

5. Declarative Diagnosis in the CLP scheme

Alexandre Tessier and Gérard Ferrand

LIFO, BP 6759, F-45067 Orléans Cedex 2, France
email: {Alexandre.Tessier, Gerard.Ferrand}@inria.fr

When a result is computed but it is considered as *incorrect* because it is *not expected*, we consider that we have a *symptom* (of error). The symptom may be a *wrong answer* or a *missing answer*. The role of *diagnosis* is to locate an *error*, that is a limited program fragment responsible for the symptom. The notions of *symptom* and *error* have a meaning only w.r.t. some notion of *expected semantics*. We consider only *declarative semantics*. The user does not need to understand the operational behaviour of the *CLP* system. Symptom and error are connected via some kind of tree and the diagnosis amounts to search for a kind of *minimal symptom* in this tree. Several search strategies are possible. The principles of an implementation are described, with a diagnosis session.

5.1 Introduction

To intuitively introduce the basic notions of *symptom* and *error* let us consider a toy program in the paradigm *clp(FD)*:

```
fac(N,F) :- B#=1, aux(N,B,F).  
aux(N,B,P) :- N#=0, B#=P.  
aux(N,B,P) :- N#=M+1, C#=N, aux(M,C,P).
```

with an *expected semantics* such as:

- *fac* is the *factorial function*, that is $fac(N, F) \Leftrightarrow F = N!$
- and $aux(N, B, P) \Leftrightarrow P = N! * B$.

So the last clause should be:

```
aux(N,B,P) :- N#=M+1, C#=N*B, aux(M,C,P).
```

For the *goal*: $N#\leq 2, fac(N,F)$ we have three *computed answer constraints* which are

```
F = 1, N = 0.  
F = 1, N = 1.  
F = 1, N = 2.
```

A goal is the description of a *relation*. Here the expected relation is: $F = 1, N = 0$ or $F = 1, N = 1$ or $F = 2, N = 2$. So this computed result is considered as *incorrect* because it is *not expected*. Let us consider that we have a *symptom* (of error). But there are two ways to understand that there is a symptom:

Firstly we have a *wrong answer*, $F = 1$, $N = 2$.

To be more formal, a way to express that it is a *wrong answer* is to say that, with respect to the *expected semantics* of the program, the following logic formula (of the form: *computed answer constraint* \Rightarrow *goal*) is *false*:

$$F = 1 \wedge N = 2 \Rightarrow N \leq 2 \wedge fac(N, F)$$

It is a first kind of symptom.

Secondly we have a *missing answer*, $F = 2$, $N = 2$.

Unlike the first kind of symptom, to be able to say that it is a *missing answer* we have to take *all the computed answer constraints* into account: It is missing because among these *computed answer constraints* we do not see $F = 2$, $N = 2$.

To be more formal, a way to express that it is a *missing answer* is to say that, with respect to the *expected semantics* of the program, the following logic formula is *false*:

$$N \leq 2 \wedge fac(N, F) \Rightarrow (F = 1 \wedge N = 0) \vee (F = 1 \wedge N = 1) \vee (F = 1 \wedge N = 2)$$

(since with respect to the *expected semantics*, for $F = 2$, $N = 2$, the left hand side is true but the right hand side is false). It is a second kind of symptom.

So we have *two* kinds of symptoms: for a *question* (that is a goal) it is possible to have *several* computed *answers* $C_1 ; \dots ; C_i ; \dots$ so we may be interested either in a single answer or in the sequence of all the answers. We consider that there are two levels of computation and two kinds of results. At each level the computation is *finite*: At a first level a result is a *single* C_i . If C_i is a *wrong answer* we have a symptom of the first kind. At a second level of computation it is the sequence $C_1 ; \dots ; C_i ; \dots$ which is the result. To be more precise, since it is a finite computation, it is a sequence: $C_1 ; \dots ; C_n$ (terminated by “no more” answer). *Finite failure* is the particular case where $n = 0$. If an expected answer C is *missing* among $C_1 ; \dots ; C_i ; \dots$ we have a symptom of the second kind. With these intuitive motivations we call *positive* the first level, where there is the first kind of symptom, and we call *negative* the second level, where there is the second kind of symptom ([5.8]). It is because we are in a *relational* paradigm that these two levels are shown to be so different.

From an intuitive viewpoint symptoms are “caused” by *errors*. Roughly speaking an error is a limited program fragment responsible for the symptom and the role of *diagnosis* is to locate the error.

For the *positive* level, in our example, the clause

$$aux(N, B, P) :- N\#=M+1, C\#=N, aux(M, C, P)$$

is erroneous and is responsible for the wrong answer $F = 1$, $N = 2$, that is to say for the *positive symptom* which is formalised by the (unexpected) formula

$$F = 1 \wedge N = 2 \Rightarrow N \leq 2 \wedge fac(N, F)$$

but it is possible to give more information about the “cause” of a symptom: In this small example it is easy to understand that the symptom, that is $fac(2, 1)$ which is false, comes from $aux(2, 1, 1)$, which comes from $aux(1, 2, 1)$, which comes from $aux(0, 1, 1)$. But $aux(0, 1, 1)$ is true whereas $aux(1, 2, 1)$ is false. This transition between true and false is through the erroneous clause *and the constraint* $N = 1, B = 2, P = 1, M = 0, C = 1$, since it is for these values that the body of the erroneous clause is true and its head is false. So the constraint gives more information about the “cause” of the symptom. We consider that it is the *pair* made of this erroneous clause and this constraint which is an *error (incorrectness)*, called *positive error (positive incorrectness)* because it is responsible for the positive symptom.

For the *negative* level the program fragment responsible for the symptom is a “packet of clauses” (all the clauses beginning with a same predicate symbol): the intuitive reason is that some answers are missing because some clause instances are missing. In our example, the “packet”

$$\begin{aligned} aux(N, B, P) & :- N \neq 0, B \neq P. \\ aux(N, B, P) & :- N \neq M + 1, C \neq N, aux(M, C, P). \end{aligned}$$

is erroneous and is responsible for the missing answer $F = 2, N = 2.$, that is to say for the *negative symptom* which is formalised by the (unexpected) formula

$$N \leq 2 \wedge fac(N, F) \Rightarrow (F = 1 \wedge N = 0) \vee (F = 1 \wedge N = 1) \vee (F = 1 \wedge N = 2)$$

but it is possible to give more information about the “cause” of a negative symptom, like for the positive level. In this small example it is easy to understand that there is again a transition between true and false through the erroneous “packet” *and a constraint*: With respect to the *expected semantics*, $aux(N, B, P)$ is true for $N = 1, B = 2, P = 2$ but it is not possible to have one of the two bodies: either $N = 0, B = P$ or $N = M + 1, C = N, aux(M, C, P)$ true for some values of M, C . A more formal way to express this is to say that for $N = 1, B = 2, P = 2$ the formula

$$\exists M \exists C (N = 0 \wedge B = P) \vee (N = M + 1 \wedge C = N \wedge aux(M, C, P))$$

is false w.r.t. the expected semantics. We consider again that it is the *pair* made of the erroneous “packet” and the constraint $N = 1, B = 2, P = 2$ which is an *error (incorrectness)*, called *negative error (negative incorrectness)* because it is responsible for the negative symptom.

The notions of *symptom* and *error* have a meaning only w.r.t. some notion of *expected semantics*. We consider only *declarative semantics* and we presuppose that, during a diagnosis session, the user is able to decide, for some computed answers, if they are *wrong*, or if some expected answer is

missing. In practice a computed answer may be intricate, it is the origin of the *presentation problem* (stressed by Lloyd [5.5]) which may be a difficulty of declarative diagnosis. However this presupposition is necessary to give a meaning to declarative debugging questions. From a conceptual viewpoint the user behaves like an *oracle* which is able to decide if something is wrong or missing (it is the same abstract notion of oracle and the same theoretical framework if we can partially replace the user by a system using some form of specification for the expected semantics of the program).

The notion of positive symptom and positive error comes from Shapiro's seminal work (wrong answer and incorrectness, [5.7]).

For the negative side, the notion of negative symptom and negative error does not correspond to Shapiro's notions (missing answer and insufficiency) which need more complex interaction with the oracle. These notions come from W. Drabent, S. Nadjm-Tehrani, J. Małuszyński (*non completely covered atom*, [5.2]).

The rest of the chapter is organised as follows. Section 5.2 defines the theoretical notions of symptom and error. Section 5.3 defines the proof-trees and the diagnosis scheme. Section 5.4 describes the diagnosis algorithms. Section 5.5 links the computation of a result which is a symptom with the proof-tree used in the diagnosis scheme. Section 5.6 describes an implementation. Section 5.7 shows a diagnosis session. Section 5.8 presents a conclusion and future work.

5.2 Basic Notions of Symptom and Error

We use the basic theoretical notions of the *CLP Scheme* ([5.4]). \mathcal{D} is a *constraint domain*. We consider only *definite programs*, that is to say without negation (thanks to disequations and global constraints, negation as failure is less useful than in classical Prolog). A *program* P is supposed to be *normalised* in such a way that only distinct variables are allowed as predicate arguments (all the links between variables are expressed by the constraints). It is a facility to simplify the explanation but it is not a loss of generality of the diagnosis method.

An *interpretation* I for the language of the program is supposed to be given. It is a formalisation of the *expected semantics* of the program. I must be an *expansion* of \mathcal{D} , that is to say that it adds to \mathcal{D} an interpretation for the new predicate symbols.

The following definitions are intuitively motivated by the previous introduction.

Definition 5.2.1. $C \rightarrow G$ (C constraint, G goal) is a computed positive symptom (of P w.r.t. I) if C is a computed answer for G , but $C \rightarrow G$ is false in I .

Definition 5.2.2. A positive error (positive incorrectness) (of P w.r.t. I) is a pair made up of a clause $p(\overline{X}) \leftarrow B$ in P and a constraint C such that, for some solution of C , B is true in I but $p(\overline{X})$ is false in I .

In some logic formulas we use the following *notation*: $\exists_{-A} F$ means quantification over the variables of F which have no occurrence in the expression A .

In the program P a “packet” of clauses

$$\begin{aligned} p(\overline{X}) &\leftarrow B_1 \\ \dots \\ p(\overline{X}) &\leftarrow B_m \end{aligned}$$

is the set of all the clauses of P beginning with a same p . The “packet” is also called the *definition* of the predicate p .

The *completed definition* of the predicate p is the formula

$$p(\overline{X}) \leftrightarrow \exists_{-p(\overline{X})} (B_1 \vee \dots \vee B_m)$$

It can be rewritten as

$$[p(\overline{X}) \leftarrow \exists_{-\overline{X}} (B_1 \vee \dots \vee B_m)] \wedge [p(\overline{X}) \rightarrow \exists_{-\overline{X}} (B_1 \vee \dots \vee B_m)]$$

$p(\overline{X}) \leftarrow \exists_{-\overline{X}} (B_1 \vee \dots \vee B_m)$ is merely equivalent to $p(\overline{X}) \leftarrow (B_1 \vee \dots \vee B_m)$, which is merely equivalent to the “packet” of p .

The other direction: $p(\overline{X}) \rightarrow \exists_{-\overline{X}} (B_1 \vee \dots \vee B_m)$ cannot be simplified so much but it is going to be a *useful notation*. $FI(P)$ (*only-if*(P)) is the set of the formulas

$$p(\overline{X}) \rightarrow \exists_{-\overline{X}} (B_1 \vee \dots \vee B_m)$$

Remark that we could also mention $IF(P)$, the set of the formulas

$$p(\overline{X}) \leftarrow \exists_{-\overline{X}} (B_1 \vee \dots \vee B_m)$$

but it is merely equivalent to P . Usually the set of the *completed definitions* is denoted by P^* , it is equivalent to $IF(P) \wedge FI(P)$.

Definition 5.2.3. $G \rightarrow \exists_{-G} (C_1 \vee \dots \vee C_n)$ (C_i constraints, G goal) is a computed negative symptom (of P w.r.t. I) if for the goal G there exists a finite SLD-tree whose computed answer constraints are $C_1 ; \dots ; C_n$, but $G \rightarrow \exists_{-G} (C_1 \vee \dots \vee C_n)$ is false in I .

Note that in the particular case $n = 0$ there is a *finite failure* and in such a case $(C_1 \vee \dots \vee C_n)$ is merely *false* so $G \rightarrow \exists_{-G} (C_1 \vee \dots \vee C_n)$ is $\neg G$.

Definition 5.2.4. A negative error (negative incorrectness) (of P w.r.t. I) is a pair made up of $p(\overline{X}) \rightarrow \exists_{-\overline{X}} (B_1 \vee \dots \vee B_m)$ in $FI(P)$ and a constraint C such that, for some solution of C , $p(\overline{X})$ is true in I but $\exists_{-\overline{X}} (B_1 \vee \dots \vee B_m)$ is false in I .

In the rest of this section we explain why, if there is a *computed positive* (resp. *negative*) *symptom* then there is a *positive* (resp. *negative*) *error*. It is a short and easy verification but this result is purely logical and non constructive. In the next section we set out a refinement of this result giving more information about the connection between symptom and error.

Positive level: At first we recall the basic *soundness* result ([5.4]): If C is a *computed answer constraint* for the goal G then $P \models_{\mathcal{D}} C \rightarrow G$, that is to say $C \rightarrow G$ is true in all the expansions of \mathcal{D} which are models of P . With regard to the question of the applicability to *CLP* systems with *incomplete* solvers, it is interesting to remark that, for this kind of soundness, neither correctness nor completeness is required for the solver (intuitively: if an incomplete solver does not reject C , if C is unsatisfiable then $C \rightarrow G$ is true. If an incorrect solver rejects C , C is not a computed answer, even if it is satisfiable).

Let $C \rightarrow G$ be a computed positive symptom. It is false in I so, thanks to the previous soundness result, I is not a model of P so there is in P some clause $p(\overline{X}) \leftarrow B$ which is false in I , so for some *assignment* a in \mathcal{D} , (namely for some values in \mathcal{D}), B is true in I but $p(\overline{X})$ is false in I . Such a could give some information about the “cause” of symptoms. To have a positive error it is sufficient to take a C such as a is solution of C . There is always such a C , e.g. *true*. However the more precise C is, the more informative it is.

Negative level: Similar explanation, using another basic *soundness* result: If for the goal G there is a *finite SLD-tree* whose *computed answer constraints* are $C_1 ; \dots ; C_n$, then $FI(P) \models_{\mathcal{D}} G \rightarrow \exists_{-G}(C_1 \vee \dots \vee C_n)$.

It is interesting to remark that now, for this kind of soundness, correctness (but not completeness) is required for the solver (intuitively: a satisfiable C_i cannot be removed from the conclusion of the implication, but an unsatisfiable C_i can be added).

5.3 Connection between Symptom and Error via Proof-Trees

Now we consider a notion of symptom which is more general than a computed symptom. Intuitively in these symptoms the constraint may be more informative than in a computed symptom.

Definition 5.3.1. $C' \wedge C \rightarrow G$ (C, C' constraints, G goal) is a positive symptom (of P w.r.t. I) if C is a computed answer for G , but $C' \wedge C \rightarrow G$ is false in I .

So if $C' = \text{true}$ the symptom is a computed symptom. Likewise:

Definition 5.3.2. $C' \wedge G \rightarrow \exists_{-G}(C_1 \vee \dots \vee C_n)$ (C', C_i constraints, G goal) is a negative symptom (of P w.r.t. I) if for the goal G there exists a finite

SLD-tree whose computed answer constraints are $C_1 ; \dots ; C_n$, but $C' \wedge G \rightarrow \exists_{-G}(C_1 \vee \dots \vee C_n)$ is false in I .

At each level of computation, we can get a logical representation of the computation as a *tree*. This tree is a *proof-tree* according to some *rules* as usual in logical formalisms. The rules and the proof-trees are not the same for the two levels of computation but the diagnosis scheme is the same and it is easy to explain it because at each level the diagnosis amounts to search for a kind of *minimal symptom* in this tree.

To clearly explain what are the rules and the proof-trees we use a new toy theoretical example: The program is

$$\begin{aligned} p(X) &\leftarrow q(X), r(X). \\ q(X) &\leftarrow X > 0. \\ q(X) &\leftarrow X < 0. \\ r(X) &\leftarrow X < 1. \end{aligned}$$

and the goal is $p(X)$.

At the *positive* level a computation is a *SLD-derivation*. In our example for the goal $p(X)$ there is a *SLD-derivation* giving the computed answer $X > 0 \wedge X < 1$. We can consider this computation as a proof of the formula $X > 0 \wedge X < 1 \rightarrow p(X)$ and to be more precise we can consider the computation as the construction of the following tree:

$$\frac{\frac{\frac{X > 0 \rightarrow X > 0}{X > 0 \rightarrow q(X)}}{X > 0 \wedge X < 1 \rightarrow q(X) \wedge r(X)}}{X > 0 \wedge X < 1 \rightarrow p(X)} \quad \frac{\frac{\frac{X < 1 \rightarrow X < 1}{X < 1 \rightarrow r(X)}}{X > 0 \wedge X < 1 \rightarrow X > 0 \wedge r(X)}}{X > 0 \wedge X < 1 \rightarrow p(X)}$$

Such a tree is made of *rules* and is called a *proof-tree* according to these rules. There are two kinds of rules:

- For each clause $p(\bar{X}) \leftarrow B$ in P , a “program” rule

$$\frac{C' \rightarrow B}{C \wedge C' \rightarrow C \wedge p(\bar{X})}$$

- and the “logical” rules. In our framework it is convenient to consider *all* the rules

$$\frac{(C_1 \rightarrow G_1), \dots, (C_n \rightarrow G_n)}{C \rightarrow G}$$

where $C \rightarrow G$ is a *logical consequence* of $(C_1 \rightarrow G_1), \dots, (C_n \rightarrow G_n)$. These rules are called *logical rules*. For example we can get the previous proof-tree with the two following rules:

$$\frac{C \rightarrow G \quad C' \rightarrow C \wedge G'}{C' \rightarrow G \wedge G'}$$

$$\overline{C \rightarrow C}$$

But from the same computation giving the same answer we can also extract another proof-tree:

$$\frac{\frac{X > 0 \wedge X < 1 \rightarrow X > 0}{X > 0 \wedge X < 1 \rightarrow q(X)} \quad \frac{X > 0 \wedge X < 1 \rightarrow X < 1}{X > 0 \wedge X < 1 \rightarrow r(X)}}{X > 0 \wedge X < 1 \rightarrow q(X) \wedge r(X)} \\ \overline{X > 0 \wedge X < 1 \rightarrow p(X)}$$

We can get this second proof-tree with the “program rule” and the following “logical” rules:

$$\frac{C \rightarrow G_1 \cdots C \rightarrow G_n}{C \rightarrow G_1 \wedge \cdots \wedge G_n}$$

$$\overline{C \wedge C_1 \wedge \cdots \wedge C_n \rightarrow C}$$

So for *each computed answer constraint* C for a goal G , the formula $C \rightarrow G$ is the root of various proof-trees, called (*positive*) *proof-trees*, according to these various rules, and each of these proof-trees can be easily obtained from the computation namely from the corresponding *SLD-derivation*.

In particular if $C \rightarrow G$ is a *computed (positive) symptom* then it is the root of various (*positive*) *proof-trees*. Let us consider such a proof-tree. Each node of this proof-tree is labelled by a $C' \rightarrow G'$. It may be a *symptom node* (the root is a symptom node). Let us consider the notion of *minimal* symptom node where “minimal” is defined w.r.t the binary relation: x *child of* y . So a node is a minimal symptom node if it is a symptom node but no child of it is a symptom node.

To each node of the proof-tree there is a rule which is associated (the label $C \rightarrow G$ of the node is the conclusion of the rule). Clearly in a “logical” rule, if the hypotheses are not symptoms then the conclusion is not a symptom. So the rule which is associated to a *minimal* symptom node cannot be a “logical” rule so necessarily it is a “program” rule

$$\frac{C' \rightarrow B}{C \wedge C' \rightarrow C \wedge p(\overline{X})}$$

Moreover, in this program rule, for some solution of $C \wedge C'$, $p(\overline{X})$ is false in I (since $C \wedge C' \rightarrow C \wedge p(\overline{X})$ is a symptom), but B is true in I (since $C' \rightarrow B$ is not a symptom), so the clause $p(\overline{X}) \leftarrow B$ and the constraint $C \wedge C'$ of this program rule give a *positive error*.

Clearly there are always minimal symptoms (since proof-trees are finite) and *any way* to find a *minimal symptom* in a *positive proof-tree* gives the localisation of a *positive error*. Moreover, to find a *minimal symptom* the following method is sufficient: To consider only conclusions of program rules (nodes of the form $C \wedge C' \rightarrow C \wedge p(\bar{X})$), and to search for *minimal symptoms with respect to these nodes*. So we have to test formulas $C \wedge C' \rightarrow C \wedge p(\bar{X})$ for symptoms. But such a test amounts to querying (the oracle) whether $C \wedge C' \rightarrow p(\bar{X})$ is *expected*, namely is *true* in I .

At the *negative* level a computation is a *finite SLD-tree*. In our example for the goal $p(X)$ there is a *finite SLD-tree* giving the computed answers $(X > 0 \wedge X < 1)$ and $(X < 0 \wedge X < 1)$. We can consider this computation as a proof of the formula $p(X) \rightarrow (X > 0 \wedge X < 1) \vee (X < 0 \wedge X < 1)$ and to be more precise we can consider the computation as the construction of the following tree:

$$\frac{\frac{\frac{X > 0 \rightarrow X > 0}{q(X) \rightarrow X > 0 \vee X < 0} \quad \frac{X < 0 \rightarrow X < 0}{X < 0 \wedge r(X) \rightarrow X < 0 \wedge X < 1}}{\frac{X < 1 \rightarrow X < 1}{X < 0 \wedge r(X) \rightarrow X < 0 \wedge X < 1}} \quad \frac{X < 1 \rightarrow X < 1}{X < 0 \wedge r(X) \rightarrow X < 0 \wedge X < 1}}{\frac{q(X) \wedge r(X) \rightarrow (X > 0 \wedge X < 1) \vee (X < 0 \wedge X < 1)}{p(X) \rightarrow (X > 0 \wedge X < 1) \vee (X < 0 \wedge X < 1)}}$$

Such a tree is made of *rules* and is called a *proof-tree* according to these rules. There are two kinds of rules (with, as usual, appropriate conditions on the free variables, which are not detailed here):

- For each $p(\bar{X}) \rightarrow \exists_{-\bar{X}}(B_1 \vee \dots \vee B_m)$ in $FI(P)$, a “program” rule

$$\frac{\text{for } i : B_i \rightarrow \exists_{-B_i} \bigvee_j C_j^i}{C \wedge p(\bar{X}) \rightarrow \exists_{-C p(\bar{X})} \bigvee_i \bigvee_j C \wedge C_j^i}$$

- and “logical” rules, for example

$$\frac{G \rightarrow \exists_{-G} \bigvee_i C_i \quad \text{for } i : C_i \wedge G' \rightarrow \exists_{-C_i G'} \bigvee_j C_j^i}{G \wedge G' \rightarrow \exists_{-GG'} \bigvee_i \bigvee_j C_j^i}$$

$$\frac{}{C \rightarrow C}$$

So if for a goal G there is a *finite SLD-tree* whose *computed answer constraints* are $C_1 ; \dots ; C_n$, then the formula $G \rightarrow \exists_{-G}(C_1 \vee \dots \vee C_n)$ is the root of various proof-trees, called (*negative*) *proof-trees*, according to these rules, and each of these proof-trees can be easily obtained from the computation namely from the corresponding *SLD-tree* ([5.3], [5.6]).

In particular if $G \rightarrow \exists_{-G}(C_1 \vee \dots \vee C_n)$ is a *computed (negative) symptom* then it is the root of various (*negative*) *proof-trees*. Let us consider such a proof-tree, in which each node is labelled by a $G' \rightarrow \exists_{-G'}(C'_1 \vee \dots \vee C'_n)$. A node may be a *symptom node* (the root is a symptom node). Let us consider the notion of *minimal symptom node* where “minimal” is defined w.r.t the

binary relation: x child of y . So a node is a minimal symptom node if it is a symptom node but no child of it is a symptom node.

To each node of the proof-tree there is a rule which is associated (the label $G \rightarrow \exists_{-G}(C_1 \vee \dots \vee C_n)$ of the node is the conclusion of the rule). Clearly in a “logical” rule, if the hypotheses are not symptoms then the conclusion is not a symptom. So the rule which is associated to a *minimal* symptom node cannot be a “logical” rule so necessarily it is a “program” rule

$$\frac{\text{for } i : B_i \rightarrow \exists_{-B_i} \bigvee_j C_j^i}{C \wedge p(\overline{X}) \rightarrow \exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C \wedge C_j^i}$$

Moreover in this program rule, for some solution of C , since $C \wedge p(\overline{X}) \rightarrow \exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C \wedge C_j^i$ is a symptom, $p(\overline{X})$ is true in I but $\exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C \wedge C_j^i$ is false in I . But the $B_i \rightarrow \exists_{-B_i} \bigvee_j C_j^i$ are not symptoms. Let us suppose $\exists_{-\overline{X}}(B_1 \vee \dots \vee B_m)$ is true in I , then, for some i , $\exists_{-\overline{X}} B_i$ is true, so $\exists_{-B_i} \bigvee_j C_j^i$ is true, so $\exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C \wedge C_j^i$ is true which is a contradiction. So $\exists_{-\overline{X}}(B_1 \vee \dots \vee B_m)$ is false in I . So $p(\overline{X}) \rightarrow \exists_{-\overline{X}}(B_1 \vee \dots \vee B_m)$ and the constraint C of this program rule give a *negative error*.

Clearly there are always minimal symptoms (since proof-trees are finite) and *any way* to find a *minimal symptom* in a *negative proof-tree* gives the localisation of a *negative error*. Moreover, to find a *minimal symptom* the following method is sufficient: To consider only conclusions of program rules (nodes of the form $C \wedge p(\overline{X}) \rightarrow \exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C \wedge C_j^i$), and to search for *minimal symptoms with respect to these nodes*. Such a minimal symptom can be found by testing if some $C \wedge p(\overline{X}) \rightarrow \exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C \wedge C_j^i$ are symptoms. But this test amounts to querying (the oracle) whether $C \wedge p(\overline{X}) \rightarrow \exists_{-Cp(\overline{X})} \bigvee_i \bigvee_j C_j^i$ is *expected*, namely is *true* in I .

5.4 Diagnosis Algorithm

So for each level of computation we have a notion of proof-tree. And any way to find a *minimal symptom* in a proof-tree gives the localisation of an *error*. Let us consider a proof-tree rooted by a (computed) symptom. The diagnoser queries the oracle about the labels of the nodes of the proof-tree, it is a test: “is this a symptom?”. The oracle does not need to understand the operational behaviour of the system.

The objective of the diagnoser is to locate a minimal symptom (a minimal symptom node). With this end in view the diagnoser uses a strategy in order to choose the node which corresponds to the next query to the oracle.

Let us assume that each node of the proof-tree can be either *expected* (it is not a symptom) or *unexpected* (it is a symptom) or *unknown* (it is not yet determined). The diagnosis algorithm is the following:

while no minimal symptom appears in the proof tree

choose an <i>unknown</i> node according to some strategy	
query the oracle about the chosen node in order to determine if it is	
<i>expected</i> or <i>unexpected</i>	

show the error associated with a minimal symptom

It is easy to show that this algorithm is correct and that it is complete in the sense that it always finds an error (the proof-tree is finite).

In general the oracle is the user and we have to take into account the fact that the user cannot answer some queries of the diagnoser. Thus we have a fourth category of nodes: the *dontask* nodes (the oracle cannot determine if it is a symptom). It is trivial to see that this possibility involves that the diagnoser is not always able to locate a minimal symptom. Moreover it involves that the strategies used to choose a node are more intricate and sometimes cannot choose a node.

As an example, we describe here the strategies *Top-Down* and *Divide&Query* which are the most useful in practice.

The Top-Down strategy follows a path of *unexpected* nodes from the root until it reaches an *unknown* node. This *unknown* node is the chosen node. Because of the *dontask* nodes, this strategy does not always find a node to query (see Fig 5.7). It looks like a prefix traversal of the proof-tree where sub-proof-trees rooted by an *expected* or *dontask* node are removed (see Fig. 5.1).

The relation $topdown(Node, ChosenNode)$ provides the node chosen ($ChosenNode$) by the Top-Down strategy in the tree rooted by $Node$. If the strategy cannot conclude $ChosenNode = notfound$.

$$\frac{topdown(Node, ChosenNode)}{\begin{array}{|l} \text{if } Node \text{ is } unknown \\ \quad \text{then } Node = ChosenNode \\ \quad \text{else if } Node \text{ is } unexpected \\ \quad \quad \text{then } topdown'(children(Node), ChosenNode) \\ \quad \quad \text{else } ChosenNode = notfound \end{array}}$$

$$\frac{topdown'([], notfound) \quad topdown'([Node|Children], ChosenNode)}{topdown(Node, ChosenNode') \quad \begin{array}{|l} \text{if } ChosenNode' = notfound \\ \quad \text{then } topdown'(Children, ChosenNode) \\ \quad \text{else } ChosenNode = ChosenNode' \end{array}}$$

Fig. 5.1 shows an example of proof-tree and the node chosen by the Top-Down strategy. The proof-tree is drawn as usual: the root of the proof-tree is at the top of the drawing, the parent relation of the proof-tree links a node which is the conclusion of a rule with its hypotheses which are its children on the drawing. In the drawing the following notation are used: *unexpected*

nodes are labelled by “N”, *expected* nodes by “Y”, *dontask* nodes by “X” and *unknown* nodes by “?”).

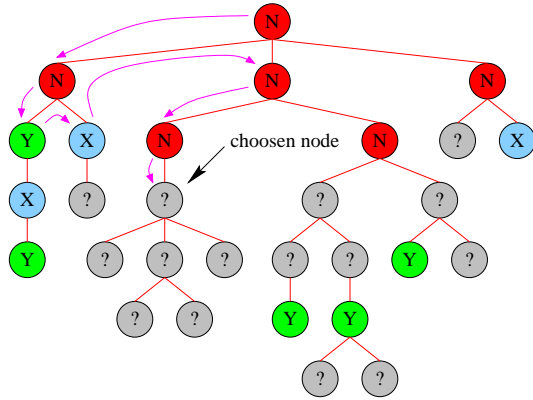


Fig. 5.1. Top-Down Strategy

The basic principle of the Divide&Query strategy is to choose a node in order to divide the search space in two parts:

- if the node is *expected*, the subtree rooted by this node can be removed from the search space,
- if the node is *unexpected*, each node which is not a descendant of this node can be removed from the search space.

The point is to choose a node such that there is as much *unknown* nodes in its subtree as out of its subtree (considering that all subtrees rooted by an *expected* node are removed from the search space). Note that it is not always possible to have as much *unknown* nodes in the subtree as out of it, but the strategy chooses the nodes which is the most near this condition.

The Divide&Query strategy here is an improvement of the Divide-and-query strategy of [5.7]. It really chooses the best node (that is the node which better divides the search space in two parts) and it takes into account *dontask* nodes and the fact that the strategy may be changed during a diagnosis session.

The tree considered is the tree rooted by an *unexpected* node with the least number of *unknown* and *dontask* nodes, but where a minimal symptom is always possible (remember the *dontask* nodes).

In Fig. 5.2 we use the same notations as in the previous example. In addition, the integer at the left of the nodes is the number of *unknown* or *dontask* nodes in their subtrees. The 1st subtree and the 4th subtree are not considered because of the *dontask* nodes: if the *unknown* nodes are *expected* no minimal symptom can be detected. The 3rd subtree is considered because the number of *unknown* nodes is lower than in the 2nd subtree. The 3rd subtree has 5 *unknown* nodes so the Divide&Query strategy chooses the root of the

The user invokes the diagnoser when a (positive or negative) symptom appears at the end of a (positive or negative) computation. The computation is either a SLD-derivation (a branch of a SLD-tree) or a SLD-tree. But the diagnoser uses a proof-tree!

The point is that it is possible to compute directly a (positive or negative) proof-tree rooted by the (positive or negative) computed symptom from a (positive or negative) computation, that is from (a branch of) a search-tree.

In order to simplify, the computation rule is assumed to be without co-routining (for instance the standard computation rule of Prolog), then proof-trees can be deduced from search-tree using the notion of *erasing* defined below. [5.6] shows an extension to any computation rule.

The main useful notion is: let x and y be two nodes of the search-tree, y is a node where x is *erased* if

- A_i is the atom selected at the node x in its goal (the goal on x is $A_1, \dots, A_i, \dots, A_n$ and the store is C),
- y is a descendants of x where A_i is fully solved (the goal on y is $A_1, \dots, A_{i-1}, A_{i+1}, \dots, A_n$ and the store is $C \wedge C'$ where C' is a computed answer to A_i).

Let x be a node of the search-tree, we denote by

- $store(x)$ the set of constraints accumulated from the root until the node x ,
- $select(x)$ the atom selected at the node x in its goal,
- $erased(x)$ the set of nodes where x is erased,
- $stchildren(x)$ the set of children of x in the search-tree.

Let C be a computed answer constraint for the goal G such that $C \rightarrow G$ is a positive symptom. C is made of the constraints accumulated along a success branch of the search-tree. The set of nodes of this branch is denoted by *branch*.

In order to define the positive proof tree which corresponds to the positive symptom computed, we have to determine:

1. the root of the proof-tree;
2. the binary relation: x *child of* y in the proof-tree;
3. the formula (query) which labels a node of the proof-tree.

We consider, in order to simplify, that the nodes of the proof-tree are a subset of the nodes of the search-tree, but the labels of a node is different depending on whether it is considered as a node of the proof-tree or a node of the search-tree.

First, the root of the positive proof-tree is the root of the search-tree.

Secondly, the list L of children of a node x of the positive proof-tree is given by the relation *+children*:

$$\frac{+children(x, L)}{\left| \begin{array}{l} \{y\} = stchildren(x) \cap branch \\ \text{if } y \in erased(x) \\ \quad \left| \begin{array}{l} \text{then } L = \square \\ \text{else } +children'(x, y, L1), L = [y|L1] \end{array} \right. \end{array} \right.}$$

$$\frac{+children'(x, y, L)}{\left| \begin{array}{l} \{z\} = erased(y) \cap branch \\ \text{if } z \in erased(x) \\ \quad \left| \begin{array}{l} \text{then } L = \square \\ \text{else } +children'(x, z, L1), L = [z|L1] \end{array} \right. \end{array} \right.}$$

Finally, the formula associated with a node x of the positive proof-tree is $store(y) \rightarrow select(x)$ where y is the success leaf of $branch$. We use the constraint $store(y)$ in the formula because it is the most precise we can know, but as said in Section 5.3 there exists various proof-trees.

For example, let us consider the small program (without constraints and variables to be more concise, in other words the constraints are always *true*):

```

p ← q, r.
p ← q, a.
r ← a.
q ← w.
w.
a ← z.
a.

```

The positive proof-tree which corresponds to the first answer to the goal p is given by Fig. 5.3. The nodes of the search-tree used by the proof-tree have been duplicated for the legibility of the drawing. The selected atom in the goals is underlined in the search-tree. For example, p has two children in the positive proof-tree: its child q in the search-tree, the node r where q is erased. Note that the node \square where r is erased is also the node where p is erased.

On Fig. 5.3 you can recognise a more classical notion of proof-tree in logic programming (a node corresponds to the head of a clause of the program and its children correspond to the body of the clause).

Let C_1, \dots, C_n be the computed answer constraints for the goal G such that $G \rightarrow \exists_{-G}(C_1 \vee \dots \vee C_n)$ is a negative symptom.

In order to define the negative proof tree which corresponds to the negative symptom computed, we have to determine:

1. the root of the proof-tree;
2. the binary relation: x *child of* y in the proof-tree;
3. the formula (query) which labels a node of the proof-tree.

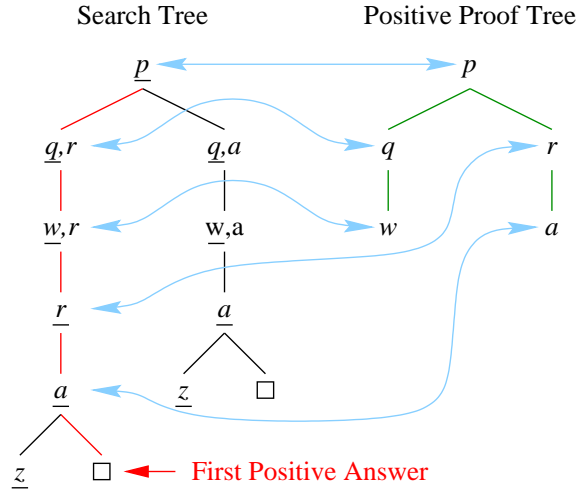


Fig. 5.3. From a branch of the search-tree to the positive proof-tree.

First, the root of the negative proof-tree is the root of the search-tree.

Secondly, the list L of children of a node x of the negative proof-tree is given by the relation $-children$:

$$\frac{-children(x, L)}{\text{let } S \text{ be the list of nodes of } (stchildren(x) \setminus erased(x))}{-children'(x, S, L)}$$

$$\frac{-children'(x, [], [])}{-children'(x, [y|S], [y|L])} \quad \frac{-children''(x, y, L1)}{-children'(x, S, L2)} \quad \text{append}(L1, L2, L)$$

$$\frac{-children''(x, y, L)}{\text{let } S \text{ be the list of nodes of } (erased(y) \setminus erased(x))}{-children'(x, S, L)}$$

Finally, the formula associated with a node x of the negative proof-tree is $store(x) \wedge select(x) \rightarrow \bigvee_{y \in erased(x)} store(y)$.

For example, let us consider again the previous program:

$p \leftarrow q, r.$
 $p \leftarrow q, a.$
 $r \leftarrow a.$
 $q \leftarrow w.$
 $w.$
 $a \leftarrow z.$
 $a.$

The negative proof-tree which corresponds to the search-tree for the goal p is given by Fig. 5.4. The nodes of the search-tree used by the proof-tree have been duplicated for the legibility of the drawing. The selected atom in the goals is underlined in the search-tree. For example, p has four children in the negative proof-tree: its children q and q in the search-tree, the node r where the first q is erased, the node a where the second q is erased. Note that the nodes where r and a are erased are also the nodes where p is erased.

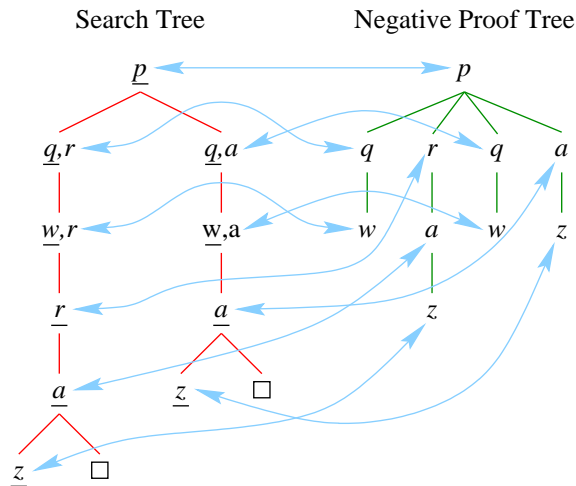


Fig. 5.4. From the search-tree to the negative proof-tree.

5.6 Implementation

The positive part of the declarative diagnoser has been implemented and tested on the INRIA platform: TkCalypso, developed in the DiSCiPl project. At the time of writing this book, implementation of the negative part is in progress.

TkCalypso is an extension of GNU-Prolog [5.1]. It includes a graphical interface (Fig. 5.5) and some debugger modules. Each module can be plugged or unplugged. Fig. 5.6 shows the structure of TkCalypso with the three modules that are actually implemented: search-tree visualisation, static debugger and declarative diagnoser. Communications between GNU-Prolog and the graphical interface are handled by the “Core/Gestionary” packages. This section describes the main features of the module called “Declarative Diagnoser”.

When a goal is given to TkCalypso, it stores informations on the search-tree in order to recompute it efficiently and make post-mortem analysis of the search-tree.

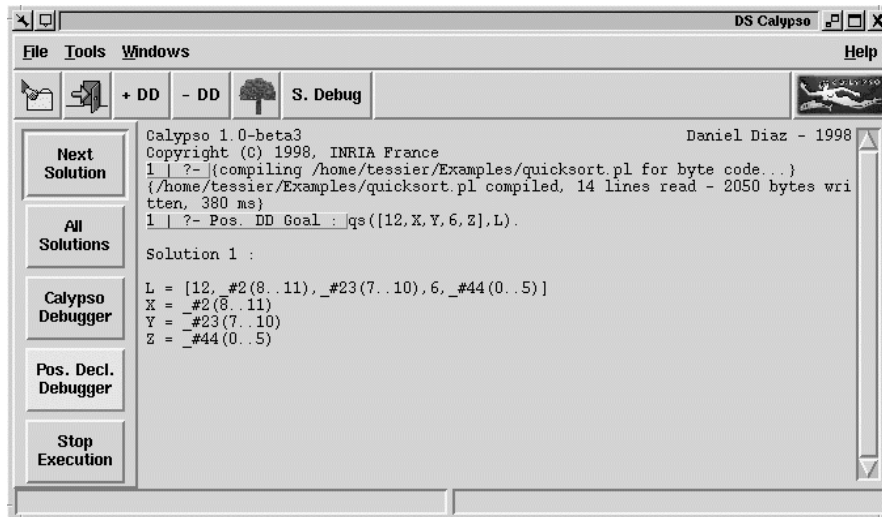


Fig. 5.5. Graphical interface: a positive symptom

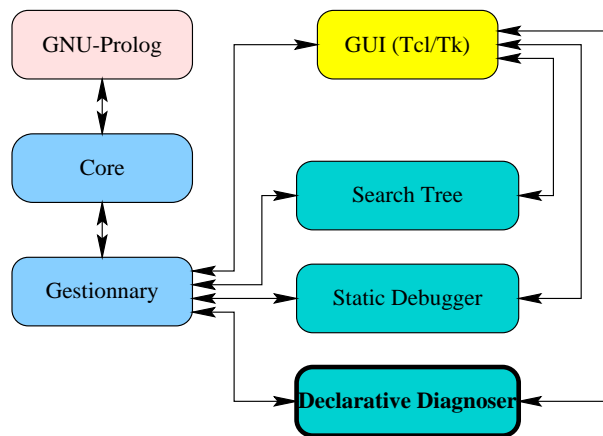


Fig. 5.6. Architecture of the TkCalypso platform.

If the user notices a computed symptom, then the declarative diagnoser is called and the parent relation of the (positive or negative) proof-tree is computed dynamically from the search-tree.

Once we have a (positive or negative) proof-tree, the diagnosis principle is always the same: choose a node of the proof-tree (according to some strategy), check if it is expected or not expected, until a minimal symptom is founded.

Several strategies have been implemented in order to choose the node to query: Top-Down, Bottom-Up, Divide&Query, Nearby-Error and User-Guided (Top-Down and Divide&Query are described in Section 5.4).

Built-in predicates are known as correct predicates, but also user predicates could be known as correct. For example, when the static analysis (see Chapter 2) has proved their correctness. Another example is when the user is convinced that some predicates are correct. So the user can set a list of correct predicates, which will be used by the diagnoser.

It is possible that the user does not want to be questioned on some predicates. For example, the semantics of the predicate is very intricate and the user want to suspend the queries on that predicate as long as possible. So the user can set a list of predicates which must not be questioned.

The status of a node of the proof-tree is more precise than the ones describe in Section 5.4, because we want to know the origin of this status, it can be:

- *unknown* when the predicate associated to the node is not a built-in and the node has not been queried,
- *expected(user)* when the user said that the node is expected,
- *expected(list)* when the predicates associated to the node is in the list of correct predicates (when the user is convinced that the predicate is correct),
- *expected(system)* when the predicate associated to the node is a built-in predicate (they are not suspected!),
- *unexpected(user)* when the user said that the node is not expected,
- *dontask(user)* when the user does not want to answer the query associated to the node (for example the query is very intricate), note that the user can come back later to the query associated with this node,
- *dontask(list)* when the predicates associated to the node is in the list of predicates that the user does not want to answer (for example the user does not know the semantics of the predicates or the user wants to delay the queries about the predicate).

The user has the possibility to dynamically add some predicates to the list of correct predicates or remove some predicates from it. If the user adds a predicate, each *unknown* or *dontask(-)* node concerning the predicate is labelled by *expected(list)*; if a node is *unexpected(user)* then the problem is given to the user: either remove the predicate from the correct predicate list or label the node as *expected(user)*. When the user removes a predicate of the list, each *expected(list)* node concerning the predicate are labelled by *unknown*.

The user can also change dynamically the list of predicates which must not be questioned. If the user adds a predicate to the list, each *unknown* node concerning the predicate becomes a *dontask(list)* node. If the user removes a predicate of the list, each *dontask(list)* node concerning the predicate becomes an *unknown* node.

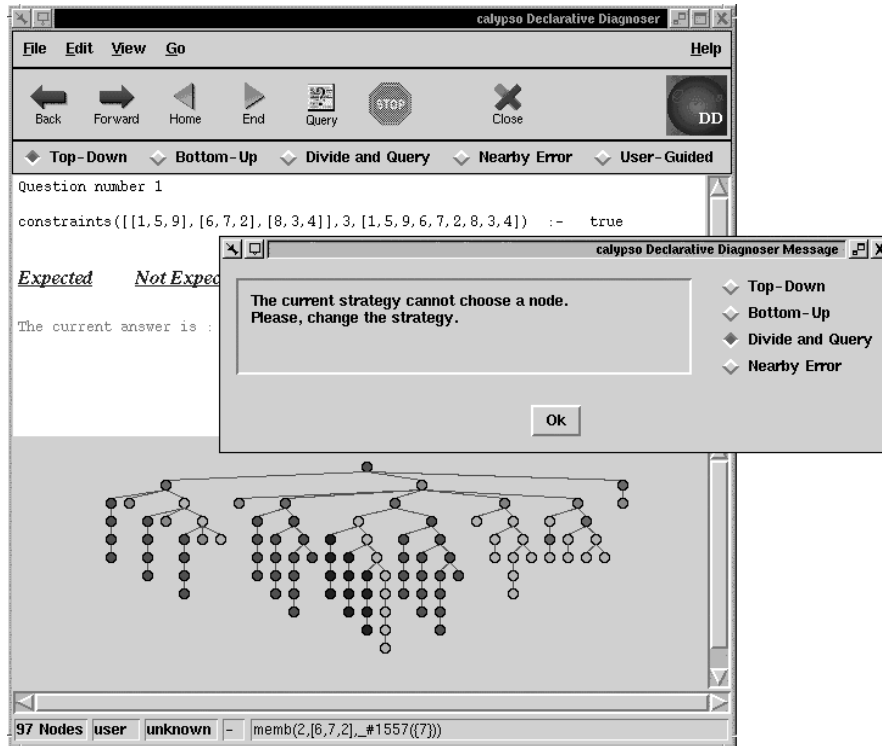


Fig. 5.7. Declarative Diagnoser Interface

The graphical interface (see Fig.5.7) of the diagnoser works like an hyper-text navigator: the user can navigate between queries, because its answers are seen as hyper-link between queries (it is possible to go back, go forward, see an history...). The proof-tree is displayed with informations about the nodes. For example, the user can see the status of a node

- with colours: red = *unexpected*, grey = *unknown*, blue = *dontask*, green = *expected*,
- and with shades: light = *system*, medium = *user*, dark = *list*.

The user can also choose the query with the mouse on the proof tree: it is the “User Guided” strategy.

The query $C \rightarrow A$ is displayed as $A :- \exists_{-A} C$. The diagnoser uses a trivial store simplification in order to simplify the constraint $\exists_{-A} C$ in the query. But this point has to be improved (this is discussed later in Section 5.8).

5.7 A Diagnosis Session

Let us consider the following QuickSort program which is intended to sort a list of finite domain terms:

```
qs([], []).
qs([Pivot | List], SortList) :-
    partition(Pivot, List, MinList, MaxList),
    qs(MinList, SortMinList),
    qs(MaxList, SortMaxList),
    append([Pivot|SortMinList], SortMaxList, SortList).

partition(_, [], [], []).
partition(Pivot, [X | List], [X | MinList], MaxList) :-
    X #< Pivot,
    partition(Pivot, List, MinList, MaxList).
partition(Pivot, [X | List], MinList, [X | MaxList]) :-
    X #> Pivot,
    partition(Pivot, List, MinList, MaxList).
```

In the expected model of `qs(A,B)` A is a list of distinct finite domain terms, B is a permutation of A and B is an increasing list (for example $X < 5 \rightarrow qs([5, X, 7], [X, 5, 7])$ is expected). In the expected model of `partition(A, B, C, D)` A is a finite domain term, B is a list of finite domain terms, C is the list of members of B which are lower than A, D is the list of members of B which are greater than A, B is obtained by a fusion of C and D.

Fig. 5.5 shows that for the goal `qs([12,X,Y,6,Z],L)` the first answer is:

```
L = [12, _#2(8..11), _#23(7..10), 6, _#44(0..5)]
X = _#2(8..11)
Y = _#23(7..10)
Z = _#44(0..5)
```

($X = _#2(8..11)$ means that X is a finite domain variable whose domain is 8..11). It is a positive symptom: L is not an increasing list. So we call the positive declarative diagnoser module of TkCalypso and a new window appears on the screen, with a drawing of the positive proof tree.

The diagnoser queries the user on some nodes of the proof-tree. For example in Fig. 5.8, the user will answer (click on) “*Not Expected*” because the last constraint in the store is `_#44(0..5) #< 6`, so `[6, _#44(0..5)]` is not an increasing list. The store simplification has been disabled on the figure and we see that without simplification the store quickly becomes unreadable.

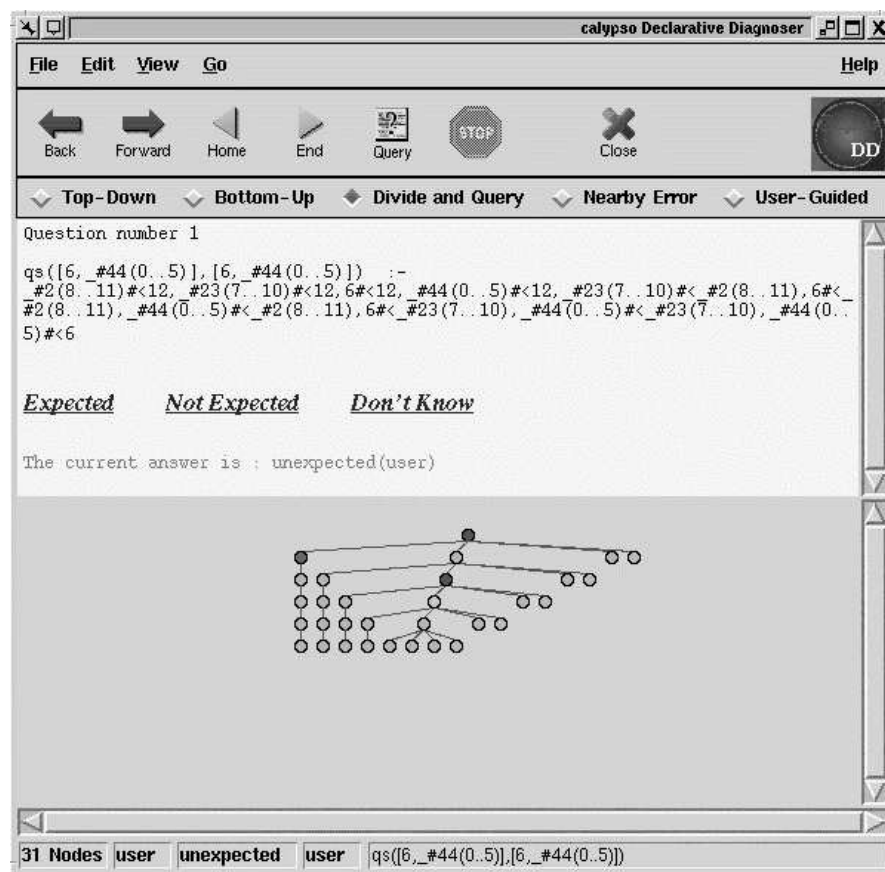


Fig. 5.8. A query

After several queries the diagnoser finds a minimal symptom and shows the corresponding error. Fig 5.9 shows an error, first it displays the incorrect clause and next the error: a clause instance and a constraint store. The problem is that in `append([A|F], G, C)` the pivot A should be between the list F and the list G: `append(F, [A|G], C)`.

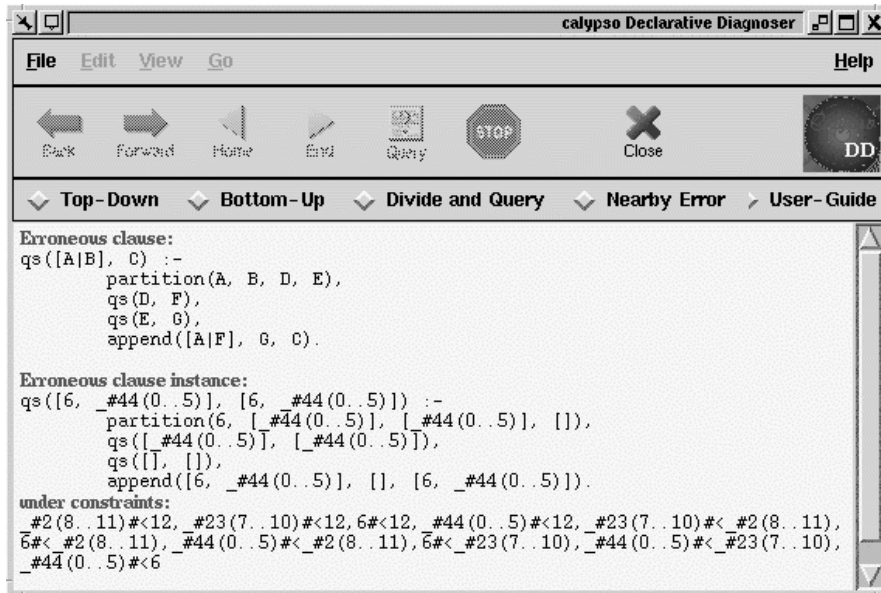


Fig. 5.9. An error

Sometimes it is not easy to fix the error provided by the declarative diagnoser, but the user is sure that the clause is incorrect, so the user does not need to search elsewhere in the program. Thus declarative diagnosis is very efficient especially for large programs (with a lot of clauses) or for large computations (search-tree with a lot of nodes). It is also a good tool for educational purpose.

5.8 Conclusion

The main remaining task concerns the interaction with the oracle.

Of course, answers to previous queries may be used by the declarative diagnoser in order to automatically answer to some other queries: Let us consider a query " $C \rightarrow A$ expected?" If it is stored that $C' \rightarrow A$ is expected and if $C \rightarrow C'$ is true then $C \rightarrow A$ is expected. If it is stored that $C' \rightarrow A$ is unexpected and if $C' \rightarrow C$ is true then $C \rightarrow A$ is unexpected. Likewise for the negative side.

The difficulty is to decide if $C \rightarrow C'$ is true (or in general $C_1 \vee \dots \vee C_m \rightarrow C'_1 \vee \dots \vee C'_n$), that is the *entailment problem*.

It is interesting to study how assertions defined in Chapter 1 could be used to answer to queries of the declarative diagnoser. Then some assertions are viewed as a partial specification of the expected semantics of the program.

Despite these techniques it is not possible to completely avoid interaction with the user. So, works in progress concerns the *presentation problem*, that is to show queries in an understandable form. Variable elimination, redundant constraint elimination, constraint simplification and approximation may be useful methods to present queries to the user.

References

- 5.1 D. Diaz. A Native Prolog Compiler with Constraint Solving over Finite Domains Edition 1.0, for GNU Prolog version 1.0.0, 1999. <http://www.gnu.org/software/prolog/>
- 5.2 W. Drabent, S. Nadjm-Tehrani, and J. Maluszyński. Algorithmic Debugging with Assertions. In Harvey Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 501–522. The MIT Press, 1989.
- 5.3 G. Ferrand and A. Tessier. Clarification of the bases of Declarative Diagnosers for CLP. Deliverable D.WP2.1.M1.1-1. Debugging Systems for Constraint Programming (ESPRIT 22532), 1997. <http://discipl.inria.fr/>
- 5.4 J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. Semantics of Constraint Logic Programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.
- 5.5 J. W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, 1995.
- 5.6 B. Malfon and A. Tessier. An Adaptation of Negative Declarative Error Diagnosis to any Computation Rule. Deliverable D.WP2.1.M2.1-1. Debugging Systems for Constraint Programming (ESPRIT 22532), 1998. <http://discipl.inria.fr/>
- 5.7 E. Y. Shapiro. Algorithmic Program Debugging. ACM Distinguished Dissertation. The MIT Press, 1982.
- 5.8 A. Tessier. Correctness and Completeness of CLP Semantics revisited with (Co-)Induction. Deliverable D.WP2.1.M2.1-2. Debugging Systems for Constraint Programming (ESPRIT 22532), 1998. <http://discipl.inria.fr/>