



ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

**Electronic Notes in
Theoretical Computer
Science**

Electronic Notes in Theoretical Computer Science 256 (2009) 119–135

www.elsevier.com/locate/entcs

A Logical Framework for Debugging in Declarative Constraint Programming

Rafael del Vado Vírveda^{1,2}*Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, Spain*

Abstract

We present a logical and semantic framework for diagnosing wrong computed answers in $CFLP(\mathcal{D})$, a newly proposed generic scheme for lazy Constraint Functional Logic Programming which can be instantiated by any constraint domain \mathcal{D} given as parameter, and supports a powerful combination of functional and constraint logic programming over \mathcal{D} . Our approach extends and combines declarative debugging techniques previously developed for less expressive programming paradigms, namely the $CLP(\mathcal{D})$ scheme and lazy functional logic languages. Debugging starts with the observation of a *wrong computed answer* which the user regards as incorrect w.r.t. an *intended model* that provides a declarative description of the program's semantics. Debugging proceeds by exploring an abridged *proof tree* that provides a purely declarative view of the computation, so that the user does not need to understand the complex underlying operational mechanisms. Debugging ends with the detection of a function rule in the program that is incorrect w.r.t. the intended model. We prove the logical correctness of the debugging method for any sound $CFLP(\mathcal{D})$ -system whose computed answers are logical consequences of the program, and we describe a practical tool which implements the debugging method for the domain of arithmetic constraints over the real numbers.

Keywords: Logical Frameworks, Declarative Programming, Algorithmic Debugging, Constraints.

1 Introduction

Debugging tools are a practical need for diagnosing the causes of erroneous computations. 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, *declarative diagnosis* 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 an *error symptom* by the user.

¹ The author has been partially supported by the Spanish National Projects FAST-STAMP (TIN2008-06622-C03-01), MERIT-FORMS (TIN2005-09027-C03-03), PROMESAS-CAM (S-0505/TIC/0407), and UCM-BSCH-GR58/08-910502 (GPD-UCM).

² Email: rdelvado@sip.ucm.es

Each node in a *CT* represents the computation of some observable result, depending on the results of its children nodes. Declarative diagnosis explores a *CT* looking for a so-called *buggy node* which computes an incorrect result from children whose results are correct; such a node must point to an incorrect program fragment. 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 [16]. Declarative diagnosis was first proposed in the field of logic programming [20,9], and it has been successfully extended to other declarative programming paradigms, including lazy functional programming [17,19], constraint logic programming [21,10] and functional logic programming [5,6]. In contrast to recent approaches to error diagnosis using *abstract interpretation* (as e.g. [7,12,1] and some of the approaches described in [8]), 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 [2,6], safe inference of information from answers previously given by the user [5], or *CTs* tailored to the needs of a particular debugging problem over a particular computation domain [10]. Current research in declarative diagnosis has still to face many challenges regarding both the foundations and the development of practical tools.

The aim of this paper is to present a logical and semantic framework for diagnosing wrong computed answers in $CFLP(\mathcal{D})$, a newly proposed generic programming scheme 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} [14]. 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 *CTs*. Our aim is to achieve a natural combination of previous approaches that were separately developed for the $CLP(\mathcal{D})$ scheme [21] and for lazy functional logic languages [5]. 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 *DDT*, developed as an extension of previously existing but less powerful tools [3,6]. *DDT* implements the proposed diagnosis method for $CFLP(\mathcal{R})$ -programming in the *TCOY* system [15] using the domain \mathcal{R} of arithmetic constraints over the real numbers.

The rest of the paper is organized as follows: Section 2 motivates our approach by presenting a debugging example which is used as illustration along the rest of the paper. Section 3 recalls the $CFLP(\mathcal{D})$ scheme from [14] to the extent needed for understanding the theoretical results in this paper. 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 *DDT*. Section

6 concludes and points to some plans for future work.

2 A Motivating Example

As a motivation for the rest of the paper, we consider the following program fragment written in \mathcal{TOY} [15], a programming system which supports several instances of the $CFLP(\mathcal{D})$ scheme:

Example 2.1 (Building Ladders 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

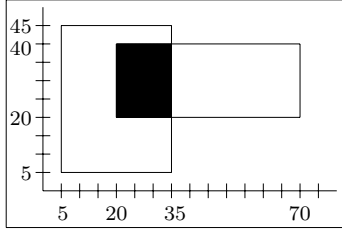
rect :: point -> real -> real -> figure
rect (X,Y) LX LY (X',Y') = (X' >= X) && (X' <= X+LX) &&
                                (Y' <= Y) && (Y' <= Y+LY)

% 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]

```



Here, \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 number 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 $(\text{rect } (20,20) \ 50 \ 20)$ and $(\text{rect } (5,5) \ 30 \ 40)$, computed indirectly as the first steps of two particular ladders. The diagram included in Example 2.1 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 $R \rightarrow \text{true}$ and three constraints imposed on the variables X and Y ³. The only constraint imposed on Y (namely $Y \leq 5$) 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 w.r.t. the intended meaning of the program. As we will see in Sections 4 and 5, the declarative debugging technique presented in this paper leads to the diagnosis of the program rule for the function `rect` as responsible for the wrong answer. Indeed, this program rule is incorrect w.r.t. the intended program semantics; as shown in Example 2.1, the third inequality at the right hand side should be $Y' \geq Y$ instead of $Y' \leq Y$.

The traditional approach to declarative debugging in the $CLP(\mathcal{D})$ scheme includes the diagnosis of both *wrong* and *missing* computed answers [21]. However, the declarative diagnosis of missing answers falls outside the scope of this paper.

3 The $CFLP(\mathcal{D})$ Programming Scheme

In this section we recall the essentials of the $CFLP(\mathcal{D})$ scheme [14] 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 paper.

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 Σ_{\perp} for the

³ There are other five computed answers consisting of the substitution $R \rightarrow \text{false}$ and various constraints imposed on X and Y .

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, \dots and a set \mathcal{U} of *primitive elements* u, v, \dots (as e.g. the set \mathbb{R} of the real numbers) mutually disjoint and disjoint from Σ_\perp . *Expressions* $e \in Exp_\perp(\mathcal{U})$ have the following syntax:

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

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:

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

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 2.1 is a functional data value represented as pattern.⁴

As usual, we define *substitutions* $\sigma \in Sub_\perp(\mathcal{U})$ as mappings $\sigma : \mathcal{V} \rightarrow Pat_\perp(\mathcal{U})$ extended to $\sigma : Exp_\perp(\mathcal{U}) \rightarrow Exp_\perp(\mathcal{U})$ 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 $p^\mathcal{D} \subseteq D_\mathcal{U}^n \times D_\mathcal{U}$ of each $p \in PF^n$ satisfying the technical *monotonicity*, *antimonotonicity*, and *radicality* requirements given in [14]. We use the notation $p^\mathcal{D} \bar{t}_n \rightarrow t$ to indicate that $(\bar{t}_n, t) \in p^\mathcal{D}$.

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 $p \bar{e}_n \rightarrow !t$, meaning that the primitive function $p \in PF^n$ with parameters $\bar{e}_n \in Exp_\perp(\mathcal{U})$ returns a *total* result $t \in Pat_\perp(\mathcal{U})$ (i.e., with no occurrences of \perp). Constraints whose atomic parts have the 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} .

⁴ Note that (5, 5) can be seen as syntactic sugar for (*pair* 5 5), *pair* being a constructor for ordered pairs.

Example 3.1 (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:

- (i) $eq_{\mathbb{R}}$, 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.
- (ii) 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 w.r.t. the *information ordering* introduced in [14] and defined in the Appendix; $seq^{\mathcal{R}} t s \rightarrow \perp$ otherwise. In the sequel, $e_1 == e_2$ abbreviates $seq e_1 e_2 \rightarrow! true$.
- (iii) $+$, $-$, $*$, 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 $*^{\mathcal{R}}$.
- (iv) $<$, \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 into 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 [14] 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: ⁵

- (i) 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.
- (ii) $\Delta \subseteq DCon_{\perp}(\mathcal{D})$ 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 TOY program presented in Section 2 can be viewed as an example of

⁵ In practice, 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 program semantics.

$CFLP(\mathcal{R})$ -program written in \mathcal{TOY} 's syntax. The reader is referred to [14] for more explanations and examples in other constraint domains.

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 $G : \exists \bar{U}. (P \sqcap \Delta)$, where \bar{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 $e \rightarrow s$ fulfilling the *admissibility conditions* given in [14], 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 \bar{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 $S : \exists \bar{U}. (\sigma \sqcap \Pi)$, where σ is a finite set of productions $X \rightarrow t$ or $s \rightarrow Y$ interpreted as the variable bindings of an idempotent substitution and $\Pi \subseteq PCon_{\perp}(\mathcal{D})$ 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 2.1 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 $R \rightarrow true \sqcap X \leq 35 \wedge X \geq 20 \wedge Y \leq 5$.

Goal solving systems can be implementations of $CFLP$ languages such as *Curry* [11] or \mathcal{TOY} [15], or formal *goal solving calculi* including recent proposals such as the $CDNC(\mathcal{D})$ calculus [22], which is sound and complete w.r.t. 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 Declarative semantics

In this subsection we recall some notions and results on the declarative semantics of $CFLP(\mathcal{D})$ -programs which were developed in [14] and are needed for the rest of this paper. 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:

- (i) *c-productions* $e \rightarrow t \Leftarrow \Pi$, 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 $e \rightarrow t$). As a particular kind of *c-productions* useful for debugging we distinguish *c-facts* $f \bar{t}_n \rightarrow t \Leftarrow \Pi$ with $f \in DF^n$. A *c-production* is called *trivial* iff $t = \perp$ or $Sol_{\mathcal{D}}(\Pi) = \emptyset$.
- (ii) *c-atoms* $p \bar{e}_n \rightarrow !t \Leftarrow \Pi$, with $p \in PF^n$ and t total (if Π is empty they boil down to unconstrained atoms written as $p \bar{e}_n \rightarrow !t$). 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 [5] to arbitrary constraint domains. We write $\mathcal{I} \models_{\mathcal{D}} \varphi$ 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 [14]. Now we are in a position to define various semantics notions which rely on a given c-interpretation \mathcal{I} over \mathcal{D} .

Definition 3.2 (Interpretation-dependent Semantic Notions)

- (i) 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) $Sol_{\mathcal{I}}(\delta) = \{\eta \in Val_{\perp}(\mathcal{D}) \mid \mathcal{I} \models_{\mathcal{D}} \delta\eta\}$, 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 $Sol_{\mathcal{I}}(\Delta) = \bigcap_{\delta \in \Delta} Sol_{\mathcal{I}}(\delta)$.
- (ii) The set of solutions of a production $e \rightarrow t$ is a subset $Sol_{\mathcal{I}}(e \rightarrow t) \subseteq Val_{\perp}(\mathcal{D})$ defined as $Sol_{\mathcal{I}}(e \rightarrow t) = \{\eta \in Val_{\perp}(\mathcal{D}) \mid \mathcal{I} \models_{\mathcal{D}} e\eta \rightarrow t\eta\}$. The set of solutions of a set of productions P is defined as $Sol_{\mathcal{I}}(P) = \bigcap_{(e \rightarrow t) \in P} Sol_{\mathcal{I}}(e \rightarrow t)$.
- (iii) The set of solutions of an admissible goal $G : \exists \bar{U}. (P \sqcap \Delta)$ is a subset $Sol_{\mathcal{I}}(G) \subseteq Val_{\perp}(\mathcal{D})$ defined as follows: $Sol_{\mathcal{I}}(G) = \{\eta \in Val_{\perp}(\mathcal{D}) \mid \eta' \in Sol_{\mathcal{I}}(P) \cap Sol_{\mathcal{I}}(\Delta) \text{ for some } \eta' \text{ such that } \eta'(X) = \eta(X) \text{ for all } X \notin \bar{U}\}$.

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 3.3 (Model-theoretic Semantics) Let \mathcal{P} a $CFLP(\mathcal{D})$ -program and \mathcal{I} a c-interpretation.

- (i) \mathcal{I} is a *model* of \mathcal{P} (in symbols, $\mathcal{I} \models_{\mathcal{D}} \mathcal{P}$) iff every constrained program rule $(f\bar{t}_n \rightarrow r \Leftarrow \Delta) \in \mathcal{P}$ 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 $(f\bar{t}_n \rightarrow r \Leftarrow \Delta)\eta$ is ground, $\mathcal{I} \models_{\mathcal{D}} \Delta\eta$ and $\mathcal{I} \models_{\mathcal{D}} r\eta \rightarrow t$ one has $\mathcal{I} \models_{\mathcal{D}} (f\bar{t}_n)\eta \rightarrow t$ (or equivalently, $((f\bar{t}_n)\eta \rightarrow t) \in \mathcal{I}$).
- (ii) A solved form S is a *semantically valid answer* for a goal G w.r.t. 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 $\mathcal{I} \models_{\mathcal{D}} \mathcal{P}$.

4 Declarative Diagnosis of Wrong Answers in $CFLP(\mathcal{D})$

In this section, we present the logical and semantic framework of the declarative diagnosis method 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, $\text{rect}(X, Y) \text{ LX LY } (A, B) \rightarrow \text{false} \Leftarrow A < X \wedge \text{LX} > 0 \wedge \text{LY} > 0$ could belong to the intended model \mathcal{I} for the program fragment shown in Example 2.1. As explained in Subsection 3.4, 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 3.3 above) corresponds to the so-called *weak semantics* from [14], 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 paper, as long as the ground c-facts valid in them are the same.

The aim of declarative diagnosis 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 4.1 (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:

- (i) S is a *wrong answer* w.r.t \mathcal{I} (serving as *symptom*) iff $\text{Sol}_{\mathcal{D}}(S) \not\subseteq \text{Sol}_{\mathcal{I}}(G)$.
- (ii) \mathcal{P} is *incorrect* w.r.t. \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 3.3).

For instance, the computed answer shown in Example 2.1 is wrong w.r.t. the intended model for the program assumed in that example, for the reasons already discussed in Section 2. As illustrated by this 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 program \mathcal{P} , declarative diagnosis 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. 1. We will say that the $CPPC(\mathcal{D})$ -proof *witnesses* the computed answer.

Most of these 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 [14]. The main novelties in $CPPC(\mathcal{D})$ are the addition of rule **EX** (to deal with the existential quantifiers in computed answers) and a reformulation of rule **DF_P**, which is presented as the consecutive application of two inference steps named **AR_f** and **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 **FA_f**, called *boxed c-facts* in the sequel. As we will see, only boxed c-facts will appear

EX Existential	$\frac{G\sigma \Leftarrow \Pi}{G \Leftarrow \exists \bar{U}. (\sigma \sqcap \Pi)}$	if $fvar(G) \cap \bar{U} = \emptyset$.
TI Trivial Inference	$\frac{}{\varphi}$	if φ is a trivial c-statement.
RR Restricted Reflexivity	$\frac{}{t \rightarrow t \Leftarrow \Pi}$	if $t \in \mathcal{U} \cup \mathcal{V}$.
SP Simple Production	$\frac{}{s \rightarrow t \Leftarrow \Pi}$	if $s \in Pat_{\perp}(\mathcal{U})$, $s \in \mathcal{V}$ or $t \in \mathcal{V}$, and $Sol_{\mathcal{D}}(\Pi) \subseteq Sol_{\mathcal{D}}(s \rightarrow t)$.
DC Decomposition	$\frac{e_1 \rightarrow t_1 \Leftarrow \Pi \dots e_m \rightarrow t_m \Leftarrow \Pi}{h\bar{e}_m \rightarrow h\bar{t}_m \Leftarrow \Pi}$	
IR Inner Reduction	$\frac{e_1 \rightarrow t_1 \Leftarrow \Pi \dots e_m \rightarrow t_m \Leftarrow \Pi}{h\bar{e}_m \rightarrow X \Leftarrow \Pi}$	if $h\bar{e}_m \notin Pat_{\perp}(\mathcal{U})$, $X \in \mathcal{V}$ and $Sol_{\mathcal{D}}(\Pi) \subseteq Sol_{\mathcal{D}}(h\bar{t}_m \rightarrow X)$
PF Primitive Function	$\frac{e_1 \rightarrow t_1 \Leftarrow \Pi \dots e_n \rightarrow t_n \Leftarrow \Pi}{p\bar{e}_n \rightarrow t \Leftarrow \Pi}$	if $p \in PF^n$, $t_i \in Pat_{\perp}(\mathcal{U})$ for each $1 \leq i \leq n$, and $Sol_{\mathcal{D}}(\Pi) \subseteq Sol_{\mathcal{D}}(p\bar{t}_n \rightarrow t)$.
DF_P P-Defined Function	$\frac{e_1 \rightarrow t_1 \Leftarrow \Pi \dots e_n \rightarrow t_n \Leftarrow \Pi \quad \frac{\Delta \Leftarrow \Pi \quad r \rightarrow t \Leftarrow \Pi \quad (\mathbf{FA}_f)}{f\bar{t}_n \rightarrow t \Leftarrow \Pi}}{f\bar{e}_n \rightarrow t \Leftarrow \Pi} \quad (\mathbf{AR}_f)$ $\frac{e_1 \rightarrow t_1 \Leftarrow \Pi \dots e_n \rightarrow t_n \Leftarrow \Pi \quad \frac{\Delta \Leftarrow \Pi \quad r \rightarrow s \Leftarrow \Pi \quad (\mathbf{FA}_f)}{f\bar{t}_n \rightarrow s \Leftarrow \Pi}}{f\bar{e}_n \bar{a}_k \rightarrow t \Leftarrow \Pi} \quad (\mathbf{AR}_f)$	
	<p>if $f \in DF^n$ ($k > 0$), $(f\bar{t}_n \rightarrow r \Leftarrow \Delta) \in [\mathcal{P}]_{\perp} \equiv \{R\theta \mid R \in \mathcal{P}, \theta \in Sub_{\perp}(\mathcal{U})\}$ and $s \in Pat_{\perp}(\mathcal{U})$.</p>	
AC Atomic Constraint	$\frac{e_1 \rightarrow t_1 \Leftarrow \Pi \dots e_n \rightarrow t_n \Leftarrow \Pi}{p\bar{e}_n \rightarrow !t \Leftarrow \Pi}$	if $p \in PF^n$, $t_i \in Pat_{\perp}(\mathcal{U})$ for each $1 \leq i \leq n$, and $Sol_{\mathcal{D}}(\Pi) \subseteq Sol_{\mathcal{D}}(p\bar{t}_n \rightarrow !t)$.

Fig. 1. The Constraint Positive Proof Calculus $CPPC(\mathcal{D})$

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 **DF_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 $Sol_{\mathcal{D}}(\Pi) \subseteq Sol_{\mathcal{D}}(p\bar{t}_n \rightarrow t)$ which is true iff $p^{\mathcal{D}}\bar{t}_n\eta \rightarrow t\eta$ holds for all $\eta \in Sol_{\mathcal{D}}(\Pi)$. Some other inference rules in Fig. 1 have similar conditions.

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 nowhere else in the proof tree. Fig. 2. shows a *PPT*(\mathcal{R}) representing a *CPPC*(\mathcal{R})-derivation which witnesses the computed answer from Example 2.1, which is wrong w.r.t. 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 4.2 (Existence of *CPPC*(\mathcal{D})-sound goal solving systems) *The goal solving calculus $CDNC(\mathcal{D})$ given in [22] is *CPPC*(\mathcal{D})-sound.*

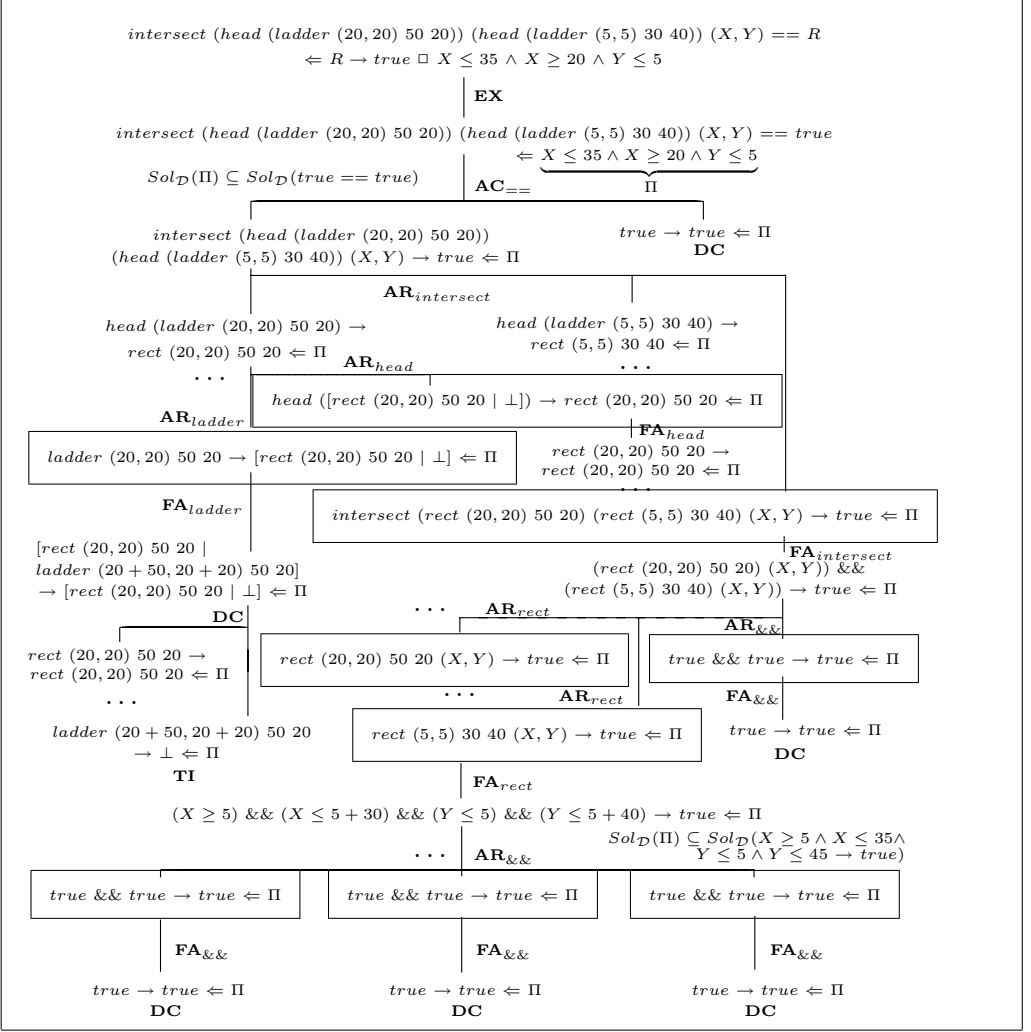
Proof. Straightforward adaptation of the soundness theorem for *CDNC*(\mathcal{D}) presented in [22]. \square

In addition to *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* [11] and *TOY* [15] 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 3.3:

Theorem 4.3 (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**, **AR_f**, and **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 [14].

- 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 $\mathcal{I} \models_{\mathcal{D}} G \Leftarrow \exists \bar{U}. (\sigma \sqcap \Pi)$, i.e., $Sol_{\mathcal{D}}(\exists \bar{U}. (\sigma \sqcap \Pi)) \subseteq Sol_{\mathcal{I}}(G)$. Let $\eta \in Sol_{\mathcal{D}}(\exists \bar{U}. (\sigma \sqcap \Pi))$. By the syntactic form of solved goals, $\eta \in Sol_{\mathcal{D}}(\exists \bar{U}. (\bar{X}_n \rightarrow t_n \wedge \bar{s}_m \rightarrow \bar{Y}_m \sqcap \Pi))$ and $\eta \in Sol_{\mathcal{D}}(\exists \bar{U}. (\bar{X}_n = t_n \wedge \bar{Y}_m = \bar{s}_m \sqcap \Pi))$. By applying Definition 3.2, there

Fig. 2. A Positive Proof Tree in $CPPC(\mathcal{R})$

exists $\eta' \in \text{Val}_\perp(\mathcal{D})$ such that $\eta' =_{\setminus \bar{U}} \eta \text{ y } \eta' \in \text{Sol}_{\mathcal{D}}(\overline{X_n = t_n} \wedge \overline{Y_m = s_m} \sqcap \Pi)$, and therefore, $\eta' \in \text{Sol}_{\mathcal{D}}(\overline{X_n = t_n} \wedge \overline{Y_m = s_m})$ (i.e., $\eta' \in \text{Sol}_{\mathcal{D}}(\sigma)$) and $\eta' \in \text{Sol}_{\mathcal{D}}(\Pi)$. Since by *induction hypothesis* $\text{Sol}_{\mathcal{D}}(\Pi) \subseteq \text{Sol}_{\mathcal{I}}(G\sigma)$, it follows that $\eta' \in \text{Sol}_{\mathcal{I}}(G\sigma)$. Moreover, since $\eta' \in \text{Sol}_{\mathcal{D}}(\sigma)$, we obtain $\eta' \in \text{Sol}_{\mathcal{I}}(G)$. In consequence, there exists $\eta' \in \text{Val}_\perp(\mathcal{D})$ such that $\eta' =_{\setminus \bar{U}} \eta$ and $\eta' \in \text{Sol}_{\mathcal{I}}(G)$. Finally, using the condition of applicability $\text{fvar}(G) \cap \bar{U} = \emptyset$ associated to the rule **EX**, we can conclude that $\eta \in \text{Sol}_{\mathcal{I}}(G)$.

- The rule **AR_f** is semantically correct. Let \mathcal{I} be an arbitrary model of \mathcal{P} such that $\mathcal{I} \models_{\mathcal{D}} e_i \rightarrow t_i \Leftarrow \Pi$ for each $1 \leq i \leq n$ (i.e., $\text{Sol}_{\mathcal{D}}(\Pi) \subseteq \text{Sol}_{\mathcal{I}}(e_i \rightarrow t_i)$ for each $1 \leq i \leq n$), $\mathcal{I} \models_{\mathcal{D}} f\bar{t}_n \rightarrow s \Leftarrow \Pi$ (i.e., $\text{Sol}_{\mathcal{D}}(\Pi) \subseteq \text{Sol}_{\mathcal{D}}(f\bar{t}_n \rightarrow s)$) and $\mathcal{I} \models_{\mathcal{D}} s\bar{a}_k \rightarrow s \Leftarrow \Pi$ (i.e., $\text{Sol}_{\mathcal{D}}(\Pi) \subseteq \text{Sol}_{\mathcal{I}}(s\bar{a}_k \rightarrow t)$). We prove that $\mathcal{I} \models_{\mathcal{D}} f\bar{e}_n\bar{a}_k \rightarrow t \Leftarrow \Pi$, i.e., $\text{Sol}_{\mathcal{D}}(\Pi) \subseteq \text{Sol}_{\mathcal{I}}(f\bar{e}_n\bar{a}_k \rightarrow t)$. Let $\eta \in \text{Sol}_{\mathcal{D}}(\Pi)$. We have then $\eta \in \text{Sol}_{\mathcal{I}}(e_i \rightarrow t_i)$ for each $1 \leq i \leq n$, and by Definition 3.2, $\mathcal{I} \Vdash_{\mathcal{D}} e_i\eta \rightarrow t_i\eta$

for each $1 \leq i \leq n$. Analogously, $\eta \in \text{Sol}_{\mathcal{I}}(f\bar{t}_n \rightarrow s)$, by Definition 3.2, $\mathcal{I} \Vdash_{\mathcal{D}} f\bar{t}_n\eta \rightarrow s\eta$, and by the *Conservation Property* (see [14] for details), $(f\bar{t}_n\eta \rightarrow s\eta) \in \mathcal{I}$. Analogously, $\eta \in \text{Sol}_{\mathcal{I}}(s\bar{a}_k \rightarrow t)$ and by Definition 3.2, $\mathcal{I} \Vdash_{\mathcal{D}} (s\eta)(\bar{a}_k\eta) \rightarrow t\eta$. But then, by applying of the rule **DF** _{\mathcal{I}} (see [14] for details), we have that $\mathcal{I} \Vdash_{\mathcal{D}} f(\bar{e}_n\eta)(\bar{a}_k\eta) \rightarrow t\eta$. From Definition 3.2, we obtain finally $\eta \in \text{Sol}_{\mathcal{I}}(f\bar{e}_n\bar{a}_k \rightarrow t)$.

- The rule **FA** _{f} is semantically correct. By definition of $[\mathcal{P}]_{\perp}$, there are $(f\bar{t}'_n \rightarrow r' \Leftarrow \Delta') \in \mathcal{P}$ and $\theta \in \text{Sub}_{\perp}(\mathcal{U})$ such that $(f\bar{t}'_n \rightarrow r' \Leftarrow \Delta')\theta \equiv (f\bar{t}_n \rightarrow r \Leftarrow \Delta)$. Let \mathcal{I} be an arbitrary model of \mathcal{P} such that $\mathcal{I} \models_{\mathcal{D}} \Delta \Leftarrow \Pi$ (i.e., $\text{Sol}_{\mathcal{D}}(\Pi) \subseteq \text{Sol}_{\mathcal{I}}(\Delta)$) and $\mathcal{I} \models_{\mathcal{D}} r \rightarrow s \Leftarrow \Pi$ (i.e., $\text{Sol}_{\mathcal{D}}(\Pi) \subseteq \text{Sol}_{\mathcal{I}}(r \rightarrow s)$). We prove that $\mathcal{I} \models_{\mathcal{D}} f\bar{t}_n \rightarrow s \Leftarrow \Pi$, i.e., $\text{Sol}_{\mathcal{D}}(\Pi) \subseteq \text{Sol}_{\mathcal{I}}(f\bar{t}_n \rightarrow s)$. Let $\eta \in \text{Sol}_{\mathcal{D}}(\Pi)$. Then we have $\eta \in \text{Sol}_{\mathcal{I}}(\Delta)$, and by Definition 3.2, $\mathcal{I} \Vdash_{\mathcal{D}} \Delta\eta$, and also, $\mathcal{I} \Vdash_{\mathcal{D}} \Delta'\theta\eta$. Analogously, $\eta \in \text{Sol}_{\mathcal{I}}(r \rightarrow s)$, and by Definition 3.2, $\mathcal{I} \Vdash_{\mathcal{D}} r\eta \rightarrow s\eta$, and also, $\mathcal{I} \Vdash_{\mathcal{D}} r'\theta\eta \rightarrow s\eta$. We have then $(f\bar{t}'_n \rightarrow r' \Leftarrow \Delta') \in \mathcal{P}$, $\theta\eta \in \text{Sub}_{\perp}(\mathcal{U})$ ground substitution and $s\eta \in \text{Pat}_{\perp}(\mathcal{U})$ ground such that $(f\bar{t}'_n \rightarrow r' \Leftarrow \Delta')\theta\eta \equiv (f\bar{t}_n \rightarrow r \Leftarrow \Delta)\eta$ is ground, $\mathcal{I} \Vdash_{\mathcal{D}} \Delta'\theta\eta$ and $\mathcal{I} \Vdash_{\mathcal{D}} r'\theta\eta \rightarrow s\eta$. Since \mathcal{I} is a model of \mathcal{P} , by applying Definition 3.3, we obtain $((f\bar{t}'_n)\theta\eta \rightarrow s\eta) \in \mathcal{I}$, i.e., $((f\bar{t}_n)\eta \rightarrow s\eta) \in \mathcal{I}$, or also, $(f\bar{t}_n \rightarrow s)\eta \in \mathcal{I}$. Finally, by applying the *Conservation Property* (see [14] for details), it is equivalent to $\mathcal{I} \Vdash_{\mathcal{D}} (f\bar{t}_n \rightarrow s)\eta$, and by Definition 3.2, we can conclude that $\eta \in \text{Sol}_{\mathcal{I}}(f\bar{t}_n \rightarrow s)$.

□

4.3 Declarative diagnosis using proof trees

Now we are ready to present a declarative diagnosis method 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 4.4 (Wrong answers are caused by erroneous program rules)

Assume that a *CPPC*(\mathcal{D})-sound goal solving system computes S as answer for the initial goal G using program \mathcal{P} . If S is wrong w.r.t. the user's intended interpretation \mathcal{I} then some program rule belonging to \mathcal{P} is incorrect w.r.t. \mathcal{I} .

Proof. Because of *CPPC*(\mathcal{D})-soundness of the goal solving system, we know that $\mathcal{P} \vdash_{\text{CPPC}(\mathcal{D})} G \Leftarrow S$. Then, from Theorem 4.3 we obtain $\mathcal{P} \models_{\mathcal{D}} G \Leftarrow S$, i.e., $\text{Sol}_{\mathcal{D}}(S) \subseteq \text{Sol}_{\mathcal{J}}(G)$ for each model $\mathcal{J} \models_{\mathcal{D}} \mathcal{P}$. Since S is wrong w.r.t. 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 4.1. Therefore, we can conclude that the intended model \mathcal{I} is not a model of \mathcal{P} . Then, by Definition 3.3, 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 **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}_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 CT s leads to the correct detection of program errors. A proof is given in the Appendix.

Theorem 4.5 (Declarative diagnosis of wrong answers) *Under the assumptions of Theorem 4.4, 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 Practical Debugging Tool for $CFLP(\mathcal{R})$

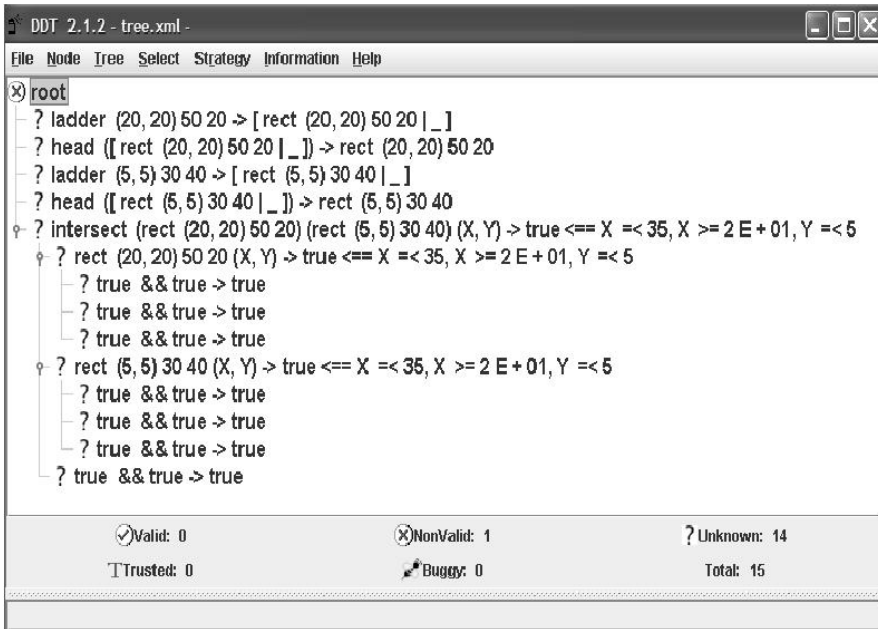
Fig. 3 shows the $APPT(\mathcal{R})$ associated to the $PPT(\mathcal{R})$ of Fig. 2. as displayed by DDT , the debugger tool included in the system \mathcal{TOY} . Although in theory all the c-facts in a $PPT(\mathcal{R})$ should include the same constraint Π , in practice the tool simplifies Π at each c-fact $f\bar{t}_n \rightarrow t \Leftarrow \Pi$, keeping only those atomic constraints related to the variables occurring on $f\bar{t}_n \rightarrow t$. It can be checked that such a simplification does not affect the intended meaning of c-facts.

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.

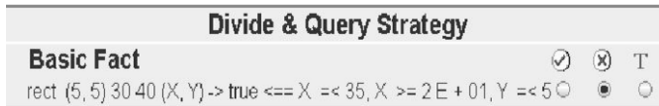
Next, the user can start a debugging session by selecting one of the two possible strategies included in DDT : the *top-down* or the *divide and query* strategy (see [6] for a comparative between both strategies in an older version of DDT which did not yet support constraints). After selecting the *divide and query* strategy, which usually leads to shorter sessions, DDT asks about the validity of the following node:

Divide & Query Strategy			
Basic Fact	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	T
intersect (rect (20, 20) 50 20) (rect (5, 5) 30 40) (X, Y) -> true <== X <= 35, X >= 20, Y <= 5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

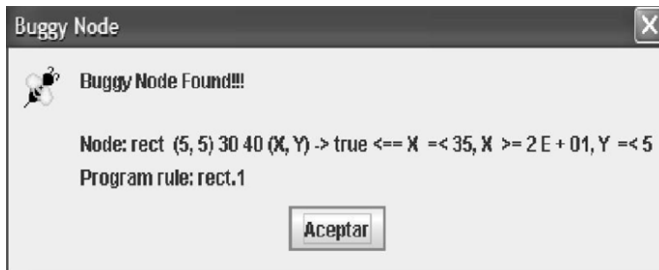
The intended program model corresponds to the intuitions explained in Section 2. Therefore, the question must be understood as: *Is (X, Y) a point in the intersection of the two rectangles for all possible values of X, Y satisfying $X \leq 35, X \geq 20, Y \leq 5$*

Fig. 3. The $APPT(\mathcal{R})$ corresponding to the $PPT(\mathcal{R})$ of Fig. 2.

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:



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:



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 3.1 and [14] for details. The tool is as an extension of older versions which did not yet support constraints over the domain \mathcal{R} [6,3], and it is part of the public distribution of the functional logic programming system *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 *DDT* can be found in [6,3].

6 Conclusions and Future Work

We have presented a logical and semantic framework for the declarative diagnosis of wrong computed answers in $CFLP(\mathcal{D})$, a generic scheme for constraint functional logic programming over a given constraint domain \mathcal{D} . The diagnosis technique 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.

A debugging tool called *DDT* which implements the proposed technique 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. *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.

As future work, we plan several improvements of *DDT*, such as enabling the diagnosis of *missing answers*, supporting finite domain constraints, and providing new facilities for simplifying the presentation of queries to the user.

Acknowledgement

The author is thankful to Mario Rodríguez Artalejo and Rafael Caballero for their collaboration, comments and contributions during the first stages of the development of this work and for the help in preparing the final version of this paper.

References

- [1] M. Alpuente, D. Ballis, F.J. Correa, and M. Falaschi. Correction of Functional Logic Programs. *Proc. ESOP'03*, Springer LNCS, 2003.
- [2] J. Boye, W. Drabent, and J. Małuszyński. Declarative Diagnosis of Constraint Programs: an Assertion-based Approach. *DiSciPl Deliverable D.WP2.2.M1.1-2*, 1997.
- [3] R. Caballero. *A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs*. *Proc. WCFLP'05*, ACM SIGPLAN, pp. 8–13, 2005.
- [4] R. Caballero, F.J. López-Fraguas, and M. Rodríguez-Artalejo. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. *Proc. of the 5th International Symposium on Functional and Logic Programming (FLOPS'2001)*, Springer LNCS 2024, pp. 170–184, 2001.

- [5] R. Caballero and M. Rodríguez-Artalejo. A Declarative Debugging System for Lazy Functional Logic Programs. *ENTCS* 64, 63 pages, 2002.
- [6] R. Caballero and M. Rodríguez-Artalejo. *DDT*: A Declarative Debugging Tool for Functional Logic Languages. *Proc. FLOPS'04*, Springer LNCS 2998, pp. 70–84, 2004.
- [7] M. Comini, G. Levi, M.C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming* 39 (1–3): 43–93, 1999.
- [8] P. Deransart, M. Hermenegildo, and J. Małuszyński (Eds.) Analysis and Visualization tools for Constraint Programming: Constraint Debugging. Springer LNCS 1870, pp. 151–174, 2000.
- [9] G. Ferrand. Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method. *The Journal of Logic Programming* 4(3), 177–198, 1987.
- [10] G. Ferrand, W. Lesaint, and A. Tessier. Towards declarative diagnosis of constraint programs over finite domains. *ArXiv Computer Science e-prints*, 2003.
- [11] M. Hanus (ed.), Curry: an Integrated Functional Logic Language, Version 0.8, April 15, 2003. <http://www.informatik.uni-kiel.de/~mh/curry/>.
- [12] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Abstract Verification and Debugging of Constraint Logic Programs. *Proc. CSCLP'02*, pp. 1–14, 2002.
- [13] J.W. Lloyd. Declarative Error Diagnosis. *New Generation Computing* 5(2), 133–154, 1987.
- [14] F.J. López-Fraguas, M. Rodríguez-Artalejo, and R. del Vado-Vírveda. 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.
- [15] F.J. López-Fraguas and J. Sánchez-Hernández. *TOY*: A Multiparadigm Declarative System. *Proc. RTA'99*, Springer LNCS 1631, pp 244–247, 1999. System and documentation available at <http://toy.sourceforge.net>.
- [16] L. Naish. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming*, 1997-3.
- [17] H. Nilsson. 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.
- [18] H. Nilsson and P. Fritzson. Algorithmic Debugging of Lazy Funcional Languages. *Journal of Functional Programming*, 4(3):337–370, 1994.
- [19] B. Pope and L. Naish. Practical aspects of declarative debugging in Haskell 98. *Proc. PPDP'03*, ACM Press, pp. 230–240, 2003.
- [20] E.Y. Shapiro. Algorithmic Program Debugging. *The MIT Press*, Cambridge, 1982.
- [21] A. Tessier and G. Ferrand. 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.
- [22] R. del Vado-Vírveda. Declarative Constraint Programming with Definitional Trees. *Proc. FroCoS'05*, Springer LNAI 3717 pp. 184–199, 2005.