Declarative Debugging in Constraint Logic Programming

Alexandre Tessier

LIFO, Université d'Orléans, BP 6759, 45067 Orléans Cedex 2, France
Alexandre.Tessier@lifo.univ-orleans.fr, http://www.univ-orleans.fr/~tessier

Abstract. This paper is motivated by the declarative insufficiency diagnosis of constraint logic programs, but focuses only on theoretical viewpoints. Constraint logic program semantics is redefined in terms of proof trees using a cover relation. We give a theoretical framework where declarative diagnosis method can be studied thanks to the inductive nature of the semantics. We define the notions of symptoms and errors and prove that if there exists a symptom then there exists an error.

1 Introduction

A great strength of Constraint Logic Programming (CLP) is its declarative nature. For a declarative language (with a semantics independent of its execution model), it is essential to consider a declarative error notion. Indeed, it is incoherent to use only low level debugging tools based on an understanding of the operational behaviour of the system and to give up a high level knowledge of the declarative meaning of the programs. But, the success of a declarative debugging tool is directly related to the language declarativity level. From this viewpoint, CLP is used in a much more declarative way than LP. In particular, the availability of global constraints and disequations makes useless: "cut", "is", "var", negation (by failure), etc. Declarative means that the user does not need to understand the operational behaviour of the system; it is evident that a computer cannot diagnose errors in a program without being told a part of what should be computed by the program. But, only the *expected declarative semantics* of the program is required.

Declarative error diagnosis in Logic Programming (LP) was introduced by E. Y. Shapiro [18] under the name of *algorithmic debugging*. But, declarative diagnosis of constraint logic programs is relatively unexplored. We focus in this paper on the theoretical aspects of the *declarative insufficiency diagnosis* of *constraint logic programs*. That is, how to declaratively point a small piece of erroneous code when an answer is lacking (see [13] for the wrong answer problem). Declarative debugging algorithms usually proposed in LP strongly use the Herbrand's models properties. They cannot be straightforwardly lifted to CLP: In CLP, Herbrand's interpretations do not represent program semantics any more. Some elements of the constraint domain are not finitely expressible in the program language (e.g. π in CLP($\mathcal{R}$) [10]). In fact, they are only handled through constraints. In LP, each answer, which is a logical consequence of the program, is

covered by a *single* more general computed answer. (the Herbrand's domain has the INC property [15]). In CLP, there is no single cover any more. For example, let us consider the CLP($\mathcal{R}$) program: $\{p(x) \leftarrow x < 0; p(x) \leftarrow x \geq 0\}$. Assume we expect the answer *true* to the goal $\leftarrow p(x)$. It is not because *true* does not occur (in the less general sense) in $\{x < 0, x \geq 0\}$ that an answer is lacking. We must consider the whole answer set and check if it covers the expected answer. In fact, the $\mathbb{R}$-theory has for logical consequence $\forall x(true \rightarrow (x < 0 \vee x \geq 0))$.

Practical implementations use incomplete constraint solvers with regard to theoretical framework [11] based on a constraint theory or interpretation. Our program semantics abstract the constraint semantics: a constraint interpretation $\mathcal{D}$ or a (non satisfaction complete) constraint theory $\mathcal{T}$ or an (incomplete) constraint solver $\mathcal{A}$ are particular cases. We show that classical program semantics [11, 8, 15, 12, 19] are instance of our own. We can remark that the three previous possible constraint semantics have not the same behaviours:

- $\mathcal{T}$ *or* $\mathcal{D}$ *vs.* $\mathcal{A}$: For example, the constraint solver of CLP($\mathcal{R}$) provides three kinds of answers: `yes` (satisfiable constraint), `no` (unsatisfiable constraint), `maybe` (it cannot decide). It answers `yes` for $x \times x = 1 \wedge x = 1$ and `maybe` for $x \times x = 1$, nevertheless $\models \exists x(x \times x = 1 \wedge x = 1) \rightarrow \exists x(x \times x = 1)$ (where $\models F$ means that F is a logical consequence of the empty theory); it answers `no` for $x = 1 \wedge x = 0$ and `maybe` for $x \times x = 1 \wedge x \times x = 0$, nevertheless $\models \neg\exists x(x = 1 \wedge x = 0) \rightarrow \neg\exists x(x \times x = 1 \wedge x \times x = 0)$.
- $\mathcal{T}$ *vs.* $\mathcal{D}$: Let us consider the program $\{int(x) \leftarrow x = 0\,, int(x) \leftarrow x = y + 1, int(y)\}$. When constraint semantics is based on the domain $\mathbb{N}$, the declarative answer *true* to the goal $\leftarrow int(x)$ is not covered by a finite part of the computed answers $\{x = 0, x = 1, \ldots, x = i, \ldots\}$. But when constraint are interpreted through a theory, a model of which is $\mathbb{N}$, *true* is not a declarative answer to $\leftarrow int(x)$, because of some non standard models of the $\mathbb{N}$-theory. When constraint semantics is based on a theory, if a constraint is covered by an infinite set of constraints, it is covered by a finite part of the set (compactness theorem of the first order logic).

We introduce a *cover relation* between a constraint c and a constraint set C, denoted by $c \vdash C$ and read: c is covered by C. The cover relation abstracts the different previous cover notions. We find the entailment relation when C is a singleton, [17] shows the interest of a framework based on the underlying inference relation rather than a constraint interpretation or theory.

We reformulate program semantics bases. Our approach, in terms of *proof trees* based on a *cover relation*, is an extension of the grammatical view of LP [4]. Proof trees give an intrinsic definition to the answers provided by a program. They are built on two kinds of rules: program rules and cover rules. Proof trees determine the declarative semantics while proof trees which only use program rules are an abstraction of the operational semantics. Relations between declarative semantics and operational semantics is then better explained.

The inductive nature of our semantics is adapted to study declarative diagnosis in the algorithmic debugging way. In this framework, where we just consider missing answers, two notions of symptoms are defined: *incompleteness symptoms*

and *insufficiency symptoms*. Two associated kinds of errors are defined: *non covered constrained atoms* and *non completely covered constrained atoms*. We prove that if there exists a symptom then there exists an error. We propose a diagnosis algorithm which, given a *computed* symptom, localizes an error in the program through implementation independent interaction with the expected semantics.

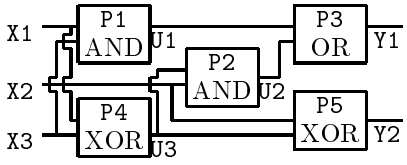
The paper is organized as follow: Sect. 2 defines the program language; Sect. 3 defines the semantics, it presents the cover relation; Sect. 4 studies the declarative insufficiency diagnosis; and the conclusion recapitulates main ideas.

2 Preliminaries

Let us consider once and for all four sets which define the program language: an infinite set of *variables* V; a set of *function symbols* Σ; a set of *constraint predicate symbols* Π_c; a set of *program predicate symbols* Π_p. Atomic formulas build on (V, Π_p) of the form $p(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are distinct variables, are called *atoms*. The constraint language CONST is a subset of the first order language build on (V, Σ, Π_c). We assume that it is closed under variable renaming, conjunction and existential quantification. A *constraint* is a formula of CONST. A *clause* is a $(n+2)$-tuple $(0 \leq n)$ denoted by $a_0 \leftarrow c \,\square\, a_1, \ldots, a_n$, where each a_i is an atom and c is a constraint. A *program* is a set of clauses. A *constrained atom* is a pair $a[c]$ where a is an atom and c is a constraint. If F is a formula then $var(F)$ denotes the free variables of F. If c is a constraint and a is an atom, then $\exists_{-a} c$ denotes the existential closure of c except on the variables of a. In order to simplify, we consider atomic goals rather than general goals. Indeed, given a general goal $\leftarrow g$, we can add the clause $p(var(g)) \leftarrow g$ (p is a new predicate symbol) and consider answers to the atomic goal $\leftarrow p(var(g))$.

This paper is illustrated by an example in PrologIII, due to A. Colmerauer [2]. The reader must not confuse our aim which is to diagnose error in a program and the example which detects a defective component in a binary adder.

Example 1. We are interested in detecting the single defective components of a binary adder which computes the binary sum of three bits `X1`,`X2`,`X3` in the form of a binary number given in two bits `Y1`,`Y2`. See [2] for further explanation.




```
go(C,E,S) :- circuit(C,E,S),enu(E) {C=[0',0',0',0',1'],S=[1',1']}.
circuit(C,E,S) :- at_most_1(C,X) {C=[P1,P2,P3,P4,P5],E=[X1,X2,X3],
  S=[Y1,Y2],X=1',~P1=>(U1<=>(X1&X3)),~P2=>(U2<=>(X2&U3)),~P3=>
  (Y1<=>(U1|U2)),~P4=>(U3<=>~(X1<=>X3)),~P5=>(Y2<=>~(X2<=>U3))}.
at_most_1(L,X) {L=[],X=0'}.
at_most_1(L,X) :- at_most_1(Q,Z) {L=[Y|Q],X=Y|Z,Y&Z=0'}.
```

```
boolean(X) {X=0'}. boolean(X) {X=1'}.
enu(L) {L=[]}. enu(L) :- boolean(X),enu(Q) {L=[X|Q]}.

?- go(C,E,S).
{C = [0',0',0',0',1'], E = [0',1',1'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',0',1'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',1',0'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',1',1'], S = [1',1']}
```

`circuit(C,E,S)` expresses the relation between inputs, outputs and the defective component. `C` is the component list, a component is defective when its value is `1'`, `E` is the input list and `S` is the output list. $go(C, E, S)$ provides the inputs when the component `P5` is defective and the outputs are `1',1'`. The last answer appears because a defective component does not always provide an erroneous output (implications in the program).

3 Semantics

In LP, a declarative answer to the goal $\leftarrow a$ is a substitution θ such that $P \models a\theta$. For each declarative answer, there exists a *single* more general computed answer. The similar results in CLP *should be* that, if $P \models c \rightarrow a$ (or $P, \mathcal{T} \models c \rightarrow a$ or $P \models_{\mathcal{D}} c \rightarrow a$) then there exists a *single* computed answer to the goal $\leftarrow a$ "more general" than c. In general this is wrong. Nevertheless, we would keep a similar property which links declarative answer and computed answer. In fact, each declarative answer is covered by a (possibly infinite) set of computed answers. This relation is called the cover relation.

Definition 1. A *cover relation* $\vdash$ is a binary relation over $\text{CONST} \times 2^{\text{CONST}}$. It verifies, for each constraint c, such that $c \vdash C$, the six following properties:

CONJ let c' be a constraint and C' be a constraint sets such that $c' \vdash C'$, then $c \wedge c' \vdash \{c'' \wedge c''' \mid c'' \in C, c''' \in C'\}$;
REFL $c \vdash \{c\}$;
EXIS let x be a variable, then $\exists x\, c \vdash \{\exists x\, c' \mid c' \in C\}$;
TRAN let $(C_{c'})_{c' \in C}$ be a constraint set family such that $c' \vdash C_{c'}$, $c' \in C$, then $c \vdash \bigcup_{c' \in C} C_{c'}$;
MONO let $C' \supseteq C$, then $c \vdash C'$;
RENA let θ be a variable renaming, then $c\theta \vdash \{c'\theta \mid c' \in C\}$.

(CONST, $\vdash$) defines a constraint system in a sense similar to [17, 9], except we do not assume C is finite (e.g. when constraint semantics is an interpretation).

In general, the cover relation is deduced from: a constraint interpretation $\mathcal{D}$, denoted by $\vdash_{\mathcal{D}}$, $c \vdash_{\mathcal{D}} C$ if each solution of c is solution of a constraint of C; a constraint theory $\mathcal{T}$, denoted by $\vdash_{\mathcal{T}}$, $c \vdash_{\mathcal{T}} C$ if, for each model $\mathcal{D}$ of $\mathcal{T}$, $c \vdash_{\mathcal{D}} C$; a constraint solver $\mathcal{A}$, denoted by $\vdash_{\mathcal{A}}$ and defined by the least relation verifying the six properties, such that $c \vdash_{\mathcal{A}} \emptyset$ if the constraint solver answers `no` for c.

Our declarative semantics has no logical nature. It is defined inductively by two kinds of rules. First rules stem from clauses of the program and the second ones are defined from the cover relation.

Definition 2. For each renamed clause $a \leftarrow c \,\square\, a_1, \ldots, a_n$ of P and each constraint $c_1, \ldots, c_n$, we define the *program rule*:

$$\frac{\{a_1[c_1], \ldots, a_n[c_n]\}}{a[\exists_{-a}(c \wedge c_1 \wedge \cdots \wedge c_n)]} \qquad \text{(when } n = 0 \text{ we deduce axiom } \frac{\emptyset}{a[\exists_{-a}\, c]}\text{)}$$

For each atom a, each constraint c and each constraint set C such that $c \vdash C$, we define the *cover rule*:

$$\frac{\{a[c'] \mid c' \in C\}}{a[c]} \qquad \text{(when } C = \emptyset \text{ we deduce axiom } \frac{\emptyset}{a[c]}\text{)}$$

As to each rule system, we associate, in the classical way, a notion of well-founded *proof trees* [1] (see Ex. 2). Proof trees determine the declarative semantics while proof trees which only use program rules determine an abstraction of the operational semantics (these one are finite):

Definition 3. An *answer* to the goal $\leftarrow a$ is the root $a[c]$ of a well-founded proof tree. A *computed answer* to the goal $\leftarrow a$ is the root $a[c]$ of a well-founded proof tree which only use program rules (thus the tree is finite).

Remark. When $c \vdash \emptyset$, for each atom a, $a[c]$ is a *trivial* answer. It is independent of the program. The system tries to eliminate, as far as possible, these trivial answers which have no interest for the user. For example, consider the CLP($\mathcal{R}$) program: $\{p(x,y) \leftarrow x > y \,\square\, ; q(x,y) \leftarrow v = x \times x \wedge w = y \times y \,\square\, p(v,w); r(x) \leftarrow x = y \,\square\, p(x,y); r(x) \leftarrow x = y \,\square\, q(x,y)\}$. There exists two answers to the goal $\leftarrow r(x)$: $r(x)[x > x]$ and $r(x)[x \times x > x \times x]$. According to $\mathbb{R}$, both constraints are unsatisfiable. Only the first one is removed.

Example 2. $go(C,E,S)[C = [0',0',0',0',1'] \wedge S = [1',1'] \wedge E = [0',1',1']]$ is a computed answer to the goal $\leftarrow go(C,E,S)$. See the following proof, denoted by A, where constraints are simplified for further legibility. There exists a similar proof, denoted by B, for the computed answer $go(C,E,S)[C = [0',0',0',0',1'] \wedge S = [1',1'] \wedge E = [1',1',0']]$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\textit{at\_most\_1}(C_5,X_5)[C_5=[] \wedge X_5=0']}}{\textit{at\_most\_1}(C_4,X_4)[C_4=[1'] \wedge X_4=1']}}{\textit{at\_most\_1}(C_3,X_3)[C_3=[0',1'] \wedge X_3=1']}}{\textit{at\_most\_1}(C_2,X_2)[C_2=[0',0',1'] \wedge X_2=1']} \quad \dfrac{}{\textit{boolean}(X''')[X'''=1'] \; \textit{enu}(E''')[E'''=[]]}}{\textit{at\_most\_1}(C_1,X_1)[C_1=[0',0',0',1'] \wedge X_1=1']} \quad \dfrac{}{\textit{boolean}(X'')[X''=1'] \quad \textit{enu}(E'')[E''=[1']]}}{\textit{at\_most\_1}(C\_,X\_)[C\_=[0',0',0',0',1'] \wedge X\_=1']} \quad \dfrac{}{\textit{enu}(E')[E'=[1',1']]} \quad \dfrac{}{\textit{boolean}(X')[X'=0']}}{\textit{circuit}(C,E,S)\,[C=[0',0',0',0',1'] \wedge S=[1',1'] \wedge E=[0',1',1']} \quad \dfrac{}{\textit{enu}(E)[E=[0',1',1']]}}{\textit{go}(C,E,S)[C=[0',0',0',0',1'] \wedge S=[1',1'] \wedge E=[0',1',1']]}$$

$go(C,E,S)[\exists A\,(C = [0',0',0',0',1'] \wedge S = [1',1'] \wedge E = [A,1',\sim A])]$ is an answer to $\leftarrow go(C,E,S)$ (because of $\exists A\,(E = [A,1',\sim A]) \vdash \{E = [0',1',1'], E = [1',1',0']\}$ and the **CONJ** property):

$$\frac{A \qquad\qquad B}{go(C,E,S)[\,\exists\, A\ (C=[0',0',0',0',1'] \wedge\ S=[1',1'] \wedge\ E=[A,1',\sim A]\ )]}$$

We define the program semantics as two constrained atom sets called the *success sets*, the first one describes the operational semantics while the second one describes the declarative semantics:

Definition 4. The success set associated to the program P
– is $SS(P) = \{a[c] \mid a[c]$ is a computed answer$\}$;
– and the cover relation $\vdash$ is $SS_{\vdash}(P) = \{a[c] \mid a[c]$ is an answer$\}$.

Remark. Let $SS^{\vdash\emptyset}(P) = \{a[c] \mid a[c] \in SS(P), not(c \vdash \emptyset)\}$, then $SS^{\vdash_{\mathcal{T}}\emptyset}(P) = SS(P,\mathcal{T})$ of [11], $SS^{\vdash_{\mathcal{D}}\emptyset}(P) = lfp(S_P^{\mathcal{D}})$ of [12], $SS^{\vdash_{\mathcal{D}}\emptyset}(P) = SS_3(P,\mathcal{D})$ of [8].

Success sets can also be defined as least fixed point of immediate consequence operators (from constrained atom sets to constrained atom sets):

Definition 5. $T_P(I) = \{a[c] \mid$ there exists a program rule $\dfrac{\{a_1[c_1],\ldots,a_n[c_n]\}}{a[c]}$, $a_i[c_i] \in I, i = 1,\ldots,n\}$
$T_{\vdash}(I) = \{a[c] \mid$ there exists a cover rule $\dfrac{\{a[c'] \mid c' \in C\}}{a[c]}, a[c'] \in I, c' \in C\}$
$T^{\cup}_{\vdash,P}(I) = T_P(I) \cup T_{\vdash}(I)$
$T^{\circ}_{\vdash,P}(I) = T_{\vdash}(T_P(I))$

Lemma 6. *1.* $lfp(T_P) = T_P \uparrow \omega = SS(P)$
2. $SS_{\vdash}(P) = \{a[c] \mid$ *there exists* C, $c \vdash C$ *and* $a[c'] \in SS(P)$, $c' \in C\}$
3. $lfp(T^{\cup}_{\vdash,P}) = T_{\vdash}(lfp(T_P)) = T_{\vdash}(SS(P)) = SS_{\vdash}(P) = T^{\cup}_{\vdash,P} \uparrow \omega + 1$
4. $lfp(T^{\cup}_{\vdash,P}) = lfp(T^{\circ}_{\vdash,P})$

Proof. 1. is a consequence of the Knaster-Tarski's theorem; *2.* is shown by induction on rules using the $\vdash$ properties; *3.* is a corollary of *2.*; *4.*$\subseteq$ $lfp(T^{\cup}_{\vdash,P}) = T_{\vdash}(lfp(T_P))$ and the **REFL** property of $\vdash$; *4.*$\supseteq$ for each ordinal number α there exists an ordinal number β such that $T^{\circ}_{\vdash,P} \uparrow \alpha \subseteq T^{\cup}_{\vdash,P} \uparrow \beta$.

The least fixed point of T_P corresponds to roots of proof trees only using program rules, the least fixed point of $T^{\cup}_{\vdash,P}$ correspond to roots of proof trees and the least fixed point of $T^{\circ}_{\vdash,P}$ correspond to roots of proof trees which use program rules at even depth and cover rules at odd depth. Note that proof trees are well-founded but may be infinite ($SS_{\vdash}(P) = T^{\cup}_{\vdash,P} \uparrow \omega + 1$).

On the one hand, for each proof tree, there exists a proof tree with same conclusion which uses a single cover rule, and it is used at the proof tree root (well known results when $\vdash$ is deduced from $\mathcal{D}$ or $\mathcal{T}$). On the other hand, for each proof tree, there exists a proof tree with same conclusion which alternates program rules and cover rules. The previous results are essential because they assert that, on the one hand, answers defined by covering computed answers and, on the other hand, answers defined by applying alternately program rules and cover rules are exactly all the declarative answers. Equality of the three sets guarantees consistency of definitions and results of Sect. 4. We emphasize that, in general, $gfp(T^{\circ}_{\vdash,P}) \neq T_{\vdash}(gfp(T_P)) \neq gfp(T^{\cup}_{\vdash,P})$.

4 Declarative Insufficiency Diagnosis

This section extends, to CLP, works on declarative error diagnosis for LP based on the Shapiro's method. Answers provided by a program are sometimes symptom of errors in the program. Error diagnosis is error localization when a symptom (missing answer in this paper) is observed. Our purpose is to elaborate an algorithm which assists the user, as well as possible, to search errors in the program when an answer is missing.

Example 3. In order to illustrate this section, we consider the program DETECT2 which is an erroneous implementation of program DETECT1 of Ex. 1:

```
go(C,E,S) :- circuit(C,E,S),enu(E) {C=[0',0',0',0',1'],S=[1',1']}.
circuit(C,E,S) :- at_most_1(C,X) {C=[P1,P2,P3,P4,P5],E=[X1,X2,X3],
  S=[Y1,Y2], X=1', ~P1=>(U1<=>(X1&X3)), ~P2=>(U2<=>(X2&U3)), ~P3=>
  (Y1<=>(U1|U2)), ~P4=>(U3<=>(~X1&~X3)), ~P5=>(Y2<=>(~X2&~U3))}.
at_most_1(L,X) {L=[],X=0'}.
at_most_1(L,X) :- at_most_1(Q,Z) {L=[Y|Q],X=Y|Z,Y&Z=0'}.
boolean(X) {X=0'}. boolean(X) {X=1'}.
enu(L) {L=[]}. enu(L) :- boolean(X),enu(Q) {L=[X|Q]}.

?- go(C,E,S).
{C = [0',0',0',0',1'], E = [0',1',0'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',0',1'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',1',1'], S = [1',1']}
```

The *expected semantics* of a program P is formalized by a set of expected constrained atoms denoted by I.

We say that the program P is *finitely incomplete* according to I for the atom a if there exists a finite SLD-tree for the goal $\leftarrow a$ whose set of successes is C and there exists a constraint c such that $a[c] \in I$ but not$(c \vdash C)$. Then, $a[c]$ is called *a computed symptom*. Thus, a computed symptom is an element of I which is not in $T_{\vdash}(lfp(T_P))$. We recall, once more, that $T_{\vdash}(lfp(T_P)) = lfp(T^{\cup}_{\vdash,P}) = lfp(T^{\circ}_{\vdash,P})$.

Example 4. $go(C,E,S)[\exists A\,(C = [0',0',0',0',1'] \wedge S = [1',1'] \wedge E = [A,1',\sim A])]$ is a computed symptom for the program DETECT2 (see Ex. 2 and Ex. 3).

Definition 7. P is *complete* (resp. *sufficient*) for the constrained atom $a[c]$ according to I if, for each constraint c' such that $c' \vdash \{c\}$ and $a[c'] \in I$: $a[c'] \in lfp(T^{\circ}_{\vdash,P})$ (resp. $a[c'] \in gfp(T^{\circ}_{\vdash,P})$).

$a[c]$ is *covered* by P according to I if $a[c] \in T^{\circ}_{\vdash,P}(I)$.

$a[c]$ is *completely covered* by P according to I if, for each constraint c' such that $c' \vdash \{c\}$ and $a[c'] \in I$: $a[c']$ is covered by P according to I.

A *strong insufficiency* is a constrained atom $a[c] \in I$ non covered by P according to I (i.e. $a[c] \in I - T^{\circ}_{\vdash,P}(I)$).

A *weak insufficiency* is a constrained atom $a[c]$ non completely covered by P according to I (i.e. there exists c', $c' \vdash \{c\}$ and $a[c']$ is a strong insufficiency).

$a[c]$ is an *incompleteness symptom* (resp. *insufficiency symptom*) of P according to I if $a[c] \in I - lfp(T^{\circ}_{\vdash,P})$ (resp. $a[c] \in I - gfp(T^{\circ}_{\vdash,P})$).

The interest is that the definitions are formulated in a uniform framework and extend existing definitions of LP [18, 6, 5, 14, 3, 7, 16]. Thus, they are better understood, particularly thanks to proof trees and $T^{\circ}_{\vdash,P}$. In the particular case of LP: our framework contributes to the comparison between above papers.

Lemma 8. *1. If $a[c]$ is a strong insufficiency then $a[c]$ is a weak insufficiency.*
2. There exists a strong insufficiency iff there exists a weak insufficiency.
3. If $a[c]$ is an insufficiency symptom then $a[c]$ is an incompleteness symptom.
4. There exists a strong insufficiency if there exists an insufficiency symptom.

Proof. *1.* and *2.* follows the definitions; *3.* because $lfp(T^{\circ}_{\vdash,P}) \subseteq gfp(T^{\circ}_{\vdash,P})$; *4.* if $I \not\subseteq gfp(T^{\circ}_{\vdash,P})$ then I is not a post fixed point of $T^{\circ}_{\vdash,P}$.

Lemma 8 is essential inasmuch as it expresses links between the different notions. The particular case where the error is the symptom establishes the correspondence between strong insufficiency and insufficiency symptom, between weak insufficiency and incompleteness symptom. It emerges from this that there exists two families of definitions concerning missing answers. The two families exists in LP and constraint introduction makes clear subtleties in their differences, thanks to $T^{\circ}_{\vdash,P}$, because its least fixed point is equal to $T_{\vdash}(lfp(T_P))$ (the two sets are proved equal to $lfp(T^{\cup}_{\vdash,P})$) and because there is not the single cover property any more. It is interesting to note that $T^{\circ}_{\vdash,P}$ has been selected in the definitions, whereas it is not the most natural to define $SS_{\vdash}(P)$.

Lemma 9. *If $a[c]$ is a computed symptom then $a[c]$ is an insufficiency symptom.*

Proof. The proof needs to define SLD-tree which is out of scope. It fully uses induction and $\vdash$ properties and is based on the three following lemmas: *1.* If there exists a finite SLD-tree for the goal $\leftarrow a$ with successes $c_1, \ldots, c_m$ then there exists an integer k such that, for each constraint c, if $a[c] \in T_P \downarrow k$ then $c \vdash \emptyset$ or $c \in \{c_1, \ldots, c_m\}$. *2.* For each integer n, if $a[c] \in T^{\circ}_{\vdash,P} \downarrow n$ then there exists C such that $c \vdash C$ and, for each constraint $c' \in C$, $a[c'] \in T_P \downarrow n$. *3.* If $a[c] \in T^{\circ}_{\vdash,P} \downarrow \omega$ and there exists a finite SLD-tree for the goal $\leftarrow a$ with success $c_1, \ldots, c_m$ then there exists $C \subseteq \{c_1, \ldots, c_m\}$ such that $c \vdash C$ (i.e. $a[c] \in SS_{\vdash}(P)$).

Consequently, if there exists a computed symptom then there exists a strong insufficiency and a weak insufficiency (regardless of the computation rule). Now, we propose an algorithm inspired from [5]. The input is a computed symptom according to the standard strategy and the output is a weak insufficiency. Thus the user just answers `yes` or `no` while it has to answer by constraint in an algorithm which searches a strong insufficiency.

The *search forest* for a consists of a tree for each non-unary clause $R \in P$, of the definition of the predicate symbol of a, such that if $a \leftarrow c_1 \square b_1, \ldots, b_n$ is a variant of R then $\mathrm{not}(c_1 \wedge \exists_{-a} c \vdash \emptyset)$. The tree is rooted by $\langle b_1, c_1 \wedge \exists_{-a} c \rangle$. Let

$\langle b_i, c_i \rangle$ be the label of a node and $C = \{c'' \mid b_i[c'']$ is an answer and not$(c'' \wedge c_i \vdash \emptyset)\}$ then, for each $c' \in C$, $\langle b_i, c_i \rangle$, has a node labeled by $\langle b_{i+1}, c' \rangle$ if $i < n$ and labeled by $\langle \Box, c' \rangle$ if $i = n$.

We call *incompleteness question* for the node N labeled by $\langle b_i, c_i \rangle$: "Is there exist c' such that $a[c'] \in I$, $c' \vdash \{c_i\}$, not$(c' \vdash \{c'' \mid \langle x, c'' \rangle$ labels a child of $N\})$?"

Let $a[c]$ be the algorithm input, we ask incompleteness question for nodes labeled by $\langle b_i, c_i \rangle$ $(b_i \neq \Box)$ of the search forest of $a[c]$. If the answer is `yes` then the algorithm is recursively called with $b_i[\exists_{-b_i} c_i]$. If the answer is `no` for every node then the algorithm terminates with output $a[c]$ which is a weak insufficiency.

Note that incompleteness questions of our algorithm are finite because the input is a computed symptom (i.e. the standard SLD-tree is finite).

Example 5. Insufficiency diagnosis session for the program DETECT2. Constraint are simplified by Prolog III.
Symptom: $go(C, E, S)[\exists A(C = [0', 0', 0', 0', 1'] \wedge E = [A, 1', \sim A] \wedge S = [1', 1'])]$
Is there exist a constraint c such that $go(C, E, S)[c]$ expected, $c \vdash \{\exists A(C = [0', 0', 0', 0', 1'] \wedge E = [A, 1', \sim A] \wedge S = [1', 1'])\}$ and not$(c \vdash \emptyset)$? `YES`
Is there exist a constraint c such that $circuit(C, E, S)[c]$ expected, $c \vdash \{\exists A(C = [0', 0', 0', 0', 1'] \wedge E = [A, 1', \sim A] \wedge S = [1', 1'])\}$ and not$(c \vdash \emptyset)$? `YES`
Is there exist a constraint c such that $at\_most\_1(C, X)[c]$ expected, $c \vdash \{C = [0', 0', 0', 0', 1'] \wedge X = 1'\}$ and not$(c \vdash \{C = [0', 0', 0', 0', 1'] \wedge X = 1'\})$? `NO`
Error: $circuit(C, E, S)[\exists A(C = [0', 0', 0', 0', 1'] \wedge E = [A, 1', \sim A] \wedge S = [1', 1'])]$
The algorithm has isolated a definition which is responsible for the symptom. Error is due to the definition of *circuit* (XOR has been coded by NOT OR).

5 Conclusion

We reformulate operational semantics and declarative semantics of constraint logic programs in a uniform formal framework. Computed answers and declarative answers are straightforwardly linked through an abstract cover relation. The program semantics can be defined in reference to a constraint interpretation or theory, and can also take into account incompleteness of constraint solvers.

The interest of this novel reformulation is to study error diagnosis. We have defined insufficiency symptoms and incompleteness symptoms and the two associated kinds of errors in a uniform framework contributing to their comparison. Thanks to the reformulation in terms of proof trees, the results becomes more natural and straightforward.

The diagnosis algorithm behaves as an intelligent trace following the straight path toward the error, generates only relevant information, provides a limited piece of erroneous code (predicate definition) which makes correction easier.

Since the two families of symptoms and errors are defined in a single framework, we can study algorithms in which they collaborate.

References

1. P. Aczel. *An Introduction to Inductive Definitions*, chapter 7, pages 739–782. In

J. Barwise, editor, Handbook of Mathematical Logic. North-Holland Publishing Company, 1977.
2. A. Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.
3. M. Comini, G. Levi, and G. Vitiello. Abstract Debugging of Logic Programs. In L. Fribourg and F. Turini, editors, *Logic Program Synthesis and Transformation and Metaprogramming*, volume 883 of *Lecture Notes in Computer Science*, pages 440–450. Springer-Verlag, 1994.
4. P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming.* MIT Press, 1993.
5. W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. Algorithmic Debugging with Assertions. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 501–522. MIT Press, 1989.
6. G. Ferrand. Error Diagnosis in Logic Programming: an adaptation of E. Y. Shapiro's method. *Journal of Logic Programming*, 4:177–198, 1987.
7. G. Ferrand. The Notions of Symptom and Error in Declarative Diagnosis of Logic Programs. In P. A. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 40–57. Springer-Verlag, 1993.
8. M. Gabbrielli and G. Levi. Modeling answer constraints in Constraint Logic Programs. In V. A. Saraswat and K. Ueda, editors, *International Conference on Logic Programming*, pages 238–252. MIT Press, 1991.
9. R. Giacobazzi, S. K. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, 25(3):191–248, 1995.
10. N. C. Heintze, J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. Yap. *The CLP($\mathcal{R}$) Programmer's Manual Version 1.2.* IBM Thomas J. Watson Research Center, 1992.
11. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In 14^{th} *ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM, 1987.
12. J. Jaffar and M. J. Maher. Constraint Logic Programming: a survey. *Journal of Logic Programming*, 19-20:503–581, 1994.
13. F. Le Berre and A. Tessier. Declarative Incorrectness Diagnosis in Constraint Logic Programming. In P. Lucio, M. Martelli, and M. Navarro, editors, *Joint Conference on Declarative Programming*, pages 379–391, 1996.
14. J. W. Lloyd. Declarative Error Diagnosis. *New Generation Computing*, 5(2):133–154,, 1987.
15. M. J. Maher. A Logic Programming view of CLP. In Warren, editor, *International Conference on Logic Programming*, pages 737–753. MIT Press, 1993.
16. L. Naish. A declarative debugging scheme. Technical Report 95/1, Department of Computer Science, University of Melbourne, 1995.
17. V. A. Saraswat. The Category of Constraint Systems is Cartesian-Closed. In *Logic In Computer Science*. IEEE Press, 1992.
18. E. Y. Shapiro. *Algorithmic Program Debugging.* ACM Distinguished Dissertation. MIT Press, 1982.
19. P. Van Hentenryck. Constraint Logic Programming. *Knowledge Engineering Review*, 6(3):151–194, 1991.

This article was processed using the LaTeX macro package with LLNCS style