

Debugging Systems for Constraint Programming

ESPRIT 22532

Task T.WP2.1: Declarative Debugging in Constraint Programming

Deliverable D.WP2.1.M1.1

CLARIFICATION OF THE BASES OF DECLARATIVE DIAGNOSERS FOR CLP

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/SCIENCES/LIFO/Members/tessier/>

<http://discipl.inria.fr/>

Abstract: This report shows the theoretical bases of the techniques which will be used in implementation of declarative error diagnosers. It starts with a reformulation of the program semantics in terms of proof tree skeletons, which is suitable for declarative diagnosis studies. The program semantics is explained in terms of positive and negative semantics. Wrong answers are treated as incorrectness in the positive semantics while missing ones are treated as incorrectness in the negative semantics. This will permit to present in an uniform and simplified manner diagnosis algorithms of different kinds of errors.

Contents

1	Introduction	5
2	Abstract Notions of Symptoms and Errors for Inductive Definitions	7
2.1	Induction	8
2.2	Co-Induction	9
3	Constraint Logic Programs	9
3.1	Language	9
3.2	Atoms	10
3.3	Stores	10
3.4	Clauses	10
3.5	Notations	10
3.6	Program	11
3.7	If, Only-if, If-and-only-if Part	11
3.8	Expected Properties	12
4	Positive Side (Positive Diagnosis)	13
4.1	Skeletons	13
4.2	Partial Skeletons	14
4.3	Reject Criterion	15
4.4	Positive Computation (SLD derivation)	16
4.5	Positive Answer	17
4.6	Positive Partial Correctness (wrong positive answer)	18

4.7	Positive Diagnosis	18
4.8	Positive Partial Insufficiency (missing positive answer)	19
5	Negative Side (Negative Diagnosis)	19
5.1	Negative Computation (SLD tree)	19
5.2	Negative Answer	20
5.3	Extensions	21
5.4	Negative Proof tree	22
5.5	Negative Partial Correctness (wrong negative answer)	24
5.6	Negative Diagnosis	26
6	Conclusion	27

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 [3, 5, 4]) and the expected properties can be defined by using an abstract (approximate, graphical, ...) view of the computed result (see task 2.2 and work package 3 of the project). This question is relevant to the *presentation problem* ([15]).

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 (DD) was introduced in Logic Programming (LP) by Shapiro (and called *Algorithmic Program Debugging* [24]) (see also [8, 14, 6,

7, 21, 5, 18, 19]). *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 ([11, 12, 16, 10]) 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*. In our framework the well founded relation is very particular, it corresponds to a parent relation of a tree. This tree is a proof tree for a system of inductive rules.

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 ([2]) 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).

Our approach gives a new framework to understand and to generalize the algorithm of [5, 17] (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.

2 Abstract Notions of Symptoms and Errors for Inductive Definitions

Symptom and errors can be defined in an abstract scheme ([7, 19]). They have been first defined for definite programs by Shapiro ([24]) using essentially an inductive framework: on the one hand least fixpoint, i.e. *induction*, for the relation between incorrectness symptom (“wrong answer”) and incorrectness (a kind of error), on the other hand greatest fixpoint,

i.e. *co-induction*, for the relation between insufficiency symptom (“missing answer”) and insufficiency (another kind of error).

2.1 Induction

In an abstract inductive framework, the notions of symptom and error, and the relation between them, come straightforwardly from fixpoint theory:

Let H be a set and $T : 2^H \rightarrow 2^H$ a *monotonic* operator. So $\text{lfp}(T)$ (the *least fixpoint* of T) is defined, and it is the least $I \subseteq H$ such that $T(I) \subseteq I$. So $T(I) \subseteq I \Rightarrow \text{lfp}(T) \subseteq I$ (it is the basis of the *proof method by induction*).

Let $S \subseteq H$ and let us suppose that we are expecting that $\text{lfp}(T) \subseteq S$ but that actually we see an $h \in \text{lfp}(T)$ such that $h \notin S$. Such an h shows that something goes wrong in T . h is called a *symptom* (for T wrt S). But $\text{lfp}(T) \not\subseteq S$ so $T(S) \not\subseteq S$.

Now we have to define a convenient notion of *error*. Let us suppose that T is the *operator defined by a set \mathcal{R} of rules*, i.e. $T(I)$ is defined by: for any $h \in H$, $h \in T(I)$ if and only if there is a rule in \mathcal{R} the head of which is h and the body is a subset of I (a rule is a pair denoted by $h \leftarrow B$ where $h \in H$ and $B \subseteq H$. h is the *head* and B is the *body* of the rule).

$T(S) \not\subseteq S$ means that there is a rule $h' \leftarrow B'$ in \mathcal{R} with $B' \subseteq S$ and $h' \notin S$. The existence of such rules $h' \leftarrow B'$ explains that $\text{lfp}(T) \not\subseteq S$ i.e. the appearance of symptoms. So such a rule $h' \leftarrow B'$ is called an *error* (of \mathcal{R} wrt S).

Moreover, the set of rules defines a notion of *proof tree* and the members of $\text{lfp}(T)$ are exactly the roots of proof trees. In particular if the rules are *finitary* (i.e. their bodies are finite sets) a *proof tree* for \mathcal{R} is merely a *finite tree* where each node is (an occurrence of) a member of H and such that, for each node h , if its children are b_1, \dots, b_n then $h \leftarrow \{b_1, \dots, b_n\}$ is a rule of \mathcal{R} ; such a rule is said to be *in the proof tree*. (In particular if h is a *leaf* of the proof tree then $h \leftarrow \emptyset$ is in \mathcal{R}).

A relation of “*causality*” between an error and a symptom can be expressed with the notion of *proof tree*. It is easy to see that each symptom is the root of a proof tree in which there is a rule which is an error. This error can be seen as a “*cause*” of the symptom.

In some sense, to localize an error amounts to localize a *minimal* symptom in the proof tree. This notion of minimality refers to a *well-founded* relation which is the *parent relation*. Note that the transitive closure of this relation is also well-founded and it is a strict order. But it is exactly the parent relation which has to be considered and not the strict order; because otherwise the notion of error associated to a minimal symptom would not be a local notion since not only the children but all the descendant should not be symptoms.

2.2 Co-Induction

Fixpoint theory has also a *dual* aspect: $gfp(T)$ (the *greatest fixpoint* of T) is defined also, and it is the greatest $I \subseteq H$ such that $I \subseteq T(I)$. So $I \subseteq T(I) \Rightarrow I \subseteq GFP(T)$ (it is the basis of the *proof method by co-induction*).

Let $S \subseteq H$ and let us suppose that we are expecting that $S \subseteq GFP(T)$ but that actually we see an $h \in S$ such that $h \notin GFP(T)$. Such an h shows that something goes wrong in T . h is called a *co-symptom* (for T wrt S). But $S \not\subseteq GFP(T)$ so $S \not\subseteq T(S)$.

Let us suppose again that T is the *operator defined by a set \mathcal{R} of rules*. $S \not\subseteq T(S)$ means that there is a $h' \in S$ such that in \mathcal{R} there is no rule $h' \leftarrow B'$ with $B' \subseteq S$. Such a h' is called an *co-error* (of \mathcal{R} wrt S).

There is again a relation between a co-error and a co-symptom that can be expressed with a notion of “partial proof tree” which cannot be “completed”.

Lastly we can note that *no error and no co-error* means S is a *fixpoint* of T , and this implies $lfp(T) \subseteq S \subseteq GFP(T)$ (the converse is not valid).

3 Constraint Logic Programs

3.1 Language

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* Ξ ;
- a set of *program predicate symbols* Π .

Each symbol is equipped with its arity.

3.2 Atoms

A *pure atom* (in short *atom*) is an atomic formula $p(\tilde{x})$ built over the first order language $\mathcal{L}(V, \emptyset, \Pi)$, where \tilde{x} is a sequence of distinct variables. $ATOM$ denotes the set of atoms.

3.3 Stores

The set of *basic constraints* is a subset of the first order language $\mathcal{L}(V, \Sigma, \Xi)$ closed under variable renaming. A *store* is a member of the least set which contains the set of basic constraints and is closed under conjunction and existential quantification. We denote by $STORE$ the set of stores.

3.4 Clauses

A *body* is a member of the least set which contains $STORE \cup ATOM$ and is closed under conjunction and existential quantification. We denote by $BODY$ the set of bodies.

A *clause* is the \forall -closure of a formula $a \leftarrow b$, where $a \in ATOM$ and $b \in BODY$.

3.5 Notations

We use the following notations, where e is a first order formula built over the language $\mathcal{L}(V, \Sigma, \Pi \cup \Xi)$, $\tilde{x} = \{x_1, \dots, x_n\} \subseteq V$ and $a \in ATOM$:

- $var(e)$ denotes the set of free variables of e ;
- $\forall e$ denotes the universal closure of e ;

- $\exists e$ denotes the existential closure of e ;
- $\exists_{\tilde{x}} e$ denotes $\exists x_1 \cdots \exists x_n e$;
- $\exists_{-\tilde{x}} e$ denotes $\exists_{\text{var}(e) \setminus \tilde{x}} e$;
- $\exists_{-a} e$ denotes $\exists_{-\text{var}(a)} e$.

For practical purpose, using usual transformations over formulas,

- a store will be always identified with a formula $\exists x_1 \cdots \exists x_n c$, where c is a conjunction of basic constraints;
- a body will be always identified with a formula $\exists x_1 \cdots \exists x_n (c \wedge b)$, where c is a conjunction of basic constraints, b is a conjunction of atoms;
- a clause will be always identified with a formula $\forall (a \leftarrow c \wedge b)$, where a is an atom, c is a conjunction of basic constraints and b is a conjunction of atoms¹.

3.6 Program

A *program* is a family of clauses. In this paper P is a program. The set of indexes of P is denoted by $\delta(P)$.

A *name of clause* is a member of $\delta(P)$. The *definition* of $p \in \Pi$ in P is the sub-family of clauses of P whose head predicate symbol is p ; it is indexed by the subset $\delta_p(P) \subseteq \delta(P)$. We assume $\delta_p(P)$ is finite for each $p \in \Pi$. The clause whose name (i.e. index) is f is denoted by $\text{clause}(P, f)$.

3.7 If, Only-if, If-and-only-if Part

For each $p \in \Pi$, let

$$\text{IF}(P, p) = \forall (a \leftarrow \bigvee_{f \in \delta_p(P)} (\exists_{-a_f} b_f) \theta_f)$$

¹Note that $\forall (a \leftarrow c \wedge b)$ is equivalent to $\forall (a \leftarrow \exists_{-a} (c \wedge b))$.

where for each $f \in \delta_p(P)$: $clause(P, f) = \forall(a_f \leftarrow b_f)$ and θ_f is a renaming such that $a_f\theta_f = a$.

For each $p \in \Pi$, let $FI(P, p)$ be the formula obtained from $IF(P, p)$ using the reverse implication, i.e.

$$FI(P, p) = \forall(a \rightarrow \bigvee_{f \in \delta_p(P)} (\exists_{-a_f} b_f)\theta_f)$$

For each $p \in \Pi$, let $IFF(P, p)$ be the formula obtained from $IF(P, p)$ using an equivalence instead of the implication, i.e.

$$IFF(P, p) = IF(P, p) \wedge FI(P, p) = \forall(a \leftrightarrow \bigvee_{f \in \delta_p(P)} (\exists_{-a_f} b_f)\theta_f)$$

Remark. When the definition of p is empty: $IFF(P, p) = \forall(\neg p(\tilde{x}))$. ◇

We define the *if-part* of P (equivalent to P), the *only-if-part* of P and finally the *if-and-only-if-part* of P (the *completion* without constraint theory) as follows:

- $IF(P) = \{IF(P, p) \mid p \in \Pi\}$;
- $FI(P) = \{FI(P, p) \mid p \in \Pi\}$;
- $IFF(P) = \{IFF(P, p) \mid p \in \Pi\}$.

3.8 Expected Properties

To be more understandable, we assume that the expected properties are formalized by a \mathcal{D} -interpretation I , where \mathcal{D} is an interpretation of the constraint language.

I is expected to be a \mathcal{D} -model of $IFF(P)$.

4 Positive Side (Positive Diagnosis)

Following the ideas of [9] we consider two level of semantics: *positive semantics* and *negative semantics*. At the positive level, we consider formulae of the form $\forall(b \leftarrow c)$ ($b \in BODY$, $c \in STORE$).

Remark. $\forall(c_0 \wedge a_1 \wedge \dots \wedge a_n \leftarrow c)$ is equivalent to $\forall(c_0 \leftarrow c) \wedge \bigwedge_{i=1, \dots, n} \forall(a_i \leftarrow c)$. So at the positive level it is sufficient to consider formulae of the form $\forall(a \leftarrow c)$. \diamond

4.1 Skeletons

Ignoring constraint solver, the operational semantics of CLP can be seen as a search for proofs.

The formulae of the form $\forall(a \leftarrow c)$ are proved inductively using skeletons:

a *skeleton* is a typed term built over $\delta(P)$, i.e. the functions symbols are the names of clauses.

For each $f \in \delta(P)$, let $clause(P, f) = \forall(p(\tilde{x}) \leftarrow c \wedge p_1(\tilde{x}_1) \wedge \dots \wedge p_n(\tilde{x}_n))$,

- the arity of f is n ;
- the type of f is $p_1 \times \dots \times p_n \rightarrow p$.

The formula proved by a skeleton is defined inductively as follow: let s be a skeleton,

- if s is a constant (i.e. $clause(P, s)$ is a fact $\forall(a \leftarrow c)$) then the formula proved by s is

$$clause(P, s)$$

- if $s = f(s_1, \dots, s_n)$ and the formula proved by each s_i is $\forall(a_i \leftarrow c_i)$ and $clause(P, f) = \forall(a \leftarrow c \wedge a_1 \wedge \dots \wedge a_n)$ then the formula proved by s is

$$\forall(a \leftarrow c \wedge \exists_{-a_1} c_1 \wedge \dots \wedge \exists_{-a_n} c_n)$$

Lemma 1 *If $\forall(a \leftarrow c)$ is the formula proved by a skeleton then $\text{IF}(P) \models \forall(a \leftarrow c)$.*

Corollary 2 *Let \mathcal{D} be a constraint interpretation and \mathcal{T} be a constraint theory²:*

1. *if $\forall(a \leftarrow c)$ is the formula proved by a skeleton then $\text{IF}(P) \models_{\mathcal{D}} (\forall a \leftarrow c)$.*
2. *if $\forall(a \leftarrow c)$ is the formula proved by a skeleton then $\text{IF}(P), \mathcal{T} \models (\forall a \leftarrow c)$.*

4.2 Partial Skeletons

A derivation is a top-down construction of a skeleton.

In order to describe derivations, we define partial skeletons (i.e. with undefined sub-terms) by adding constants which represent undefined sub-terms: previous skeletons are now called *complete skeleton*. We generalize the notion of skeleton by adding for each $p \in \Pi$, p itself as a constant symbol whose type is $\rightarrow p$.

The formula proved by a skeleton s is defined inductively as follow:

- if s is a constant and $s \in \Pi$ then the formula proved by s is

$$\forall(s(\tilde{x}) \leftarrow s(\tilde{x}))$$

- if s is a constant and $s \in \delta(P)$ then the formula proved by s is

$$\text{clause}(P, s)$$

- if $s = f(s_1, \dots, s_n)$ and for each $i = 1, \dots, n$ the formula proved by s_i is $\forall(a_i \leftarrow b_i)$ and $\text{clause}(P, f) = \forall(a \leftarrow c \wedge a_1 \wedge \dots \wedge a_n)$, then the formula proved by s is

$$\forall(a \leftarrow c \wedge \exists_{-a_1} b_1 \wedge \dots \wedge \exists_{-a_n} b_n)$$

²A constraint interpretation (resp. theory) is an interpretation (resp. theory) of the constraint language.

Note that the formula proved by a skeleton is a clause which is a fact when the skeleton is complete.

In a skeleton s , an occurrence ν of the symbol denoted by $symbol(s, \nu)$ is a symbol $p \in \Pi$ only if ν is a leaf of s . Such a leaf is said undefined.

$undef(s)$ denotes the set of all the undefined leaves in the skeleton s . $def(s)$ denotes the set of the other occurrences.

Each atom of the body of the clause proved by s is associated to a $\nu \in undef(s)$. So each conjunction of atoms is associated with a subset of $undef(s)$.

Let $\forall(a \leftarrow c \wedge b)$ be the formula proved by a skeleton s , then the *store associated with s* is $store(s) = \exists c$ and the store associated with s and a set of free variables \tilde{x} is $store(s, \tilde{x}) = \exists_{-\tilde{x}} c$.

For each $f \in \delta(P)$, we denote by $sk(f)$ the skeleton $f(p_1, \dots, p_n)$ where each $p_i \in \Pi$.

Let \sqsubseteq be a partial order over the set of states defined by $s \sqsubseteq s'$ if:

- each occurrence of s is an occurrence of s' ;
- for each occurrence $\nu \in def(s)$: $symbol(s, \nu) = symbol(s', \nu)$.

4.3 Reject Criterion

From a practical viewpoint, there is no need to compute something like $\forall(a \leftarrow c \wedge b)$ if c is unsatisfiable: $\forall(a \leftarrow c \wedge b)$ is always true and does not depend on the program. It does not provide information on the relation p .

The CLP systems use a constraint solver which detect some unsatisfiable stores in order to stop useless computations. For decidability or efficiency purpose constraint solvers are often incomplete.

In our framework, the constraint solver is formalized by a property called the *reject criterion* verifying some monotonicity.

Definition 3 A reject criterion is a set RC which formalize a property of conjunctions of basic constraints such that if $c \in RC$:

- for each renaming θ , $c\theta \in \text{RC}$;
- for each basic constraint c' , $c \wedge c' \in \text{RC}$;
- $\emptyset \notin \text{RC}$ (\emptyset is the empty conjunction of stores).

The reject criterion RC is *correct* wrt \mathcal{D} (resp. \mathcal{T}) if $c \in \text{RC}$ implies $\models_{\mathcal{D}} \neg c$ (resp. $c \in \text{RC} \Rightarrow \mathcal{T} \models \neg c$).

From a reject criterion RC , we define a property of the set of skeletons, also denoted by RC , in the following way: a skeleton $s \in \text{RC}$ if $\text{store}(s) = \exists c$ and $c \in \text{RC}$ (c is a conjunction of basic constraints).

Remark.

1. A skeleton p , $p \in \Pi$, is not rejected.
2. If $s \in \text{RC}$ then for each skeleton s' such that $s \sqsubseteq s'$: $s' \in \text{RC}$.
3. If $s \in \text{RC}$ then for each skeleton s' such that s is a sub-term of s' : $s' \in \text{RC}$.

◇

4.4 Positive Computation (SLD derivation)

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

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 $\nu \in \text{undef}(s)$ and a clause name $f \in \delta_{\text{symbol}(s, \nu)}(P)$ such that s' is $s[\nu \leftarrow \text{sk}(f)]$ (grafting at the occurrence ν).

We say s' derives from s by the occurrence ν .

\hookrightarrow defines a transition system between (computation) states.

An *initial state* is a skeleton p , where $p \in \Pi$.

A state s is a *final state* when 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 p , $p \in \Pi$, is a (finite or infinite) sequence of states $s_1 s_2 \cdots s_i \cdots$ such that $s_1 = p$, for each $j = 2, \dots, i, \dots$: $s_{j-1} \hookrightarrow s_j$ 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.

4.5 Positive Answer

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

In our framework the basis of the result known as “independence of the computation rule” or “confluence” is the following lemma:

Lemma 7 s is a sk-positive answer if and only if s is a complete state.

We denote by $ans_P(p)$ the set of sk-positive answers of type $\rightarrow p$, $p \in \Pi$.

A member of $ans_P(p)$ is a sk-positive answer to p .

Definition 8 If s is a sk-positive answer to p then the formula proved by s is a positive answer formula to the goal $\forall(\leftarrow p(\tilde{x}))$.

Definition 9 A computation rule is a mapping r which provides an occurrence of $undef(s)$ 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 occurrence $r(s)$. We say s' derives from s according to the computation rule r . Obviously, we define the notions of sk-positive answer according to r and positive answer formula according to r .

Lemma 10 Sk-positive answers are independent of the computation rule.

4.6 Positive Partial Correctness (wrong positive answer)

Definition 11 A positive symptom is a positive answer formula $\forall(a \leftarrow c)$ which is not true in I .

Let $\forall(p(\tilde{x}) \leftarrow c)$ be a positive symptom. This formula is proved by a skeleton s .

Each sub-term of s is a sk-positive answer, thus each sub-term proves a positive answer formula.

Definition 12 A positive incorrectness is a name of clause f of arity n and n positive answer formula $\forall(a_1 \leftarrow c_1), \dots, \forall(a_n \leftarrow c_n)$ such that for each $i = 1, \dots, n$: $\forall(a_i \leftarrow c_i)$ is true in I but $\forall(a \leftarrow c \wedge \exists_{-a_1} c_1 \wedge \dots \wedge \exists_{-a_n} c_n)$ is not true in I , where $\text{clause}(P, f) = \forall(a \leftarrow c \wedge a_1 \wedge \dots \wedge a_n)$.

The clause of a positive incorrectness is not valid in I and the n positive answer formulae explain the reason of the incorrectness of the clause.

Lemma 13 If there exists a positive symptom then there exists a positive incorrectness.

4.7 Positive Diagnosis

The diagnosis consists in checking the formulae proved by the sub-terms of the original symptom sk-positive answer in order to find a sk-positive answer $s' = f(s_1, \dots, s_n)$ such that s' is a symptom but the s_i 's are not. Obviously f and the positive answer formulae proved by each s_i is a positive incorrectness.

Each strategy in order to locate a positive incorrectness can be considered, for example the top-down strategy [24, 6, 14, 19]. Usually the search terminates when the first incorrectness is found, so changing the strategy can optimize the number of queries. Strategies better than the top-down (or bottom-up) are for example the divide-and-query strategy [24, 1], or some heuristics based strategies [22, 20].

Finally, we have considered positive answer formula as positive symptom, but we can use also another definition of positive symptom: $\forall(a \leftarrow c \wedge c')$ not valid in I such that $\forall(a \leftarrow c)$ is a sk-positive answer. Then the algorithm is adapted and the notion of positive incorrectness founded is more precise (less general) (see previous papers [13, 25, 9, 8]).

4.8 Positive Partial Insufficiency (missing positive answer)

We do not develop this approach of diagnosis for missing answers here.

The general scheme, developed in [24, 7, 8], is based on co-induction and the positive semantics.

This method will be studied among the solutions proposed for the declarative diagnosis part of the DiSCiPl project.

We prefer to show in this deliverable another approach based on a *negative semantics* of the program and called *negative partial incorrectness*.

For a detailed comparison between positive partial insufficiency and negative partial incorrectness see [26].

5 Negative Side (Negative Diagnosis)

At the negative level, we consider formulae of the form $\forall(b \rightarrow c_1 \vee \dots \vee c_n)$ ($b \in BODY$, $\{c_1, \dots, c_n\} \subseteq STORE$).

5.1 Negative Computation (SLD tree)

Definition 14 Let r be a computation rule and $p \in \Pi$. 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 the types of s and s' are $\rightarrow p$.

Lemma 15 Let $p \in \Pi$ and r be a computation rule. \hookrightarrow_r^p has the property of a parent relation over the set $\{s \mid p \hookrightarrow_r^* s\}$ (where \hookrightarrow_r^* is the reflexive transitive closure of \hookrightarrow_r^p).

It is the *SLD tree* for p according to the computation rule r .

Remark. A branch of the SLD tree is a SLD derivation (according to r) for p . \diamond

Lemma 16 *The set of success leaves of a SLD tree for p , $p \in \Pi$, does not depend on the computation rule. Given a computation rule r , s is a sk-positive answer to p , $p \in \Pi$, if and only if s is a success leaf the SLD tree for p .*

5.2 Negative Answer

Definition 17 *A sk-negative answer to p , $p \in \Pi$, is $ans_P(p)$ if there exists a computation rule r such that the SLD tree for p according to r is finite.*

Remark. The sk-negative answer to p does not depend on r . \diamond

This is the “independence of the computation rule” of the negative computation.

Definition 18 *If $ans_P(p)$ is a sk-negative answer to p then, the negative answer formula to the goal $\forall(\leftarrow p(\tilde{x}))$ is $\forall(p(\tilde{x}) \rightarrow \bigvee\{store(s, \tilde{x}) \mid s \in ans_P(p)\})$.*

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

We have defined two levels of answers corresponding to two levels of computation: sk-positive answers (SLD derivations, positive computations) and sk-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 finite computation is the computation of exactly one formula.

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

Lemma 19 *Let \mathcal{T} be a constraint theory and \mathcal{D} be a model of \mathcal{T} such that RC is correct wrt \mathcal{T} . If $\forall(a \rightarrow c_1 \vee \dots \vee c_n)$ is a negative answer formula then:*

1. $\text{FI}(P) \models_{\mathcal{D}} \forall(a \rightarrow c_1 \vee \dots \vee c_n)$;
2. $\text{FI}(P), \mathcal{T} \models \forall(a \rightarrow c_1 \vee \dots \vee c_n)$.

The positive computations are the kind of computation which is usually considered, but we can note that for the operational semantics of Prolog IV [23] and Escher [15] the definition of answer is similar to our definition of negative answer formula.

5.3 Extensions

We define the set of extensions of a state s for a subset $g \subseteq \text{undef}(s)$ by:

$$\text{ans}_P(s, g) = \{s' \mid s \sqsubseteq s', \text{undef}(s) \setminus g = \text{undef}(s')\}$$

Remark. for each state s and each constant state $p \in \Pi$:

- $\text{ans}_P(s, \emptyset) = \{s\}$
- $\text{ans}_P(p, \text{undef}(s)) = \text{ans}_P(p)$

◇

The extensions we consider are not any. We consider extensions of pairs (s, g) where s is a state of the SLD tree and $g = \emptyset$ or $g = \{r(s)\}$ or $g = \{r(s)\} \cup \{\nu \mid \nu \in \text{undef}(s), \nu \text{ and } r(s) \text{ are siblings}\}$. In the following, each pair (s, g) has the previous property.

Lemma 20 *If the SLD tree for p according to r is without co-routining and finite then, for each pair (s, g) :*

1. *all the states of $ans_P(s, g)$ are in the SLD tree;*
2. *$ans_P(s, g)$ is finite.*

For each pair (s, g) we define the formula³:

$$\mathcal{F}_P(s, g) = \forall(\exists_{-var(A_g)}c \wedge A_g \rightarrow \bigvee_{s' \in ans_P(s, g)} \exists_{-var(A_g)}c_{s'})$$

where

- for each $s' \in ans_P(s, g)$: $\forall(a \leftarrow c_{s'} \wedge A)$ is the formula proved by s' ;
- $\forall(a \leftarrow c \wedge A \wedge A_g)$ is the formula proved by s ;
- A_g is the conjunction of atoms associated to g in s (A is the conjunction of atoms associated to $undef(s) \setminus g$ in s).

Remark. $\mathcal{F}_P(s, \emptyset) = \forall(store(s) \rightarrow store(s))$ is always valid. \diamond

5.4 Negative Proof tree

A SLD tree can be seen as a proof of the negative answer formula by considering that its sub-trees are proofs of formulae of the form $\forall(c \wedge a_1 \wedge \dots \wedge a_n \rightarrow c_1 \vee \dots \vee c_m)$ ⁴.

Whereas at the positive level the formulae computed by the systems have the form $\forall(c \wedge a_1 \wedge \dots \wedge a_n \leftarrow c')$ and can be obviously reduced to formulae of the form $\forall(a \leftarrow c'')$, the formulae computed at the negative level of the form $\forall(c \wedge a_1 \wedge \dots \wedge a_n \rightarrow c_1 \vee \dots \vee c_m)$ cannot reduce to something referring to only one atom on the left part of the implication. This seems to be a

³Intuitively, in the formula $\mathcal{F}_P(s, g) = \forall(b \rightarrow \bigvee_i c_i)$ the c_i 's are the computed answer to the goal b and b is the goal denoted by (s, g) (the store of s and the atoms associated to g).

⁴ $c \wedge a_1 \wedge \dots \wedge a_n$ is the current goal and c_1, \dots, c_m are the computed answers to this goal.

real problem because it appears very intricate to consider negative answers formula to intermediate goals which could be very large.

We consider other proofs of negative answer formulae.

A way to prove $\forall(c \wedge a_1 \wedge \cdots \wedge a_i \wedge \cdots \wedge a_n \rightarrow C)$ is to prove:

$$\forall(c \wedge a_i \rightarrow c_1 \vee \cdots \vee c_m)$$

and for each $j = 1, \dots, m$ to prove

$$\forall(c \wedge \exists_{-a_i} c_j \wedge a_1 \wedge \cdots \wedge a_{i-1} \wedge a_{i+1} \wedge \cdots \wedge a_n \rightarrow C_j)$$

where

$$C = \bigvee \{C_j \mid j = 1, \dots, m\}$$

In this way, we show that we can consider only formulae of the form $\forall(c \wedge a \rightarrow c_1 \vee \cdots \vee c_m)$. Unfortunately it requires the existence of an SLD tree without co-routining (see [9]).

A finite SLD tree for p according to r , without co-routining, is supposed to be given.

Let us consider the following set of rules⁵, for each pair (s, g) :

- if $g = \emptyset$

$$\frac{\emptyset}{\mathcal{F}_P(s, \emptyset)}$$

- if $g = \{r(s)\}$

$$\frac{\{\mathcal{F}_P(s', \text{undef}(s') \setminus \text{undef}(s)) \mid s \hookrightarrow_r s'\}}{\mathcal{F}_P(s, \{r(s)\})}$$

- if $g \neq \emptyset$ and $g \neq \{r(s)\}$

$$\frac{\{\mathcal{F}_P(s, \{r(s)\})\} \cup \bigcup \{\mathcal{F}_P(s', g \setminus \{r(s)\}) \mid s' \in \text{ans}_P(s, \{r(s)\})\}}{\mathcal{F}_P(s, g)}$$

Remark. To each pair (s, g) corresponds exactly one rule. \diamond

⁵The premises of the rules are multi-set (because a formula can have several occurrences), but we use the set notations for legibility.

Lemma 21 *To each pair (s, g) corresponds exactly one proof tree of the formula $\mathcal{F}_P(s, g)$.*

Lemma 22 *The proof tree which corresponds to $(p, \text{undef}(p))$ is rooted by the negative answer formula to the goal $\forall(\leftarrow p(\tilde{x}))$.*

Lemma 23 *If e is the formula at the root of a proof tree then $\text{FI}(P) \models_{\mathcal{D}} e$ ($\text{FI}(P), \mathcal{T} \models e$), where RC is correct wrt \mathcal{D} (wrt \mathcal{T}).*

It is possible to define a kind of skeleton for the proof trees whose function symbols would be the pair (s, g) , but this is useless here.

An important point is to note that some of the rules are tautologies. That is, if the premises hold then the conclusion holds. They are the rules with conclusion $\mathcal{F}_P(s, g)$, $g \neq \{r(s)\}$:

Lemma 24 *For each state s in the SLD tree:*

1. $\emptyset \models \mathcal{F}_P(s, \emptyset)$
2. for each pair (s, g) , $g \neq \emptyset$, $g \neq \{r(s)\}$:

$$\emptyset \models \mathcal{F}_P(s, \{r(s)\}) \wedge \bigwedge_{s' \in \text{ans}_P(s, \{r(s)\})} \mathcal{F}_P(s', g \setminus \{r(s)\}) \rightarrow \mathcal{F}_P(s, g)$$

5.5 Negative Partial Correctness (wrong negative answer)

Let us assume that the negative answer formula computed by the SLD tree is not valid in I .

Nevertheless the negative answer formula is proved by the proof tree.

For the time being we define a notion of negative symptom:

A *symptom* is a formula $\mathcal{F}_P(s, g)$ not valid in I , (where s is a state of the SLD tree and (s, g) is pair with the property describes before).

The proof tree is rooted by a symptom consequently a rule of the proof tree is not valid: its premises are not symptoms but its conclusion is a symptom.

We saw that the rules whose conclusion is $\mathcal{F}_P(s, g)$ with $g \neq \{r(s)\}$ are always valid, thus a minimal symptom has the form $\mathcal{F}_P(s, \{r(s)\})$.

Definition 25 A negative symptom is a formula $\mathcal{F}_P(s, \{r(s)\})$ not valid in I , where s is a state of the SLD tree.

We give some preliminary definitions in order to define the notion of error which corresponds to a minimal symptom.

Definition 26 A fact $\forall(p(\tilde{x}) \leftarrow c)$ is covered (by P) if:

1. for each $f \in \delta_p(P)$, let $\text{clause}(P, f) = \forall(p(\tilde{x}) \leftarrow c^f \wedge a_1^f \wedge \dots \wedge a_{n_f}^f)$
2. for each $f \in \delta_p(P)$, there exists n_f stores $c_1^f, \dots, c_{n_f}^f$ such that for each $i = 1, \dots, n_f$: $\forall(a_i^f \leftarrow c_i^f)$ is valid in I ;
3. for each $f \in \delta_p(P)$, $\forall(a \leftarrow \exists_{-a}(c^f \wedge \bigwedge_{i=1, \dots, n_f} c_i^f))$ is valid in I ;
4. $\forall(c \rightarrow \bigvee_{f \in \delta_p(P)} \exists_{-a}(c^f \wedge \bigwedge_{i=1, \dots, n_f} c_i^f))$ is true in the constraint interpretation \mathcal{D} .

Definition 27 A conjunction $c \wedge a$ is completely covered (by P) if, for each store c' such that $\forall(c' \rightarrow c \wedge a)$ is expected (i.e. $\forall(c' \rightarrow c)$ is valid in \mathcal{D} and $\forall(a \leftarrow c')$ is valid in I), $\forall(a \leftarrow c')$ is covered.

Note that the previous definition is more intricate than in pure logic programming because there is no more independence of negated constraints [16]. But, if each valuation v is the unique solution of a store c_v then $\forall(p(\tilde{x}) \leftarrow c_v)$ (in some sense $v(p(\tilde{x}))$) is covered if there exists a clause $\forall(p(\tilde{x}) \leftarrow b)$ of the definition of p such that v is a solution of the body b of the clause in \mathcal{I} . $c \wedge a$ is completely covered if for each valuation v solution of $c \wedge a$ in \mathcal{I} , $\forall(a \leftarrow c_v)$ is covered.

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

Definition 28 A negative incorrectness is a conjunction $c \wedge a$ which is not completely covered.

Lemma 29 *Let $\forall(c \wedge a \rightarrow C)$ be a minimal symptom then $c \wedge a$ is a negative incorrectness.*

Corollary 30 *If there exists a negative symptom then there exists a negative incorrectness.*

5.6 Negative Diagnosis

Let \prec be the relation over the set of nodes of the proof tree labeled by a formula of the form $\mathcal{F}_P(s, \{r(s)\})$ defined as follow: $N' \prec N$ if N' is a descendant of N (in the proof tree) and each node N'' descendant of N and ancestor of N' is labeled by a formula of the form $\mathcal{F}_P(s'', g'')$, $g'' \neq \{r(s'')\}$.

Remark. \prec is well-founded (finite) and it is a parent relation which defines a tree. ◇

The diagnosis consists in checking the nodes of the proof tree labeled by formulae $\mathcal{F}_P(s, \{r(s)\})$ in order to find a negative symptom such that all the predecessors according to \prec are not labeled by negative symptoms.

As for the positive case, each strategy can be considered: top-down, bottom-up, divide-and-query, etc.

The novel framework (negative proof trees and incorrectness diagnosis for missing answers) define a wide family of algorithms for negative diagnosis.

The algorithm proposed in [26] is a member of this family and is a lifting of the algorithm proposed in [5] 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 queries is allowed, not only the order of the strategy used to build the SLD tree because all the whole search space has been identified.

6 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 answers can be considered in the same framework. Already in LP, two insufficiency notions was defined: a *non covered atom* ([24, 7, 14]) and a *non completely covered atom* ([5]). For example, [26] 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 for the negative side 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] D. F. J. Binks. *Declarative Debugging in Gödel*. PhD thesis, University of Bristol, 1995.
- [2] P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming*. MIT Press, 1993.
- [3] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59:133–155, 1988.
- [4] W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Fifth Generation Computer Systems*, pages 573–581, 1988.
- [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. Fritzon, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 40–57. Springer-Verlag, 1993.
- [8] G. Ferrand and A. Tessier. Declarative Debugging. *The Newsletter of the European Network in Computational Logic*, 3(1):71–77, 1996. COMPULOG NET.
- [9] G. Ferrand and A. Tessier. Positive and Negative Diagnosis for Constraint Logic Programs in terms of Proof Skeletons. In M. Kamkar, editor, *International Workshop on Automated Debugging*, pages 141–154, 1997.
- [10] 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.

- [11] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *14th ACM Symposium on Principles of Programming Languages*, pages 111–119, 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] J. W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, 1995.
- [16] M. J. Maher. A Logic Programming view of CLP. In Warren, editor, *International Conference on Logic Programming*, pages 737–753. MIT Press, 1993.
- [17] S. Nadjm-Tehrani. Debugging Prolog Programs Declaratively. In *Workshop on Meta-programming in Logic*, pages 137–155, 1990.
- [18] L. Naish. Declarative Diagnosis of Missing Answers. *New Generation Computing*, 10(3):255–285, 1992.
- [19] L. Naish. A Declarative Debugging Scheme. Technical Report 95/1, Department of Computer Science, University of Melbourne, 1995.
- [20] A. E. Nicholson. Declarative Debugging of the Parallel Logic Programming Language GHC. In *Australian Computer Science Conference*, pages 225–236, 1988.
- [21] 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.
- [22] L. M. Pereira and M. Calejo. A Framework for Prolog Debugging. In R. A. Kowalski and K. A. Bowen, editors, *Joint International Conference and Symposium on Logic Programming*, pages 481–495. MIT Press, 1988.

- [23] PrologIA. *Le manuel de Prolog IV*, 1996.
- [24] E. Y. Shapiro. *Algorithmic Program Debugging*. ACM Distinguished Dissertation. MIT Press, 1982.
- [25] A. Tessier. *Approche, en termes de squelettes de preuve, de la sémantique et du diagnostic d'erreur des programmes logiques avec contraintes*. PhD thesis, LIFO, University of Orléans, 1996.
- [26] 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.

Index

- answer
 - negative formula, 20
 - positive formula, 17
 - sk-negative, 20
 - sk-positive, 17
- atom, 10
- body, 10
- clause, 10
 - name, 11
- computation
 - computation rule, 17
 - negative, 19
 - positive, 16
 - state, *see* state
- constraint, 10
 - basic, 10
 - store, *see* store
- covered, 25
 - completely, 25
- definition, 11
- derivation, *see* SLD
- derive, 16
- if-and-only-if-part, *see* program
- if-part, *see* program
- incorrectness
 - negative, 25
 - positive, 18
- negative
 - answer, *see* answer
 - computation, 19
 - incorrectness, 25
 - symptom, 25
- only-if-part, *see* program
- positive
 - answer, *see* answer
 - computation, 16
 - incorrectness, 18
 - symptom, 18
- program, 11
 - if-and-only-if-part, 12
 - if-part, 12
 - only-if-part, 12
- reject criterion, 15
 - correct, 16
- skeleton, 13, 14
 - associated store, 15
 - complete, 14
 - formula proved, 14
 - state, *see* state
- SLD
 - derivation, 17
 - failure, 17
 - success, 17
 - tree, 20
- state, 16, *see* skeleton
 - complete, *see* skeleton
 - failure, 16
 - final, 16
 - initial, 16
 - success, 16
 - transition system, 16
- store, 10
 - of a skeleton, 15
- symptom
 - negative, 25
 - positive, 18

Notations

$\forall e$, 10	$\text{IFF}(P)$, 12
$\exists e$, 11	$\text{lfp}(T)$, 8
$\exists_{\tilde{x}}e$, 11	P , 11
$\exists_{-\tilde{x}}e$, 11	Π , 10
$\exists_{-a}e$, 11	RC , 15
\hookrightarrow , 16	$\text{sk}(f)$, 15
\hookrightarrow_r , 17	STORE , 10
\hookrightarrow_r^p , 19	$\text{store}(s)$, 15
\sqsubseteq , 15	$\text{store}(s, \tilde{x})$, 15
$\delta(P)$, 11	Σ , 9
$\delta_p(P)$, 11	$\text{symbol}(s, \nu)$, 15
$s[\nu \leftarrow s]$, 16	\mathcal{T} , 14
$\text{ans}_P(p)$, 17	$\text{undef}(s)$, 15
$\text{ans}_P(s, g)$, 21	V , 9
ATOM , 10	$\text{var}(e)$, 10
BODY , 10	Ξ , 10
$\text{clause}(P, f)$, 11	
\mathcal{D} , 12	
$\text{def}(s)$, 15	
$\mathcal{F}_P(s, g)$, 22	
$\text{FI}(P)$, 12	
$\text{gfp}(T)$, 9	
$\text{IF}(P)$, 12	