

Positive and Negative Diagnosis for Constraint Logic Programs in terms of proof skeletons

G rard Ferrand and Alexandre Tessier

LIFO, Universit  d'Orl ans, BP 6759, 45067 Orl ans Cedex 2, France
{Gerard.Ferrand,Alexandre.Tessier}@lifo.univ-orleans.fr,http://www.univ-orleans.fr/~tessier

Abstract

The paper is motivated by the declarative debugging of constraint logic programs. It deals with the theoretical basis of declarative incorrectness diagnosis. It starts with a reformulation of the program semantics in terms of proof tree skeletons, which is suitable for declarative diagnosis study. The program semantics is explained in terms of positive semantics and negative semantics. The problem of wrong answer is treated as an incorrectness of the positive semantics while the problem of missing answer is treated as an incorrectness of the negative semantics. Incorrectness diagnosis is based on a well-founded relation over computation states.

1 Introduction

The first motivation for a work on debugging is a computation producing a result which is considered as incorrect. Since there is a result, it is not an infinite computation. An incorrect result is called a *symptom*. This notion of *symptom* depends on some *expected properties* of the program, so a symptom is a result which is *not expected*. If the motivation is not debugging but program proving, with expected properties defined by a specification, the impossibility of producing a symptom is the definition of *partial correctness*. However our notion of expected properties may be more general than a complete specification of the program semantics. From a conceptual viewpoint we have only to presuppose an *oracle* which is able to decide that a result is expected or not. In practice the presentation of a result can be very intricate so the ability for deciding could seem unrealistic. However this presupposition is necessary to give a meaning to debugging questions and in fact it is the notion of expected properties which has to be realistic. In practice the oracle can be embodied by the programmer or by other means (for example assertions [2, 4, 3]) and the expected properties can be defined by using an abstract (approximate, graphical, ...) view of the computed result. This question is relevant to the *presentation problem* ([12]).

Symptoms are caused by *errors* in the program. An error is a piece of code. The first step of debugging is *error diagnosis* that is error localization. If we carry on comparing with program proving, for example in Hoare style, an error is a construction in the program which makes a proof of partial correctness impossible and the proof method amounts to proving that there is no error. That amounts to saying that if there is a symptom then there is an error.

This paper deals with error diagnosis of Constraint Logic Programs. For such high level languages traditional tracing techniques become difficult to use because of the complexity of the computational behaviour. Moreover it would be incoherent to use only low level debugging tools whereas these languages benefit from a declarative semantics (as opposed to operational semantics).

Declarative Debugging was introduced in Logic Programming (LP) by Shapiro (and called *Algorithmic Program Debugging* [18]) (see also [11, 5, 6, 17, 4, 15, 16]). *Declarative* means that the user has no need to consider the computational behaviour of the logic programming system, he needs only a declarative knowledge of the expected properties of the program.

The previous reflections on symptoms and errors can be applied to Constraint Logic Programming (CLP). But because of the relational nature of CLP languages we have to split the notion of (finite) computation in two notions. That is to say that we have to split the notion of result in two notions.

A *goal* being given, there is a first notion of result which is a *computed answer constraint*, the computation being a *success derivation*. This is a *first level of computation*. In the formal logical semantics for CLP ([8, 9, 13, 7]) the relation between the goal $\leftarrow G$ and the computed answer constraint c is formalized by using the implication $c \rightarrow G$. Even from a purely operational viewpoint we can consider that $c \rightarrow G$ is computed.

But there is a *second level* of computation that is to say another notion of finite computation which is represented by a *finite SLD tree* (derivation tree or search tree). Now if c_1, \dots, c_n are all the computed answer constraints of this finite SLD tree, their relation with the goal $\leftarrow G$ is formalized by the implication $G \rightarrow c_1 \vee \dots \vee c_n$. If $n = 0$ (*finite failure*) this implication amounts to $\neg G$. To be more formal, $G \rightarrow c_1 \vee \dots \vee c_n$ occurs along with the *completion* of the program and to be more precise with its *only if* part (while at the first level $c \rightarrow G$ occurs along with its *if* part that is the program itself). Even from a purely operational viewpoint we can consider that $G \rightarrow c_1 \vee \dots \vee c_n$ is computed at this second level of computation.

These remarks motivate that we call *positive* the first operational level and *negative* the second one.

A symptom at the *positive level* will be called a *positive symptom*. To say that $c \rightarrow G$ is a *positive symptom* is an abstract way to say that c is a *wrong answer* to G . If the expected semantics is defined in a logical framework with respect to an *intended interpretation*, $c \rightarrow G$ is not true in this intended interpretation.

A symptom at the *negative level* will be called a *negative symptom*. To say that $G \rightarrow c_1 \vee \dots \vee c_n$ is a *negative symptom* is an abstract way to say that there is *not enough answers* to $\leftarrow G$, there are *missing answers*, G is *not covered* by c_1, \dots, c_n . If the expected semantics is defined in a logical framework with respect to an *intended interpretation*, then $G \rightarrow c_1 \vee \dots \vee c_n$ is not true in this intended interpretation.

For the two levels, the basic principles of the diagnosis will be the same: there exists some *well founded* relation such that the diagnosis amounts to the search for a *minimal symptom*. A notion of error, called *incorrectness*, is associated with each minimal symptom. An error at the positive (resp. negative) level will be called a *positive* (resp. *negative*) *incorrectness*.

We use a description of the operational semantics of CLP in terms of proof skeletons which is an extension of the Grammatical View of LP ([1]) and we take

into account the possible incompleteness of the constraint solver. This framework is well adapted to take advantage of the properties of *confluence* (independence of the computation rule) and *compositionality* of this semantics.

Confluence is basic to define notions which are *declarative* that is to say which do not depend on a particular computational behaviour. (Moreover the notion of skeleton gives prominence to the fact that the results computed at the positive level are intrinsic in a sense which is stronger than only independence of the computation rule in a top-down computation).

Because of compositionality, it is sufficient to consider positive symptoms $c \rightarrow G$ where G is just an *atom* and negative symptoms $G \rightarrow c_1 \vee \dots \vee c_n$ where G is just the *conjunction of a constraint and an atom*.

In former works on declarative debugging ([18, 6, 11]) the duality positive / negative was not introduced in the same way. From an abstract viewpoint the former notions of *incorrectness* symptom and *insufficiency* symptom can be defined in an inductive framework, to be more precise induction (*least fixpoint*) for *incorrectness* and co-induction (*greatest fixpoint*) for *insufficiency*. In (C)LP the notion of insufficiency can be applied to missing answers because the *finite failure* set of a definite program and the greatest fixpoint of the immediate consequence operator are disjoint. In some sense in this paper we use only one abstract inductive scheme which is based on a least fixpoint to define a notion of symptom (incorrectness symptom) and error (incorrectness). But it is applied to two different semantics levels: positive level for wrong answers and negative level for missing answers. In fact the least fixpoint is only implicit and the inductive framework appears only through a well founded relation.

Our approach gives a new framework to understand and to generalize the algorithm of [4, 14] (to what extent it depends on the standard computation rule and how it can be generalized to CLP).

The paper is only devoted to the theoretical basis of the approach. An example is developed in [19]. The justifications were not given in this previous paper, they are now given in the present paper.

2 Operational Semantics

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 . Each symbol is equipped with its arity. $var(E)$ denotes the set of free variables of E , where E is a formula built over the first order language $\mathcal{L}(V, \Sigma, \Pi_p \cup \Pi_c)$.

An *atom* is a particular atomic formula $p(\tilde{x})$ of $\mathcal{L}(V, \emptyset, \Pi_p)$, where \tilde{x} is a sequence of distinct variables. ATOM denotes the set of atoms.

The set of *basic constraints* CONST is a subset of $\mathcal{L}(V, \Sigma, \Pi_c)$ closed under variable renaming. A *store* is a member of the least set which contains CONST and closed under conjunction and existential quantification. We denote by STORE the set of stores. For practical purpose a store is always written $\exists x_1 \dots \exists x_n F$, where F is a conjunction of basic constraints using the usual transformations over formulas. We use the following notations to denote a store, where $\tilde{x} = \{x_1, \dots, x_n\}$ is a sequence of variables: $\exists_{\tilde{x}} F$ denotes $\exists x_1 \dots \exists x_n F$; $\exists F$ denotes $\exists_{var(F)} F$; $\exists_{-\tilde{x}} F$ denotes

$\exists_{var(F) \setminus \tilde{x}} F$; $\exists_{-a} F$, where a is an atom, denotes $\exists_{-var(a)} F$.

A *clause* is a 3-tuple, denoted by $a \leftarrow c \square A$, where a is an atom, c is a store and A is a finite sequence of atoms. We define: $head(a \leftarrow c \square A) = a$, $store(a \leftarrow c \square A) = c$.

A *program* is a family of clauses. In this paper P is a program. The set of indexes of P is denoted by CN . A *name* of clause is a member of CN . The *definition* of $p \in \Pi_p$ in P is the sub-family of clauses of P whose head predicate symbol is p ; it is indexed by the subset $CN_p \subseteq CN$. We assume CN_p is finite for each $p \in \Pi_p$. The clause whose name is u is denoted by $clause(u)$.

A *goal* is an atom a , written $\leftarrow a$ for “historical” reasons. We consider atomic goals rather than general goals in order to simplify the framework and this is always possible by adding a new clause whose body is the goal and head is a new relation over the free variables of the goal.

A *constrained atom* is a pair $a \leftarrow c$, where a is an atom and c is a store. A *covered atom* is a pair $a \rightarrow C$, where a is an atom and C is a *disjunction* of stores. A *local cover of atom* is a 3-tuple $c \square a \rightarrow C$, where c is a store, a is an atom and C is a set of stores.

2.1 Skeletons

We introduce the central tool of the reformulation: a skeleton is a tree which put together the clauses used along a derivation regardless of the computation rule. From an abstract viewpoint, a derivation is a top-down construction of a skeleton. This construction itself is a sequence of skeletons.

Definition 1 A skeleton is an oriented tree S labeled by $CN \cup \Pi_p$, such that for each node N , $lab_S(N)$ denoting the label of N : if $lab_S(N) \in \Pi_p$ then N is a leaf; if $lab_S(N) \in CN$ and $clause(lab_S(N)) = a \leftarrow c \square p_1(\tilde{x}_1) \cdots p_n(\tilde{x}_n)$ then N has n children N_1, \dots, N_n , and each child N_i is labeled by either p_i , or a name of clause of CN_{p_i} .

The root of the skeleton S is denoted by $root(S)$. The *program predicate symbol* associated with a node N of a skeleton S is: $lab_S(N)$ if $lab_S(N) \in \Pi_p$; p if $lab_S(N) \in CN_p$. We say that S is a skeleton for p (or $\leftarrow p(\tilde{x})$) when the predicate symbol associated with $root(S)$ is p . We denote by $undef(S)$ the set of nodes of S labeled by members of Π_p (it is the set of *undefined nodes*) and we denote by $def(S)$ the set of the other nodes (it is the set of *defined nodes*). A *complete skeleton* is a skeleton S such that $undef(S) = \emptyset$. If $undef(S) \neq \emptyset$ then S is an *incomplete skeleton*.

For each $u \in CN$, we denote by $sq(u)$ the unique skeleton rooted by u such that the children of $root(S)$ are labeled by members of Π_p . For each $p \in \Pi_p$, we denote by $sq(p)$ the unique skeleton rooted by p .

Now, we want to associate a global store to a skeleton which contains the stores of the clauses of the skeleton. But as usual we are confronted with the problem of clause renaming.

Definition 2 Let S be a skeleton. A renaming function for S is a function ren_S (from $def(S)$ to the set of renamed clauses of P) such that:

1. for each node $N \in def(S)$: $ren_S(N) = clause(lab_S(N))\theta$, where θ is a renaming; let $ren_S(N) = a \leftarrow c \square a_1 \cdots a_n$ and N_1, \dots, N_n be the children of N , for each $i = 1, \dots, n$, if $N_i \in def(S)$ then $head(ren_S(N_i)) = a_i$;

2. for each pair of distinct nodes $N_1, N_2 \in \text{def}(S)$, if $\text{ren}_S(N_1) = a_1 \leftarrow c_1 \sqcap A_1$ and $\text{ren}_S(N_2) = a_2 \leftarrow c_2 \sqcap A_2$ then $(\text{var}(c_1 \sqcap A_1) \setminus \text{var}(a_1)) \cap (\text{var}(c_2 \sqcap A_2) \setminus \text{var}(a_2)) = \emptyset$.

A skeleton S of depth ≥ 1 and a renaming function ren_S for S being given, for each node N of S , the *atom associated with N* is: $\text{head}(\text{ren}_S(N))$ if $N \in \text{def}(S)$; a_i if $N \in \text{undef}(S)$ is the i^{th} child of N' and $\text{ren}_S(N') = a \leftarrow c \sqcap a_1 \cdots a_i \cdots a_n$.

The store system associated with a skeleton S and a renaming function ren_S for S is $\text{const}(S, \text{ren}_S) = \bigcup_{N \in \text{def}(S)} \text{store}(\text{ren}_S(N))$. When S is finite the conjunction of the stores of $\text{const}(S, \text{ren}_S)$ is a store and we identify $\text{const}(S, \text{ren}_S)$ and the conjunction of its members.

The renaming function ren_S is said to be a *renaming function* for S and $\leftarrow p(\tilde{x})$ if either $\text{root}(S) \in \text{CN}_p$ or $p(\tilde{x}) = \text{head}(\text{ren}_S(\text{root}(S)))$. If S is finite then the *store associated with S* and $p(\tilde{x})$ is $\text{AC}(S, p(\tilde{x})) = \exists_{-\tilde{x}} \text{const}(S, \text{ren}_S)$. Note that $\text{AC}(S, p(\tilde{x}))$ does not depend on ren_S .

Now we want to distinguish “satisfiable” skeletons.

2.2 Reject Criterion

From an abstract viewpoint, a possibly incomplete constraint solver is a *reject criterion* verifying some monotonicity.

Definition 3 A reject criterion is an unary relation RC over STORE such that for each $c \in \text{RC}$: for each renaming θ , $c\theta \in \text{RC}$; for each $c' \in \text{STORE}$, $c \wedge c' \in \text{RC}$; $\emptyset \notin \text{RC}$ (\emptyset is the empty conjunction of stores).

From a reject criterion RC , we define a relation over the set of skeletons, also denoted by RC , in the following way: $S \in \text{RC}$ if there exists a finite part c of $\text{const}(S, \text{ren}_S)$, where ren_S is a renaming function for S , such that $c \in \text{RC}$. We emphasize this property does not depend on ren_S . Note that $\text{sq}(p)$, $p \in \Pi_p$, is not rejected. In this paper, a reject criterion RC is supposed to be given.

Definition 4 A (computation) state is a skeleton S such that $S \notin \text{RC}$.

Note that infinite computation states are convenient for studying infinite computations.

2.3 Positive Computation (SLD derivation), Positive Answer

Definition 5 Let \hookrightarrow be the binary relation over the set of states, called transition relation between states, defined by: $S \hookrightarrow S'$ if there exists a leaf $N \in \text{undef}(S)$ and a clause name $u \in \text{CN}_{\text{lab}_S(N)}$ such that S' is obtained by grafting $\text{sq}(u)$ on the node N in S . Then we say S' derives from S by the leaf N .

\hookrightarrow defines a transition system between (computation) states. S is an *initial state* if there exists $p \in \Pi_p$ such that $S = \text{sq}(p)$. S is a *final state* if S is finite and for each state S' : $S \not\hookrightarrow S'$; then S is a *success state* if S is complete, it is a *failure state* otherwise.

A *SLD derivation* for the goal $\leftarrow p(\tilde{x})$ (or for p) is a (finite or infinite) sequence of states $S_1 \cdots S_i \cdots$ such that $S_1 = \text{sq}(p)$, for each $j = 2 \cdots i \cdots$: $S_{j-1} \hookrightarrow S_j$ and

the sequence is infinite or the last state is a final state. A *success* (resp. *failure*) SLD derivation is a finite SLD derivation which ends by a success (resp. *failure*) state.

Definition 6 A positive answer is the last state of a success SLD derivation.

We denote by $\text{success}(a)$ the set of positive answers for a goal $\leftarrow a$.

Lemma 7 S is a positive answer if and only if S is a finite complete state.

This is in our framework the basis of the result known as “independence of the computation rule” or “confluence”.

If S is a positive answer for $\leftarrow a$ then $\text{AC}(S, a)$ is a *positive answer store* for $\leftarrow a$.

Definition 8 A computation rule is a mapping r which provides a leaf of $\text{undef}(S)$ for each incomplete state S .

Given a computation rule r , we define the binary relation \hookrightarrow_r included in \hookrightarrow as follow: $S \hookrightarrow_r S'$ if S' derives from S by the leaf $r(S)$.

Lemma 9 The positive answers are independent of the computation rule. (see lemma 7)

2.4 Negative Computation (SLD tree), Negative Answer

Definition 10 Let r be a computation rule and $p \in \Pi_p$. Let \hookrightarrow_r^p be the binary relation included in \hookrightarrow_r defined as follow: $S \hookrightarrow_r^p S'$ if and only if $S \hookrightarrow_r S'$, where S and S' are skeletons for p .

Lemma 11 Let $\text{dom}_r^p = \{S \mid \text{sq}(p) \hookrightarrow_r^p S\}$. $(\text{dom}_r^p, \hookrightarrow_r^p)$ is a tree: \hookrightarrow_r^p has the property of a parent relation over dom_r^p . It is the SLD tree for p (or for the goal $\leftarrow p(\tilde{x})$) according to the computation rule r . A branch of an SLD tree $(\text{dom}_r^p, \hookrightarrow_r^p)$ is a SLD derivation (according to r) for p .

Lemma 12 The set of success leaves of a SLD tree for p does not depend on the computation rule. Given a computation rule r , S is a positive answer for $\leftarrow p(\tilde{x})$ if and only if S is a success leaf of $(\text{dom}_r^p, \hookrightarrow_r^p)$.

Definition 13 A negative answer for the goal $\leftarrow p(\tilde{x})$ is $\text{success}(p(\tilde{x}))$ if there exists a computation rule r such that the SLD tree $(\text{dom}_r^p, \hookrightarrow_r^p)$ is finite. Note that the negative answer for $\leftarrow a$ does not depend on r . Then, the negative answer store for the goal $\leftarrow p(\tilde{x})$ is $\bigvee_{S \in \text{success}(p(\tilde{x}))} \text{AC}(S, p(\tilde{x}))$.

From an operational viewpoint, only finite computation are interesting. That is the reason we assume the existence of a finite SLD tree to define the negative answer. In the definition of the positive answer, we have no hypothesis on the finiteness of a SLD tree, but we also consider finite computation only. But it is another level of computation: the SLD derivations.

We have defined two levels of answers corresponding to two levels of computation: positive answers (SLD derivations, positive computations) and negative answers (SLD trees, negative computations).

This two levels of computation can be considered for every languages using a non deterministic computations. Note that, at each level, each computation is deterministic.

The second level is said negative because it insures that there is no more answers of the first level (called positive).

2.5 Success Sets

Definition 14 *The positive success set is $SS^+ = \bigcup_{a \in \text{ATOM}} \{a \leftarrow \text{AC}(S, a) \mid S \in \text{success}(a)\}$. SS^+ is a set of constrained atoms.*

The negative success set is $SS^- = \{a \rightarrow \bigvee_{S \in \text{success}(a)} \text{AC}(S, a) \mid a \in \text{ATOM} \text{ and there exists a negative answer for } \leftarrow a\}$. SS^- is a set of covered atoms.

The two success sets are the success sets of the two levels of computation. We emphasize that it is not possible to deduce a success set from the other one. We can just deduce a part of SS^+ from SS^- .

The negative computation is a generalization of the finite failure usually considered to express the negative semantics of a program. Finite failure corresponds to the particular case where there exists a negative answer for $\leftarrow a$ but $\text{success}(a) = \emptyset$; then a implies the empty disjunction in SS^- (in a logical view $\neg a$ is in SS^-).

3 Declarative Diagnosis

3.1 A General Scheme based on a Well-founded Relation

Let E be a set and R be a well-founded relation over E^2 . Let $I \subseteq E$ be the set of expected members of E , we assume $I \neq E$.

Definition 15 *A symptom of R wrt I is a member of $E \setminus I$.*

While R is well-founded, $E \setminus I$ has at least one minimal element according to R^+ (the transitive closure of R).

Definition 16 *A minimal symptom of R wrt I is a minimal member of $E \setminus I$ according to R^+ .*

So, if there exists a symptom of R wrt I then there exists a minimal symptom of R wrt I .

For example an obvious diagnosis algorithm is to build a sequence $x_0 \cdots x_i \cdots$ of symptoms such that, for each i , $x_{i+1} R x_i$. The sequence is finite because R is well-founded. If the last element of the sequence has no predecessor which is in I then it is a minimal symptom. But every other strategy to detect a minimal symptom could be a good strategy.

3.2 Positive Partial Correctness (wrong positive answer)

Let S be the final state of a success SLD derivation for the goal $\leftarrow a$ and let us assume that the positive answer store $\text{AC}(S, a)$ to $\leftarrow a$ is not expected.

The positive answer S can be built from the skeletons grafted on the nodes of S . Each skeleton grafted on a node of S is a complete and non rejected skeleton, so a positive answer.

Let $<_+$ be the binary relation over the set of positive answers defined as follow: $S' <_+ S$ if S' is grafted on a child of $root(S)$ in S .

Lemma 17 $<_+$ is a well-founded relation.

An oracle being given, which points out if a positive answer is expected, we can trivially apply the definition of the previous section. For example, given a positive answer S for $\leftarrow a$, the oracle can answer considering the constrained atom $a \leftarrow AC(S, a)$. That is, the oracle is a relation over the success set associated with positive computations: SS^+ .

From an abstract viewpoint, we can consider that the expected properties of the program at the positive level are formalized by a set of expected constrained atoms I . Obviously this set must be in accordance with the reject criterion RC. For example, if the expected properties are given by an expected interpretation \mathcal{I} then \mathcal{I} is an expansion of the interpretation \mathcal{D} of the constraint language and the reject criterion RC is assumed to be *correct* wrt \mathcal{D} , that is, if $c \in RC$ then c is unsatisfiable in \mathcal{D} .

Definition 18 A positive incorrectness symptom is a constrained atom $a \leftarrow c$ of $SS^+ \setminus I$.

The cause of a symptom of $<_+$ is a minimal symptom, but what caused the minimal symptom?

Let S be a minimal symptom and $clause(lab_S(root(S))) = a \leftarrow c \square a_1 \cdots a_n$. S is a symptom, thus $a \leftarrow AC(S, a) \in SS^+ \setminus I$. It is minimal, thus if S_1, \dots, S_n are the answers grafted on the child of $root(S)$ in S then $a_i \leftarrow AC(S, a_i) \in I$. The reason of the appearance of the minimal symptom is that $a \leftarrow \exists_{-a}(c \wedge \bigwedge_{i \in \{1, \dots, n\}} AC(S_i, a_i)) \in SS^+ \setminus I$ but each $a_1 \leftarrow AC(S_i, a_i) \in SS^+ \cap I$. The clause $a \leftarrow c \square a_1 \cdots a_n$ caused the symptom and the store $\bigwedge_{i \in \{1, \dots, n\}} AC(S_i, a_i)$ provides the reason of the incorrectness of the clause.

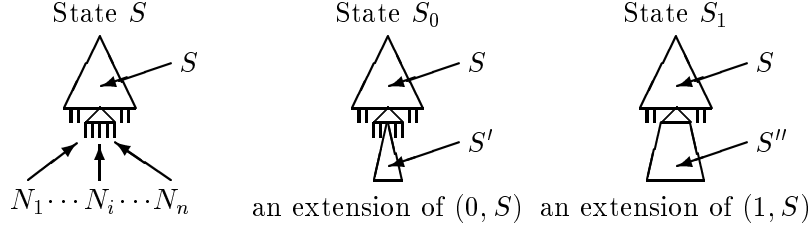
Definition 19 A positive incorrectness is a $n+1$ -tuple $\langle a \leftarrow c \square a_1 \cdots a_n, c_1, \dots, c_n \rangle$ where $a \leftarrow c \square a_1 \cdots a_n \in P$ and the c_i 's are stores, such that $\{a_i \leftarrow c_i \mid i \in \{1, \dots, n\}\} \subseteq I$, but $a \leftarrow \exists_{-a}(c \wedge c_1 \wedge \cdots \wedge c_n) \notin I$.

Lemma 20 If there exists a positive incorrectness symptom then there exists a positive incorrectness.

The positive incorrectness diagnosers deduced from this framework are those described in an inductive framework in [10]; and several optimisations can improved the diagnosers but this is out of the scope of the paper.

3.3 Negative Partial Correctness (wrong negative answer)

Consider a negative answer store C for $\leftarrow p(\tilde{x})$ and assume that C is not expected wrt some expected properties of the program. Then we can distinguish two cases: a member c of C should not be in C ; or a store c is missing in C .



$N_i = r(S)$; $N_1 \dots N_n$ is the sibling of $r(S)$; S' is a (possibly infinite) complete skeleton; S'' is a set of n (possibly infinite) complete skeletons.

Figure 1: Extension of a b -state.

The first case is a problem of partial positive correctness. The store c , which should not be in C , is a positive answer store for $\leftarrow p(\tilde{x})$ then it is obviously a problem of wrong positive answer; c is the store associated with the last state of a success SLD derivation that is an erroneous (finite) positive computation.

For the second case we cannot study a single SLD derivation. Something goes wrong in the SLD tree. It is an erroneous (finite) negative computation. We focus on this case now.

Let $(dom_r^p, \hookrightarrow_r^p)$ be the the finite SLD tree which provides the incorrect negative answer store.

Intuitively an incomplete state $S \in dom_r^p$ can be seen in two different ways: depending on whether we consider either $r(S)$ or $r(S)$ and its sibling. For this purpose, we use the disjoint sum of the set of states with itself.

Let $E_0 = \{S \mid sq(p) \hookrightarrow_r^p S, undef(S) \neq \emptyset\}$ and $E_1 = \{S \mid sq(p) \hookrightarrow_r^p S\}$ (\hookrightarrow_r^p and \hookrightarrow_r^p are respectively the reflexive transitive and transitive closures of \hookrightarrow_r^p). Let $E_0 \oplus E_1$ be the disjoint sum of E_0 and E_1 . This set is isomorphic to $(\{0\} \times E_0) \cup (\{1\} \times E_1)$ and we identify them in order to simplify notations. A member of $E_0 \oplus E_1$ is a pair (b, S) , where $b \in \{0, 1\}$ and $S \in E_0$ if $b = 0$ or $S \in E_1$ if $b = 1$. (b, S) is called a b -state; if $b = 0$ it is a 0-state, if $b = 1$ it is a 1-state.

Lemma 21 *For each 1-state $(1, S)$ there exists a unique S' such that $S' \hookrightarrow_r S$. S' is called the parent of S and will be denoted by $parent(S)$.*

Definition 22 (see Fig. 1) *Let $(0, S)$ be a 0-state. The state S' is an extension of $(0, S)$ if S' is obtained by grafting a complete skeleton on $r(S)$ in S .*

Let $(1, S)$ be a 1-state. The state S' is an extension of $(1, S)$ if S' is obtained by grafting complete skeletons on some undefined nodes of S in S such that:

$$undef(parent(S)) \setminus \{r(parent(S))\} = undef(S')$$

We denote by $prol_b(S)$ the set of the extensions of (b, S) .

Note that $prol_0(sq(p)) = success(p(\tilde{x}))$; if $r(parent(S))$ has no sibling then $prol_1(S) = \{S\}$, where $(1, S)$ is a 1-state; if $r(S)$ has no undefined sibling then $prol_0(S) = prol_1(S)$, where S is an incomplete non initial state.

Lemma 23 *Let $(1, S)$ be a 1-state:*

$$prol_1(S) = \{S' \in prol_0(parent(S)) \mid lab_{S'}(r(parent(S))) = lab_S(r(S_0))\}$$

Let $(0, S)$ be a 0-state:

$$prol_0(S) = \bigcup_{S \hookrightarrow_r S'} prol_1(S')$$

A computation rule r is without *co-routining* if, for each incomplete state S , $r(S)$ is an undefined leaf in the set of the deepest undefined leaves. For example, the standard strategy of Prolog is without *co-routining*.

Now, let us assume that the computation rule r is without co-routining.

Lemma 24 *Let (b, S) be a b -state. $prol_b(S) \subseteq dom_r^p$. Moreover, for each $S' \in prol_b(S)$: S' is a descendant of S in $(dom_r^p, \hookrightarrow_r^p)$ (or $S' = S$ if $b = 1$ and $r(parent(S))$ has no child in S).*

The previous lemma shows a noteworthy property of the SLD-trees without co-routining. For example, we deduce that if $(dom_r^p, \hookrightarrow_r^p)$ is finite then, for each b -state (b, S) : $prol_b(S)$ is a finite set of finite states.

Let $<_-$ be the binary relation over $E_0 \oplus E_1$ defined as follow:

- for each 0-state $(0, S)$, if $S \hookrightarrow_r^p S'$ then: $(1, S') <_- (0, S)$;
- for each 1-state $(1, S)$, if $r(parent(S))$ has a child in S (i.e. $prol_1(S) \neq \{S\}$) then: $(0, S) <_- (1, S)$
- for each 1-state $(1, S)$, if $r(parent(S))$ has a child in S then for each $S' \in prol_0(S)$: $(1, S') <_- (1, S)$

Lemma 25 *$<_-$ is a well-founded relation (because of the finiteness of the SLD tree $(dom_r^p, \hookrightarrow_r^p)$).*

We introduce, for the time being, a notion of positive answer store for a body of clause that is a pair $c \sqcap A$, where c is a store and A is a finite sequence of atoms.

Definition 26 *A positive answer store for $c \sqcap a_1 \cdots a_n$ is $c \wedge c_1 \wedge \cdots \wedge c_n$, such that, for each $i = 1, \dots, n$, c_i is a positive answer store for $\leftarrow a_i$ and $(c \wedge c_1 \wedge \cdots \wedge c_n) \notin RC$. We denote by $ans(c \sqcap A)$ the set of positive answer stores for $c \sqcap A$. When $ans(c \sqcap A)$ is finite we identify it with the disjunction of its members.*

Note that if $c \notin RC$ then $ans(c \sqcap \varepsilon) = \{c\}$ (ε is the empty sequence of atoms); if $c \in RC$ then for each finite sequence of atoms A : $ans(c \sqcap A) = \emptyset$; $ans(\emptyset \sqcap a) = \{c \mid a \leftarrow c \in SS^+\}$.

Lemma 27 *For each store c , for each finite sequences of atoms A_1 and A_2 , for each atom a : $ans(c \sqcap A_1 \cdot a \cdot A_2) = \bigcup_{c' \in ans(c \sqcap a)} ans(c \wedge c' \sqcap A_1 \cdot A_2)$*

We associate a pair $c \sqcap A$ to each b -state (b, S) in the following way:

- let $(0, S)$ be a 0-state, ren_S be a renaming function for S , a be the atom associated with $r(S)$, the pair associated with $(0, S)$ is $\exists_{-a} const(S, ren_S) \sqcap a$;
- let $(1, S)$ be a 1-state, ren_S be a renaming function for S and A be the sequence of atoms associated with the children of $r(parent(S))$, the pair associated with $(1, S)$ is $\exists_{-var(A)} const(S, ren_S) \sqcap A$.

The pair associated with a b -state is defined up to a renaming.

Lemma 28 Let $(0, S)$ be a 0-state and $c \sqcap a$ be the pair associated with $(0, S)$:
 $ans(c \sqcap a) = \{\exists_{-a} const(S', ren_{S'}) \mid S' \in prol_0(S), a \text{ is the atom associated with } r(S)\}$.

Let $(1, S)$ be a 1-state, $c \sqcap a_1 \cdots a_n$ be the pair associated with $(1, S)$ and $N_1 \cdots N_n$ be the children of $r(\text{parent}(S))$ in S :

$ans(C \sqcap a_1 \cdots a_n) = \{\exists_{-var(a_1 \cdots a_n)} const(S', ren_{S'}) \mid S' \in prol_1(S), a_i \text{ is the atom associated with } N_i\}$.

A b -state is expected if $c \sqcap A \rightarrow ans(c \sqcap A)$ is expected. Intuitively a b -state is expected if no positive answer store for $c \sqcap A$ is missing in $ans(c \sqcap A)$.

In order to motivate the definition we consider that the expected properties of P are given by an expected interpretation \mathcal{I} ; \mathcal{I} is an extension of the constraint interpretation \mathcal{D} and RC is correct wrt \mathcal{D} .

But our theoretical framework can be extended to a more general notion of “expected” assuming that the relation “expected” has the following natural properties: for each store c , $c \sqcap \varepsilon \rightarrow \{c\}$ is expected ($ans(c \sqcap \varepsilon)$ does not depend on P); for each store c , for each set of stores C , for each family of set of stores $\{R_{c'}\}_{c' \in C}$, for each finite sequences of atoms A_1 and A_2 , for each atom a , if $c \sqcap a \rightarrow C$ is expected and, for each $c' \in C$, $c \wedge c' \sqcap A_1 \cdot A_2 \rightarrow R_{c'}$ is expected then $c \sqcap A_1 \cdot a \cdot A_2 \rightarrow \bigcup_{c' \in C} R_{c'}$ is expected.

Lemma 29 For each 1-state $(1, S)$, if the predecessors of $(1, S)$ by $<_-$ are expected then $(1, S)$ is expected.

So, if (b, S) is a minimal symptom (according to $<_-$) then $b = 0$.

Definition 30 A pair $c \sqcap a$ is a negative incorrectness symptom if $c \sqcap a \rightarrow ans(c \sqcap a)$ is not expected.

Definition 31 A constrained atom $a \leftarrow c$ is covered if:

1. let $a \leftarrow c_i \sqcap a_i^1 \cdots a_i^{n_i}$, $i = 1, \dots, n$, be the (renamed) clauses of the definition of the predicate symbol of a ;
2. for each $i = 1, \dots, n$, there exists n_i stores $c_i^1, \dots, c_i^{n_i}$ such that $a_i^j \leftarrow c_i^j$ is expected, $j = 1, \dots, n_i$;
3. for each $i = 1, \dots, n$, $a \leftarrow \exists_{-a}(c_i \wedge \bigwedge_{j=1, \dots, n_i} c_i^j)$ is expected;
4. $c \rightarrow \bigvee_{i=1, \dots, n} \exists_{-a}(c_i \wedge \bigwedge_{j=1, \dots, n_i} c_i^j)$ is true in the constraint interpretation.

A pair $c \sqcap a$ is completely covered if, for each store c' such that $c' \rightarrow c \sqcap a$ is expected (i.e. $c' \rightarrow c$ is true in the constraint interpretation and $a \leftarrow c'$ is expected), $a \leftarrow c'$ is covered.

Note that the previous definition is more intricate than in pure logic programming because there is no more independence of negated constraints [13]. But, if each valuation v is the unique solution of a store c_v then $v(a)$ is covered if there exists a clause of the definition of a such that v is a solution of the body of the clause in \mathcal{I} . And $c \sqcap a$ is completely covered if for each valuation v solution of $c \sqcap a$ in \mathcal{I} , $v(a)$ is covered.

Now, we define the notion of error associated with a minimal negative incorrectness symptom.

Definition 32 A negative incorrectness is a pair $c \sqcap a$ which is not completely covered.

Lemma 33 If there exists a negative incorrectness symptom then there exists a negative incorrectness.

Note that the definition of negative incorrectness symptom and negative incorrectness are identical to the definition of incompleteness symptom and weak insufficiency given in [19] (in [19] they are compared with the definition of insufficiency symptom and strong insufficiency). The novel framework (negative computation and diagnosis wrt a well-founded relation) define a wide family of algorithms for negative incorrectness diagnosis.

The algorithm proposed in [19] is a member of this family and is a lifting of the algorithm proposed in [4] for pure logic programs. But the family of diagnosers described by the novel framework is more general:

- they relax the requirement of the existence of a finite standard SLD tree and replace it by the existence of a finite SLD tree without co-routining;
- any order of the questions is followed, not only the order of the strategy used to build the SLD tree.

4 Conclusion

We have described a theoretical basis for an approach of declarative diagnosis of constraint logic programs in terms of proof skeletons. This abstract framework can be applied to various notion of expected properties.

Other notions of errors for missing positive answer can be considered in the same framework. Already in LP, two insufficiency notions was defined: a *non covered atom* ([18, 6, 11]) and a *non completely covered atom* ([4]). For example, [19] studies a generalisation of the two previous insufficiency notions.

We have clearly defined the set of questions to the oracle. Moreover every order of the questions is suitable provided that a minimal symptom is founded.

We emphasize that our approach takes into account the possible incompleteness of the constraint solver of the system (but the solver is assumed to be correct). It is worth noting that the incompleteness of the constraint solver cannot be the cause of the appearance of a symptom.

References

- [1] P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming*. MIT Press, 1993.
- [2] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59:133–155, 1988.
- [3] W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Fifth Generation Computer Systems*, pages 573–581, 1988.

- [4] 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.
- [5] G. Ferrand. Error Diagnosis in Logic Programming: an adaptation of E. Y. Shapiro’s method. *Journal of Logic Programming*, 4:177–198, 1987.
- [6] G. Ferrand. The Notions of Symptom and Error in Declarative Diagnosis of Logic Programs. In P. A. Fritzon, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 40–57. Springer-Verlag, 1993.
- [7] 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.
- [8] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In 14th *ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.
- [9] J. Jaffar and M. J. Maher. Constraint Logic Programming: a survey. *Journal of Logic Programming*, 19-20:503–581, 1994.
- [10] 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.
- [11] J. W. Lloyd. Declarative Error Diagnosis. *New Generation Computing*, 5(2):133–154,, 1987.
- [12] J. W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, 1995.
- [13] M. J. Maher. A Logic Programming view of CLP. In Warren, editor, *International Conference on Logic Programming*, pages 737–753. MIT Press, 1993.
- [14] S. Nadjm-Tehrani. Debugging Prolog Programs Declaratively. In *Workshop on Meta-programming in Logic*, pages 137–155, 1990.
- [15] L. Naish. Declarative Diagnosis of Missing Answers. *New Generation Computing*, 10(3):255–285, 1992.
- [16] L. Naish. A Declarative Debugging Scheme. Technical Report 95/1, Department of Computer Science, University of Melbourne, 1995.
- [17] L. M. Pereira. Rational Debugging in Logic Programming. In E. Y. Shapiro, editor, *International Conference on Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1986.
- [18] E. Y. Shapiro. *Algorithmic Program Debugging*. ACM Distinguished Dissertation. MIT Press, 1982.
- [19] A. Tessier. Declarative Debugging in Constraint Logic Programming. In J. Jaffar, editor, *Asian Computing Science Conference*, volume 1179 of *Lecture Notes in Computer Science*, pages 64–73. Springer-Verlag, 1996.