

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

G erard Ferrand and Alexandre Tessier

LIFO, Universit e d'Orl ans, BP 6759, 45067 Orl ans Cedex 2, France
{Gerard.Ferrand,Alexandre.Tessier}@lifo.univ-orleans.fr,hhttp://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 $\exists x_1 \dots \exists x_n F$; $\tilde{\exists} F$ 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 $\text{CN}_p \subseteq \text{CN}$. 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 $\text{CN} \cup \Pi_p$, 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 $lab_S(N) \in \text{CN}$ 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 $lab_S(N) \in \text{CN}_p$. 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 $u \in \text{CN}$, 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

