2007). Borrowing ideas from CFLP(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 CTs. Our aim is to achieve a natural combination of previous approaches that were independently developed for the CLP(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(D)-system whose computed answers are logical consequences of the program in the sense of CFLP(D) semantics. We also present a practical debugger called DDT, developed as an extension of previously existing but less powerful tools (Caballero, 2005; Caballero & Rodríguez, 2004). DDT implements the proposed diagnosis method for CFLP(R)-programming in the TOY system (López & Sánchez, 1999) using the domain 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(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(D)-system. Section 5 describes the debugging tool DDT for wrong answers. Section 6 presents the abbreviated proof trees used as CTs 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(D) scheme, we consider the following program fragment written in TOY (López & Sánchez, 1999), a programming system which supports several instances of the CFLP(D) scheme:

**Example 1. (Debugging Wrong Answers in TOY)**

```lisp
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 TOY

In this example (see Fig. 1), TOY is used to implement the instance CFLP(\(R\)) of the CFLP(D) scheme, with the parameter \(D\) replaced by the real numbers domain \(R\), 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 TOY:
The goal asks for the membership of a generic point \((X,Y)\) to the intersection of the two rectangles \((\text{rect} \ (20,20) \ 50 \ 20)\) and \((\text{rect} \ (5,5) \ 30 \ 40)\), 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 \TOY system has solved the goal by a combination of lazy narrowing and constraint solving; the computed answer consists of the substitution \(R \rightarrow \text{true}\) and three constraints imposed on the variables \(X\) and \(Y\). 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 \text{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 \text{rect} as responsible for the \text{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

\[
\text{To y> /cflpr} \\
\text{Toy(R)> intersect (head (ladder (70,40) -50 -20))} \\
(\text{head (ladder (35,45) -30 -40))} \ (X,Y) == R \\
\{ R -> true \} \{ Y <= 5, X >= 2.0E+01, X <= 35 \}
\]

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 \text{rect} yields the result \text{false} whenever \(LX\) or \(LY\) is bound to a negative number, wrong answers including the substitution \(R \rightarrow \text{false}\) will be computed. Moreover, other answers including the substitution \(R \rightarrow \text{true}\) will be expected by the user but \text{missing} to occur among the computed answers.

The traditional approach to declarative debugging in the \CLP(D) scheme includes the diagnosis of both \text{wrong} and \text{missing} computed answers (Tessier & Ferrand, 2000). Now, we motivate our approach for the declarative debugging of \text{missing answers} in the \CFLP(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 \(R \rightarrow \text{false}\) and various constraints imposed on \(X\) and \(Y\).
Example 2. (Debugging Missing Answers in TOY)

The following small CFLP(H)-program $P_{FD}$, written in TOY syntax over the Herbrand domain $H$ 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 TOY system, which treats all the program defined functions as possibly non-deterministic.

```
iprivileged 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

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]
   gen XY = 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 [d1,d2,d2,d1,d1,d2,...], where the elements d1 and d2 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 $G_{FD}$:even (fDiff (gen 1 2)) == true for $P_{FD}$ 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 $P_{FD}$, the expression (fDiff (gen 1 2)) would return only the numeric value 1, and therefore the goal $G_{FD}$ 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 incompleteness symptom (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(D) programming scheme

In this section we summarize the essentials of the CFLP(D) scheme (López et al., 2006) for lazy Constraint Functional-Logic Programming over a parametrically given constraint domain $D$, 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 $\Sigma_\bot$ for the result of extending $DC^0$ with the special symbol $\bot$, 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 $V$ of variables $X, Y, \ldots$ and a set $U$ of primitive elements $u, v, \ldots$ (as e.g. the set $\mathbb{R}$ of the real numbers) mutually disjoint and disjoint from $\Sigma_\bot$. Expressions $e \in Exp_\bot(U)$ have the following syntax:

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

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

$$t ::= \bot \mid u \mid X \mid (c \overline{t_m}) \mid (f \overline{t_m})$$

where $u \in U, X \in V, c \in DC^n$ 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 $(rect (5,5) 3040)$ we met when discussing Example 1 is a functional data value represented as pattern$^2$.

As usual, we define substitutions $\sigma \in Sub_\bot(U)$ as mappings $\sigma : V \to Pat_\bot(U)$ extended to $\sigma : Exp_\bot(U) \to Exp_\bot(U)$ in the natural way. By convention, we write $e\sigma$ instead of $\sigma(e)$ for any $e \in Exp_\bot(U)$, and $\sigma\theta$ for the composition of $\sigma$ and $\theta$. A substitution $\sigma$ 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 $U$ and primitive functions $PF \subseteq FS$ is any structure $\mathcal{D} = \langle D_U, \{p^D \mid p \in PF\} \rangle$ with carrier set $D_U$ the set of ground patterns (i.e., without variables) over $U$ and interpretations $p^D \subseteq D^n_U \times D_U$ of each $p \in PF^n$ satisfying the technical monotonicity, antimonotonicity, and radicality requirements given in (López et al., 2006). We use the notation $p^D t_n \rightarrow t$ to indicate that $(\overline{t_n}, t) \in p^D$.

Constraints over a given constraint domain $\mathcal{D}$ are logical statements built from atomic constraints by means of logical conjunction $\land$ and existential quantification $\exists$. Atomic constraints can have the form $\top$ (standing for truth), $\bot$ (standing for falsity), or $p\overline{e_n} \rightarrow ! t$, meaning that the primitive function $p \in PF^n$ with parameters $\overline{e_n} \in Exp_\bot(U)$ returns a total result $t \in Pat_\bot(U)$ (i.e, with no occurrences of $\bot$). 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.
A Semantic Framework for the Declarative Debugging of Wrong and Missing Answers in Declarative Constraint Programming

form $\Diamond$, $\Diamond$ or $p \text{I}_n \rightarrow t$ with $\text{I}_n \in \text{Pat}_\bot(\mathcal{U})$ are called primitive constraints. In the sequel, we use the notation $P\text{Con}_\bot(\mathcal{D})$ for the set of primitive constraints over $\mathcal{D}$ and $D\text{Con}_\bot(\mathcal{D})$ 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 $D_\mathcal{R}$ of ground patterns over $\mathcal{R}$ and the primitives defined below:

1. $\text{eq}_{\mathcal{R}}$, equality primitive for real numbers, such that: $\text{eq}_{\mathcal{R}}^R u u \rightarrow$ true for all $u \in \mathcal{R}$; $\text{eq}_{\mathcal{R}}^R u v \rightarrow$ false for all $u, v \in \mathcal{R}, u \neq v$; $\text{eq}_{\mathcal{R}}^R t s \rightarrow \bot$ otherwise.
2. $\text{seq}$, strict equality primitive for ground patterns over the real numbers, such that: $\text{seq}_{\mathcal{R}}^R t t \rightarrow$ true for all total $t \in D_\mathcal{R}$; $\text{seq}_{\mathcal{R}}^R t s \rightarrow$ false for all $t, s \in D_\mathcal{R}$ such that $t, s$ have no common upper bound with respect to the information ordering introduced in (López et al., 2006); $\text{seq}_{\mathcal{R}}^R t s \rightarrow \bot$ otherwise. In the sequel, $e_1 \equiv e_2$ abbreviates $\text{seq}_{\mathcal{R}}^R e_1 e_2 \rightarrow !$ true.
3. $+,-,\ast$, for addition, subtraction and multiplication, such that: $x +_{\mathcal{R}}^R y \rightarrow x +^R y$ for all $x, y \in \mathcal{R}$; $t +_{\mathcal{R}}^R s \rightarrow \bot$ whenever $t \notin \mathcal{R}$ or $s \notin \mathcal{R}$; and analogously for $-_{\mathcal{R}}$ and $\ast_{\mathcal{R}}$.
4. $<,\leq,>,\geq$, for numeric comparisons, such that: $x <_{\mathcal{R}}^R y \rightarrow$ true for all $x, y \in \mathcal{R}$ with $x <^R y$; $x <_{\mathcal{R}}^R y \rightarrow$ false for all $x, y \in \mathcal{R}$ with $x \geq_{\mathcal{R}}^R y$; $t <_{\mathcal{R}}^R s \rightarrow \bot$ whenever $t \notin \mathcal{R}$ or $s \notin \mathcal{R}$; 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 $\text{Val}_\bot(\mathcal{D})$ 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 $\pi \in P\text{Con}_\bot(\mathcal{D})$ can be defined in a natural way as a subset $\text{Sol}_\mathcal{D}(\pi) \subseteq \text{Val}_\bot(\mathcal{D})$; see (López et al., 2006) for details. Moreover, the set of solutions of $\Pi \subseteq P\text{Con}_\bot(\mathcal{D})$ is defined as $\text{Sol}_\mathcal{D}(\Pi) = \cap_{\pi \in \Pi} \text{Sol}_\mathcal{D}(\pi)$.

3.3 Constraint functional-logic programming

For any given constraint domain $\mathcal{D}$, a $\text{CFLP}(\mathcal{D})$-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 $R : f \text{I}_n \rightarrow r \leftarrow \Delta$ (abbreviated as $f \text{I}_n \rightarrow r$ if $\Delta$ is empty) and is required to satisfy the conditions listed below:

1. The left-hand side $f \text{I}_n$ is a linear expression (i.e., there is no variable having more than one occurrence), and for all $1 \leq i \leq n$, $t_i \in \text{Pat}_\bot(\mathcal{U})$ are total patterns. The right-hand side $r \in \text{Exp}_\bot(\mathcal{U})$ is also total.
2. $\Delta \subseteq D\text{Con}_\bot(\mathcal{D})$ 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 $\text{TOLY}$ program presented in Example 1 can be viewed as an example of $\text{CFLP}(\mathcal{R})$-program written in $\text{TOLY}$’s syntax. The reader is referred to (López et al., 2006) for more explanations and examples in other constraint domains.

3 In practice, $\text{TOLY}$ and similar languages require program rules to be well-typed in a polymorphic type system. However, the $\text{CFLP}(\mathcal{D})$ scheme can deal also with untyped programs. Well-typedness is viewed as an additional requirement, not as part of program semantics.
The intended use of programs is to perform computations by solving goals proposed by the user. An admissible goal for a given CFLP(D)-program must have the form \( G : \exists \overline{U}. (P \land \Delta) \), 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 \( \text{fvar}(G) \)), \( P \) is a finite conjunction of so-called productions of the form \( e \rightarrow s \) fulfilling the admissibility conditions given in (López et al., 2006), with \( e \in \text{Exp}_\perp(\overline{U}) \) and \( s \in \text{Pat}_\perp(\overline{U}) \) intended to mean that \( e \) can be evaluated to \( s \), and \( \Delta \subseteq \text{DCon}_\perp(D) \) 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 \( \Delta \) without occurrences of existential variables), and solved goals (also called solved forms) of the form \( S : \exists \overline{U}. (\sigma \land \Pi) \), where \( \sigma \) is a finite set of productions \( X \rightarrow t \) or \( s \rightarrow Y \) interpreted as the variable bindings of an idempotent substitution and \( \Pi \subseteq \text{PCon}_\perp(D) \) is a finite conjunction of total primitive constraints. Finally, a goal solving system for CFLP(D) is expected to accept a program \( P \) 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 \( R \rightarrow \text{true} \land X \leq 35 \land X \geq 20 \land Y \leq 5 \).

Goal solving systems can be implementations of CFLP languages such as Curry (Hanus, 2003) or TÖY (López & Sánchez, 1999), or formal goal solving calculi including recent proposals such as the CDNC(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 TÖY system.

### 3.4 Standardized programs and negative theories

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

\[
(f_{\text{Diff}})^-_{\text{ED}} : \forall L, F. (f_{\text{Diff}} L \rightarrow F \Rightarrow \\
\exists X. (L \rightarrow [X] \land X \rightarrow F) \lor \\
\exists X, Y, Zs. (L \rightarrow (X : Y : Zs) \land X /= Y \land X \lor \text{Diff}^-(Y : Zs) \rightarrow F) \lor \\
(\bot \rightarrow F))
\]
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 acas. 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 $G_{\text{ED}}$ for program $P_{\text{ED}}$ is even $(\text{fDiff } (\text{gen } 1 \text{ } 2)) = \text{true} \Rightarrow ♦$ stating that this goal has (unexpectedly) failed. The acas at internal nodes in our computation trees have the form $f\bar{t}_n \Rightarrow t \square S \Rightarrow \bigvee_{i \in I} S_i$, asserting that the disjunction of computed answers $\bigvee_{i \in I} S_i$ covers all the solutions for the intermediate goal $G' : f\bar{t}_n \rightarrow t \square S$. Note that $G'$ asks for the solutions of the production $f\bar{t}_n \rightarrow t$ which satisfy the constraint store $S$. The acas 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 TOLy.

3.6 Declarative semantics

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

1. c-productions $e \rightarrow t \leftarrow \Pi$, with $e \in \text{Exp}_\bot(\mathcal{U})$ and $t \in \text{Pat}_\bot(\mathcal{U})$, intended to mean that $e$ can be evaluated to $t$ if $\Pi$ holds (if $\Pi$ is empty they boil down to unconstrained productions written as $e \rightarrow t$). As a particular kind of c-productions useful for debugging we distinguish c-facts $f\bar{t}_n \rightarrow t \leftarrow \Pi$ with $f \in \text{DF}^n$. A c-production is called trivial iff $t = \bot$ or $\text{Sol}_\Pi(D) = \emptyset$.

2. c-atoms $p\bar{t}_n \rightarrow !t \leftarrow \Pi$, with $p \in \text{PF}^n$ and $t$ total (if $\Pi$ is empty they boil down to unconstrained atoms written as $p\bar{t}_n \rightarrow !t$). A c-atom is called trivial iff $\text{Sol}_\Pi(D) = \emptyset$.

In the sequel, we use $\varphi$ to denote any c-statement. A c-interpretation over $D$ is defined as any set $\mathcal{I}$ of c-facts including all the trivial c-facts and closed under $D$-entailment, a generalization of the entailment notion introduced in (Caballero & Rodriguez, 2004) to arbitrary constraint domains. We write $\mathcal{I} \models_D \varphi$ to indicate that the c-statement $\varphi$ (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 $D$.

**Definition 1. (Interpretation-Dependent Semantic Notions)**

1. The set of solutions of $\delta \in \text{DCon}_\bot(D)$ is a subset $\text{Sol}_\mathcal{I}(\delta) \subseteq \text{Val}_\bot(D)$ defined as follows:
   (a) $\text{Sol}_\mathcal{I}(\pi) = \text{Sol}_D(\pi)$, for any $\pi \in \text{PCon}_\bot(D)$.
(b) \( \text{Sol}_I(\delta) = \{ \eta \in \text{Val}_\bot(D) \mid I \vdash \delta \eta \} \), for any \( \delta \in D_{\text{Con}_\bot(D) \setminus P_{\text{Con}_\bot(D)}} \).

The set of solutions of a set of constraints \( \Delta \subseteq D_{\text{Con}_\bot(D)} \) is defined as \( \text{Sol}_I(\Delta) = \cap_{\delta \in \Delta} \text{Sol}_I(\delta) \).

2. The set of solutions of a production \( e \rightarrow t \) is a subset \( \text{Sol}_I(e \rightarrow t) \subseteq \text{Val}_\bot(D) \) defined as \( \text{Sol}_I(e \rightarrow t) = \{ \eta \in \text{Val}_\bot(D) \mid I \vdash \eta \rightarrow t \eta \} \). The set of solutions of a set of productions \( P \) is defined as \( \text{Sol}_I(P) = \bigcap_{(e \rightarrow t) \in P} \text{Sol}_I(e \rightarrow t) \).

3. The set of solutions of an admissible goal \( G : \exists X. (P \sqcap \Delta) \) is a subset \( \text{Sol}_I(G) \subseteq \text{Val}_\bot(D) \) defined as follows: \( \text{Sol}_I(G) = \{ \eta \in \text{Val}_\bot(D) \mid \eta' \in \text{Sol}_I(P) \cap \text{Sol}_I(\Delta) \text{ for some } \eta' \text{ such that } \eta'(X) = \eta(X) \text{ for all } X \notin U \} \).

For primitive constraints one can easily check that \( \text{Sol}_I(\Pi) = \text{Sol}_D(\Pi) \). Moreover, we note that \( \text{Sol}_I(S) = \text{Sol}_D(S) \) for every solved form \( S \).

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

1. \( I \) is a model of \( P \) (in symbols, \( I \models D P \)) iff every constrained program rule \( (f \bar{t}_n \rightarrow r \triangleleft \Delta) \in P \) is valid in \( I \): for any ground substitution \( \eta \in \text{Sub}_\bot(U) \) and \( t \in \text{Pat}_\bot(U) \) ground such that \( (f \bar{t}_n \rightarrow r \triangleleft \Delta) \eta \) is ground, \( I \vdash_D \Delta \eta \) and \( I \vdash_D r \eta \rightarrow t \eta \) one has \( I \vdash_D (f \bar{t}_n) \eta \rightarrow t \) (or equivalently, \( (f \bar{t}_n) \eta \rightarrow t \) is in \( I \)).

2. A solved form \( S \) is a semantically valid answer for a goal \( G \) with respect to a program \( P \) (in symbols, \( P \models_D G \iff \text{Sol}_D(S) \subseteq \text{Sol}_I(G) \) for all \( I \models_D P \)).

3. \( I \) is a model of \( P^- \) iff every axiom \( (f) \downarrow_P : (f \bar{X}_n \rightarrow Y \Rightarrow D_f) \in P^- \) satisfies \( \text{Sol}_I(f \bar{X}_n \rightarrow Y) \subseteq \text{Sol}_I(D_f) \). When this inclusion holds, we say that \( (f) \downarrow_P \) is valid in \( I \), or also that \( f \)’s definition as given in \( P \) is complete with respect to \( I \).

4. The aca \( G \Rightarrow \bigvee_{i \in I} S_i \) is a logical consequence of \( P^- \) iff \( \text{Sol}_I(G) \subseteq \bigcup_{i \in I} \text{Sol}_D(S_i) \) for any model \( I \) of \( P^- \). When this happens, we also say that the disjunction of answers \( \bigvee_{i \in I} S_i \) is complete for \( G \) with respect to \( P \).

4. Declarative debugging of wrong answers in CFLP\((D)\)

In this section, we present the logical and semantic framework of the declarative diagnosis method of wrong answers for CFLP\((D)\) and prove its logical correctness. In what follows, we assume that a constraint domain \( D \) and a CFLP\((D)\)-program \( P \) 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 \( I \), which is a c-interpretation expected to satisfy \( I \models_D P \), unless \( P \) is incorrect. For instance, \( \text{rect} \ (X, Y) \text{LXLY} \ (A, B) \rightarrow \text{false} \iff A < X \land LX > 0 \land LY > 0 \) could belong to the intended model \( I \) 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 \( I \models_D P \) 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 \( I \). 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 \( I \) 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 \( \text{Sol}_D(S) \not\subseteq \text{Sol}_I(G) \).
2. \( \mathcal{P} \) is **incorrect** with respect to \( \mathcal{I} \) iff there exists some program rule \( (f\bar{t}_n \rightarrow r \leftarrow \Delta) \in \mathcal{P} \) (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 \( \text{CFLP}(\mathcal{D}) \) 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, \( \text{CT} \)) representing the computation. In our setting we will obtain the \( \text{CT} \) from a logical proof \( \mathcal{P} \models_{\text{CPPC}(\mathcal{D})} G \leftarrow S \) which derives the statement \( G \leftarrow S \) from the program \( \mathcal{P} \) in the Constraint Positive Proof Calculus (shortly \( \text{CPPC}(\mathcal{D}) \)) given by the inference rules in Fig. 2. We will say that the \( \text{CPPC}(\mathcal{D}) \)-proof witnesses the computed answer.

Most of these inference rules have been borrowed from the proof theory of \( \text{CRWL}(\mathcal{D}) \), a Constraint ReWriting Logic which characterizes the semantics of \( \text{CFLP}(\mathcal{D}) \) programs (López et al., 2006). The main novelties in \( \text{CPPC}(\mathcal{D}) \) are the addition of rule \( \text{EX} \) (to deal with existential quantifiers in computed answers) and a reformulation of rule \( \text{DF}_P \), which is presented as the consecutive application of two inference steps named \( \text{AR}_f \) and \( \text{FA}_f \), which cannot be applied separately. The purpose of this composite inference is to introduce the c-facts \( f\bar{t}_n \rightarrow t \leftarrow \Pi \) at the conclusion of inference \( \text{FA}_f \), called boxed c-facts in the sequel. As we will see, only boxed c-facts will appear at the nodes of \( \text{CTs} \) obtained from \( \text{CPPC}(\mathcal{D}) \)-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 \( \text{EX} \) (resp. \( \Delta \leftarrow \Pi \) in rule \( \text{DF}_P \)) must be understood as a shorthand for several premises \( \alpha \leftarrow \Pi \), one for each atomic \( \varphi \) in \( G\sigma \) (resp. \( \Delta \)). Moreover, rule \( \text{PF} \) depends on the side condition \( \text{Sol}_D(\Pi) \subseteq \text{Sol}_D(p\bar{t}_n \rightarrow t) \) which is true iff \( p^D \bar{t}_n\eta \rightarrow \eta \) holds for all \( \eta \in \text{Sol}_D(\Pi) \). Some other inference rules in Fig. 2 have similar conditions.
EX Existential  \[ G \sigma \subseteq \Pi \]  
if \( \text{fv}(G) \cap \Pi = \emptyset \).

TI Trivial Inference  \[ \phi \]  
if \( \phi \) is a trivial c-statement.

RR Restricted Reflexivity  \[ t \rightarrow t \subseteq \Pi \]  
if \( t \in \mathcal{U} \cup \mathcal{V} \).

SP Simple Production  \[ s \rightarrow t \subseteq \Pi \]  
if \( s \in \text{Pat} \bot (\mathcal{U}) \), \( s \in \mathcal{V} \) or \( t \in \mathcal{V} \), and \( \text{Sol}_D(\Pi) \subseteq \text{Sol}_D(s \rightarrow t) \).

DC Decomposition  \[ e_1 \rightarrow t_1 \subseteq \Pi \ldots e_m \rightarrow t_m \subseteq \Pi \]  
if \( h \sigma_m \rightarrow h t_m \subseteq \Pi \).

IR Inner Reduction  \[ e_1 \rightarrow t_1 \subseteq \Pi \ldots e_n \rightarrow t_n \subseteq \Pi \]  
if \( h \sigma_m \notin \text{Pat} \bot (\mathcal{U}) \), \( X \in \mathcal{V} \), and \( \text{Sol}_D(\Pi) \subseteq \text{Sol}_D(h t_m \rightarrow X) \).

PF Primitive Function  \[ e_1 \rightarrow t_1 \subseteq \Pi \ldots e_n \rightarrow t_n \subseteq \Pi \]  
if \( p \in \text{PF}^n \), \( t_i \in \text{Pat} \bot (\mathcal{U}) \) (1 ≤ i ≤ n), \( \text{Sol}_D(\Pi) \subseteq \text{Sol}_D(p t_n \rightarrow t) \).

DFP \( \mathcal{P} \)-Defined Function  \[ e_1 \rightarrow t_1 \subseteq \Pi \ldots e_n \rightarrow t_n \subseteq \Pi \]  
if \( f \in \text{DF}^n \) (k > 0), \( (f t_n \rightarrow r \subseteq \Delta) \in [\mathcal{P}]^\bot \equiv \{ R \theta \mid R \in \mathcal{P} \}, \theta \in \text{Sub} \bot (\mathcal{U}) \}, \) and \( s \in \text{Pat} \bot (\mathcal{U}) \).

AC Atomic Constraint  \[ e_1 \rightarrow t_1 \subseteq \Pi \ldots e_n \rightarrow t_n \subseteq \Pi \]  
if \( p \in \text{PF}^n \), \( t_i \in \text{Pat} \bot (\mathcal{U}) \) (1 ≤ i ≤ n), \( \text{Sol}_D(\Pi) \subseteq \text{Sol}_D(p t_n \rightarrow t) \).

Fig. 2. The Constraint Positive Proof Calculus CPPC(\( \mathcal{D} \))
A Semantic Framework for the Declarative Debugging of Wrong and Missing Answers in Declarative Constraint Programming

Any CPPC(\(D\))-derivation \(\mathcal{P} \vdash_{\text{CPPC(}D\text{)}} G \Leftarrow S\) can be depicted in the form of a _Positive Proof Tree_ over \(D\) (shortly, _PPT(\(D\))_), 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(\(D\)) inference rule. In particular, the statement at the root must be inferred using rule EX, which is then applied nowhereelse in the proof tree. Fig. 3 shows a PPT(\(R\)) representing a CPPC(\(R\))-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(\(D\))-sound iff for any computed answer \(S\) obtained for an initial goal \(G\)

![Fig. 3. A Positive Proof Tree in CPPC(\(R\))]
Theorem 1. (Existence of CPPC(D)-Sound Goal Solving Systems) The goal solving calculus CDNC(D) given in (López et al., 2004) is CPPC(D)-sound.

Proof. Straightforward adaptation of the soundness theorem for CDNC presented in (López et al., 2004).

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

Theorem 2. (Semantic Correctness of the CPPC(D) Calculus) If G is an initial goal for P and S is a solved goal s.t. P ⊨ CPPC(D) G ⇐ S then P ⊨ P⊢ G ⇒ S.

Proof. For each of the inference rules EX, ARf, and FAf, we prove that an arbitrary model I |= P such that the premises of the rule are valid in I, also verifies that the conclusion of the rule is valid in I. Similar proofs for the other inference rules in CFLP(D) can be found in (López et al., 2006).

- The rule EX is semantically correct. Let I be an arbitrary model of P such that I |= P Gσ ⇐ Π, i.e., SolD(Π) ⊆ SolI(Gσ). We prove that I |= P G = ∃!Π (σ □ Π), i.e., SolD(∃!Π (σ □ Π)) ⊆ SolI(G). Let η ∈ SolD(∃!Π (σ □ Π)). By the syntactic form of solved goals, η ∈ SolD(∃!Π (Xn ↠ tn ∧ sm ↠ Ym □ Π)) and η ∈ SolD(∃!Π (Xn = tn ∧ Ym = sm □ Π)). By applying Definition 1, there exists η′ ∈ ValI(D) such that η′ = Π η η′ ∈ SolD(Xn = tn ∧ Ym = sm □ Π)), and therefore, η′ ∈ SolD(∃!Π (σ □ Π)) (i.e., η′ ∈ SolD(σ)) and η′ ∈ SolD(Π). Since by induction hypothesis SolD(Π) ⊆ SolI(Gσ), it follows that η′ ∈ SolI(G). Moreover, since η′ ∈ SolI(σ), we obtain η′ ∈ SolI(G). In consequence, there exists η′ ∈ ValI(D) such that η′ = Π η and η′ ∈ SolI(G). Finally, using the condition of applicability f/val(G) ∩ ∅ associated to the rule EX, we can conclude that η ∈ SolI(G).

- The rule ARf is semantically correct. Let I be an arbitrary model of P such that I |= P ei i → ti ⇐ Π for each 1 ≤ i ≤ n (i.e., SolD(Π) ⊆ SolI(ei i → ti) for each 1 ≤ i ≤ n), I |= P fni → s ⇐ Π (i.e., SolD(Π) ⊆ SolI(fni → s)) and I |= P σa{k} → s ⇐ Π (i.e., SolD(Π) ⊆ SolI(σa{k} → t)). We prove that I |= P fen{a} → t ⇐ Π, i.e., SolD(Π) ⊆ SolI(fen{a} → t). Let η ∈ SolD(Π). We have then η ∈ SolI(ei i → ti) for each 1 ≤ i ≤ n, and by Definition 1, I |= P ei i → ti for each 1 ≤ i ≤ n. Analogously, η ∈ SolI(fni → s), by Definition 1, I |= P fni → sη, and by the Conservation Property (see López et al., 2006) for details), (fni → sη) ∈ I. Analogously, η ∈ SolI(a{k} → t) and by Definition 1, I |= P (a{k}η) → tη. But then, by applying of the rule DF (see López et al., 2006) for details), we have that I |= P fen{a}{a} → tη. From Definition 1, we obtain finally η ∈ SolI(fen{a} → t).
• The rule $\text{FA}_{\Delta}$ is semantically correct. By definition of $[\mathcal{P}]_{\perp}$, there are $(f\overline{T}_n \rightarrow r' \leftarrow \Delta') \in \mathcal{P}$ and $\theta \in \text{Sub}_\perp(U)$ such that $(f\overline{T}_n \rightarrow r' \leftarrow \Delta')\theta \equiv (f\overline{T}_n \rightarrow r \leftarrow \Delta)$. Let $\mathcal{I}$ be an arbitrary model of $\mathcal{P}$ such that $\mathcal{I} \models_{\mathcal{D}} \Delta \triangleleft \Pi$ (i.e., $\text{Sol}_{\mathcal{D}}(\Pi) \subseteq \text{Sol}_\mathcal{I}(\Delta)$) and $\mathcal{I} \models_{\mathcal{D}} r \rightarrow s \triangleleft \Pi$ (i.e., $\text{Sol}_{\mathcal{D}}(\Pi) \subseteq \text{Sol}_\mathcal{I}(f\overline{T}_n \rightarrow s)$). We prove that $\mathcal{I} \models_{\mathcal{D}} f\overline{T}_n \rightarrow s \triangleleft \Pi$, i.e., $\text{Sol}_{\mathcal{D}}(\Pi) \subseteq \text{Sol}_\mathcal{I}(f\overline{T}_n \rightarrow s)$. Let $\eta \in \text{Sol}_{\mathcal{D}}(\Pi)$. Then we have $\eta \in \text{Sol}_\mathcal{I}(\Delta)$, and by Definition 1, $\mathcal{I} \models_{\mathcal{D}} \Delta \theta \eta$, and also, $\mathcal{I} \models_{\mathcal{D}} \Delta' \theta \eta$. Analogously, $\eta \in \text{Sol}_\mathcal{I}(r \rightarrow s)$, and by Definition 1, $\mathcal{I} \models_{\mathcal{D}} r \eta \theta \eta \rightarrow s \eta$, and also, $\mathcal{I} \models_{\mathcal{D}} r' \theta \eta \rightarrow s \eta$. We have then $(f\overline{T}_n \rightarrow r' \leftarrow \Delta') \in \mathcal{P}, \theta \eta \in \text{Sub}_\perp(U)$ ground substitution and $s \eta \in \text{Pat}_\perp(U)$ ground such that $(f\overline{T}_n \rightarrow r' \leftarrow \Delta')\theta \eta \equiv (f\overline{T}_n \rightarrow r \leftarrow \Delta)\eta$ is ground, $\mathcal{I} \models_{\mathcal{D}} \Delta' \theta \eta$ and $\mathcal{I} \models_{\mathcal{D}} r' \theta \eta \rightarrow s \eta$. Since $\mathcal{I}$ is a model of $\mathcal{P}$, by applying Definition 2, we obtain $((f\overline{T}_n)\theta \eta \rightarrow s \eta) \in \mathcal{I}$, i.e., $((f\overline{T}_n)\eta \rightarrow s \eta) \in \mathcal{I}$, or also, $(f\overline{T}_n \rightarrow s)\eta \in \mathcal{I}$. Finally, by applying the Conservation Property (see López et al., 2006) for details, it is equivalent to $\mathcal{I} \models_{\mathcal{D}} (f\overline{T}_n \rightarrow s)\eta$, and by Definition 1, we can conclude that $\eta \in \text{Sol}_\mathcal{I}(f\overline{T}_n \rightarrow s)$. 

\[ \square \]

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 $\text{CPPC}(\mathcal{D})$-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 $\text{CPPC}(\mathcal{D})$-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 $\text{CPPC}(\mathcal{D})$-soundness of the goal solving system, we know that $\mathcal{P} \models_{\text{CPPC}(\mathcal{D})} G \triangleleft S$. Then, from Theorem 2 we obtain $\mathcal{P} \models_{\mathcal{D}} G \triangleleft S$, i.e., $\text{Sol}_{\mathcal{D}}(S) \subseteq \text{Sol}_\mathcal{I}(G)$ 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 $\text{Sol}_{\mathcal{D}}(S) \not\subseteq \text{Sol}_\mathcal{I}(G)$ 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 $\mathcal{I}$.

\[ \square \]

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 $\text{CPPC}(\mathcal{D})$ proof trees as $\text{CTs}$. Since $\text{DF}_{\mathcal{D}}$ is the only inference rule in the $\text{CPPC}(\mathcal{D})$ calculus that depends on the program, abbreviated proof trees will omit the inference steps related to all the other $\text{CPPC}(\mathcal{D})$ rules. More precisely, given a $\text{PPT}(\mathcal{D})$ $\mathcal{T}$, its associated *Abbreviated Positive Proof Tree* over $\mathcal{D}$ (shortly, $\text{APPT}(\mathcal{D})$) $\mathcal{AT}$ 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 $\text{DF}_{\mathcal{D}}$ inference steps.
A node in an APPT(D) is called a buggy node iff the c-statement at the node is not valid in the intended interpretation I, while all the c-statements at the children nodes are valid in I. Our last theorem guarantees that declarative diagnosis with APPT(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(D) witnessing \( \mathcal{P} \vdash_{\text{CPPC}(D)} G \leftarrow S \) (which must exist due to CPPC(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 \( \mathcal{T}O\mathcal{Y} \)

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

![Fig. 4. The APPT(\( \mathcal{R} \)) corresponding to the PPT(\( \mathcal{R} \)) of Fig. 3](image)

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 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 DDT which did not yet support constraints. After selecting the divide and query strategy, which usually leads to shorter sessions, DDT asks about the validity of the following node:

![Divide & Query Strategy](image)

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](image)

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](image)

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 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 TOY, available at [http://toy.sourceforge.net](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 DDT can be found in (Caballero, 2005; Caballero & Rodríguez, 2004).
SF Solved Form
\[ R \square S \Rightarrow D \quad \text{if } \text{Sol}_D(S) \subseteq \text{Sol}_D(D). \]

CJ Conjunction
\[ R_1 \square S \Rightarrow \bigvee_{i \in I} \exists Z_i. S_i \ldots (R_2 \& S_i) \Rightarrow \bigvee_{j \in J} \exists Z_{ij}. S_{ij} \ldots (i \in I) \quad (R_1 \& R_2) \square S \Rightarrow \bigvee_{i \in I} \bigvee_{j \in J} \exists Z_i. Z_{ij}. S_{ij} \]
if \( Z_j \notin \text{var}((R_1 \& R_2) \square S), Z_{ij} \notin \text{var}((R_1 \& R_2) \square S) \cup Z_{ij} \) for all \( i, j \in I \).

TS Trivial Statement
\[ \phi : G \Rightarrow D \quad \text{if } \text{Sol}(G) \subseteq \text{Sol}_D(D). \]

DC DeComposition
\[ \bar{e}_m \rightarrow \bar{t}_m \square S \Rightarrow D \quad \text{if } h \bar{c}_m \text{ is not a pattern.} \]

IM IMitation
\[ \bar{e}_m \rightarrow X \square (S \& h \bar{X}_m \rightarrow X) \Rightarrow \bigvee_{i \in I} \exists Z_i. S_i \]

if \( h \bar{c}_m \) is not a pattern, \( X \in \mathcal{V} \), and \( \bar{X}_m \notin \text{var}(h \bar{c}_m \rightarrow X \square S) \).

(AR)_p Argument Reduction for Primitive Functions
\[ \bar{e}_n \rightarrow X_n \square (S \& p X_n \rightarrow t) \Rightarrow \bigvee_{i \in I} \exists Z_i. S_i \]
\[ p \bar{c}_n \rightarrow \bar{t} \quad t \square S \Rightarrow (S \& \bot \rightarrow t) \lor \bigvee_{i \in I} \exists X_n. Z_i. S_i \]
if \( p \in \text{PF}^n, \bar{X}_n \notin \text{var}(p \bar{c}_n \rightarrow \bar{t} \square S), \rightarrow \forall \rightarrow \text{(production)}, \rightarrow ! (\text{constraint}). \)

(AR)_f Argument Reduction for Defined Functions
\[ (\bar{e}_n \rightarrow X_n \& f X_n \rightarrow Y \& Y \bar{a}_k \rightarrow t) \square S \Rightarrow \bigvee_{i \in I} \exists Z_i. S_i \]
\[ f \bar{c}_n \bar{a}_k \rightarrow \bar{t} \quad \square S \Rightarrow \bigvee_{i \in I} \exists X_n, Y, Z_i. S_i \]
if \( f \in \text{DF}^n (k > 0) \), and \( \bar{X}_n, Y \notin \text{var}(f \bar{c}_n \bar{a}_k \rightarrow \bar{t} \square S) \).

(DF)_f Defined Function
\[ \ldots R_i \bar{X}_n \rightarrow \bar{t}_n, Y \rightarrow t \square S \Rightarrow D_i \ldots (i \in I) \]
\[ f \bar{t}_n \rightarrow t \square S \Rightarrow (S \& \bot \rightarrow t) \lor \bigvee_{i \in I} D_i \]
if \( f \in \text{DF}^n, \bar{X}_n, Y \notin \text{var}(f \bar{t}_n \rightarrow t \square S) \), and \( (f \bar{X}_n \rightarrow Y \Rightarrow \bigvee_{i \in I} R_i) \in \mathcal{P}^- \).

Fig. 5. The Constraint Negative Proof Calculus CNPC(D)
6. Declarative debugging of missing answers in CFLP(\(D\))

The declarative debugging of missing answers also assumes an intended interpretation \(I_p\) of the CFLP(\(D\))-program \(P\), 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(\(D\))-program \(P\):

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 \(\text{Sol}_{I_p}(G) \not\subseteq \bigcup_{i \in I} \text{Sol}_{D}(S_i)\), meaning that the aca \(G \Rightarrow \bigvee_{i \in I} S_i\) is not valid in the intended interpretation \(I_p\), 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 \((f)_{P} : (f X_n \rightarrow Y \Rightarrow D_f)\) for \(f\) in \(P^-\) is not valid in \(I_p\), which means \(\text{Sol}_{I_p}(f X_n \rightarrow Y) \not\subseteq \text{Sol}_{I_p}(D_f)\), showing that \(f\)'s definition as given in \(P\) is incomplete w.r.t. \(I_p\).

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 \(P^-\). The Constraint Negative Proof Calculus CNPC(\(D\)) consisting of the inference rules displayed in Fig. 5 has been designed with the aim of enabling logical proofs \(P^- \vdash_{\text{CNPC}(D)} G \Rightarrow D\) of acas. We use a special operator \& in order to express the result of attaching to a given goal \(G\) a solved goal \(S'\) resulting from a previous computation, so that computation can continue from the new goal \(G\) & \(S'\).

Formally, assuming \(G = \exists \bar{U}\). (\(R \square (\Pi \square \sigma)\)) and \(S' = \exists \bar{U}'. (\Pi' \square \sigma')\) a solved goal such that \(\bar{U} \setminus \text{dom}(\sigma') \subseteq \bar{U}', \sigma' = \sigma',\) and \(\text{Sol}_{D}(\Pi') \subseteq \text{Sol}_{D}(\Pi\sigma')\), the operation \(G \& S'\) is defined as \(\exists \bar{U}'. (R\sigma' \square (\Pi' \square \sigma'))\). The inference rule CJ infers an aca for a goal with composed kernel \((R_1 \sqcap R_2) \square S\) from acas for goals with kernels of the form \((R_1 \square S)\) and \((R_2 \& S_i)\), respectively; while other inferences deal with different kinds of atomic goal kernels.

Any CNPC(\(D\))-derivation \(P^- \vdash_{\text{CNPC}(D)} G \Rightarrow D\) can be depicted in the form of a Negative Proof Tree over \(D\) (shortly, NPT) with acas at its nodes, such that the aca at any node is inferred from the acas at its children using some CNPC(\(D\)) inference rule. We say that a goal solving system for CFLP(\(D\)) 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 \(P^- \vdash_{\text{CNPC}(D)} G \Rightarrow \bigvee_{i \in I} S_i\) 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 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(\(D\)) which formalizes the goal solving methods underlying actual CFLP systems such as Curry or TOY.

**Proof.** A more general result can be proved: If \((\bar{R} \sqcap \bar{R}') \& S \models P_{GSC(\(D\))} D\) (with a partially developed search space of finite size \(p\) built using the program \(P\), a Goal Solving Calculus
GSC(\(D\)) inspired in (López et al., 2004), and a certain selection strategy that only selects atoms descendants of the part \(R\) then \(\mathcal{P}^- \vdash_{\text{CNPC}(\mathcal{D})} R \& S \Rightarrow D\) 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(\(D\)) Calculus)** Let \(G \Rightarrow D\) be any aca for a given CFLP(\(D\))-program \(\mathcal{P}\). If \(\mathcal{P}^- \vdash_{\text{CNPC}(\mathcal{D})} G \Rightarrow D\) 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(\(D\))-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 \(\mathcal{P}^- \vdash_{\text{CNPC}(\mathcal{D})} G \Rightarrow D\). Then the aca \(G \Rightarrow D\) is a logical consequence of \(\mathcal{P}^-\) by Theorem 6. By Definition 2, we conclude that \(\text{Sol}_{\mathcal{I}_\mathcal{P}}(G) \subseteq \text{Sol}_{\mathcal{P}}(D)\) holds for any model \(\mathcal{I}\) of \(\mathcal{P}^-\). However, we also know that \(\text{Sol}_{\mathcal{I}_\mathcal{P}}(G) \nsubseteq \text{Sol}_{\mathcal{P}}(D)\), 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 \((\text{DF})_f\) is the only inference rule in the CNPC(\(D\)) 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(\(D\)) inference rules other than \((\text{DF})_f\). More precisely, given a NPT \(\mathcal{T}\) witnessing a CNPC(\(D\)) proof \(\mathcal{P}^- \vdash_{\text{CNPC}(\mathcal{D})} G \Rightarrow D\), its associated Abbreviated Negative Proof Tree (shortly, ANPT) \(\mathcal{A}\mathcal{T}\) 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.

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 acas, and a given node $N$ is buggy iff theaca at $N$ is invalid (i.e., it represents an incomplete recollection of computed answers in the intended interpretation $I_P$) while theaca at each children node $N_j$ is valid (i.e., it represents a complete recollection of computed answers in the intended interpretation $I_P$).

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 acas 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 acas for the functions $\text{gen}$, $\text{even}$, 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 guarantees that declarative diagnosis with ANPTs used as CTs leads to the correct detection of incomplete program functions.

**Theorem 8. (ANPTs Lead to the Diagnosis of Incomplete Functions)** As in Theorem 7, assume that an incompleteness symptom has been observed for a given CFLP(\( D \))-program \( P \) as explained in Definition 4, with intended interpretation \( I_P \), admissible initial goal \( G \), and finite disjunction of answers \( D = \bigvee_{i \in I} S_i \), computed by an admissible goal solving system. Then \( P \vdash_{\text{CNPC}(D)} G \Rightarrow D \), and the ANPT constructed from any NPT witnessing this derivation, has some buggy node. Moreover, each such buggy node points to an axiom \( (f) \) which is incomplete with respect to the user’s intended interpretation \( I_P \).

7. A declarative debugging tool of missing answers in \( \text{T}O\text{Y} \)

In this section, we discuss the implementation in the \( \text{T}O\text{Y} \) 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 \( D \).

We summarize first the normal process followed by the \( \text{T}O\text{Y} \) system when compiling a source program \( P . \text{toy} \) and solving an initial goal \( G \) with respect to \( P \). During the compilation process the system translates a source program \( P . \text{toy} \) into a Prolog program \( P . \text{pl} \) including a predicate for each function in \( P \). For instance the function \( \text{even} \) of our running example is transformed into a predicate

\[
\text{even}(N,R,IC,OC) : - \quad \text{... code for even ... .}
\]

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 \( P \) is also translated into a Prolog goal and solved with respect to \( P . \text{pl} \) 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 $P.pl$ is transformed into a new Prolog program $P^T.pl$. The debugger can safely assume that $P.pl$ already exists because the tool is always initiated after some missing answer has been detected by the user. The transformed program $P^T$ behaves almost identically to $P$, 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:

   ```prolog
   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 $P^T$. 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 acas 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/.

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 $f \Gamma_n \rightarrow t \square S \Rightarrow \bigvee_{i \in I} S_i$, 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 $\text{gen } 2 \ 1 \rightarrow A = \{A = 2 : 1 : _\}$ 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, $1 / / _ \rightarrow A = \{A = 1\}$ is the aca $1 / / \bot \rightarrow A = \{A \mapsto 1\}$ 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 $1 / / \bot \rightarrow A = \{A \mapsto 1\}$ is valid with respect to the intended interpretation $I_{P_{ED}}$ of $P_{ED}$, while the statement $\forall X. (1 / / X \rightarrow A = \{A \mapsto 1\})$ has a different meaning and is not valid in $I_{P_{ED}}$.

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 $f \text{Diff} 1:2:2:1:_\rightarrow A = \{A = 1\}$. According to her knowledge of $I_{P_{ED}}$ 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 $f \text{Diff} 2:2:1:_\rightarrow A = \{\}$ (which in fact represents $f \text{Diff} 2:2:1:\bot \rightarrow A = \{A \mapsto \bot\}$, as explained before). Again, her knowledge of $I_{P_{ED}}$ leads the user to expect that $f \text{Diff} 2:2:1:\bot$ can return some defined result, and the aca is marked as invalid. After this question the debugger points out at $f \text{Diff}$ 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 $P^T.pl$ from $P.pl$ 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 $P^T.pl$ requires almost the same system resources as for $P.pl$ because writing the trace causes no significant overhead in our experiments.

3. Producing the CT from the trace is not straightforward and requires several traverses of the trace. Although more time-consuming due to the algorithmic difficulty, this process only keeps portions of the trace in memory at each moment.

4. The most inefficient phase in our current implementation is the graphical interface. Although it would be possible to keep in memory only the portion of the tree displayed at each moment, our graphical interface loads the whole CT in main memory. We plan to improve this limitation in the future. However the current prototype can cope with CTs containing thousands of nodes, which is enough for medium size computations.

5. As usual in declarative debugging, the efficiency of the tool depends on the computation tree size, which in turn usually depends on the size of the data structures required and not on the program size.

A different issue is the difficulty of answering the questions by the user. Indeed in complicated programs involving constraints the acas 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 CTs should be worked out in future improvements of the prototype. For instance, asking the user for a concrete missing instance of the initial goal and starting a diagnosis session for the instantiated goal might be helpful.

8. Conclusions and future work

We have presented a logical and semantic framework for the declarative diagnosis of wrong and missing computed answers in $CFLP(D)$, a generic scheme for Constraint Functional-Logic Programming over a given constraint domain $D$ 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 (acas). As in declarative diagnosis for FP languages, the values displayed at acas 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 $CFLP$. 

www.intechopen.com
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, acas 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 acas, 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 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 DDT, which implements the proposed technique for wrong answers over the domain R of arithmetic constraints over the real numbers has been implemented as a non-trivial extension of previously existing debugging tools. 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 TOY.

As future work, we plan several improvements of 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 CTs and the acas 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

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


The current book is a nice blend of number of great ideas, theories, mathematical models, and practical systems in the domain of Semantics. The book has been divided into two volumes. The current one is the first volume which highlights the advances in theories and mathematical models in the domain of Semantics. This volume has been divided into four sections and ten chapters. The sections include: 1) Background, 2) Queries, Predicates, and Semantic Cache, 3) Algorithms and Logic Programming, and 4) Semantic Web and Interfaces.

Authors across the World have contributed to debate on state-of-the-art systems, theories, mathematical models in the domain of Semantics. Subsequently, new theories, mathematical models, and systems have been proposed, developed, and evaluated.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following: