

## 4. Locating Type Errors in Untyped CLP Programs

Włodzimierz Drabent<sup>1,2</sup>, Jan Małuszyński<sup>1</sup>, and Paweł Pietrzak<sup>1\*</sup>

<sup>1</sup> Linköpings universitet, Department of Computer and Information Science  
S – 581 83 Linköping, Sweden

*email:* {wdr|jnz|pawpi}@ida.liu.se

<sup>2</sup> Institute of Computer Science, Polish Academy of Sciences, ul. Ordona 21,  
P1 – 01-237 Warszawa, Poland

This chapter presents a *static diagnosis* tool that locates type errors in untyped CLP programs without executing them. The existing prototype is specialised for the programming language CHIP [4.10], but the idea applies to any CLP language. The tool works with approximated specifications which describe types of procedure calls and successes. The specifications are expressed as a certain kind of term grammars. The tool automatically locates at compile time all the errors (with respect to a given specification) in a program. The located erroneous program fragments are (prefixes of) clauses. The tool aids the user in constructing specifications incrementally; often a fragment of the specification is already sufficient to locate an error. The presentation is informal. The focus is on the motivation of this work and on the functionality of the tool. Some related formal aspects are discussed in [4.15, 4.29]. The prototype tool is available from <http://www.ida.liu.se/~pawpi/Diagnoser/diagnoser.html>.

### 4.1 Introduction

In a traditional setting, the process of locating an error starts with a *symptom* observed when running the program on test input data. A symptom is a discrepancy between some user expectations and the behaviour of the program at hand, e.g. a wrong computed answer or a procedure call with arguments which are outside the intended domain of application. After a symptom has been obtained, one wants to locate the *error* that is a minimal fragment of the program causing the symptom.

A rather ad hoc approach to locating an error is tracing the execution which shows a symptom. In the case of declarative languages, tracing is particularly difficult because the execution steps are rather complex and not reflected explicitly in the program. A more systematic technique for locating errors is *declarative* or *algorithmic* debugging proposed in [4.31] for logic programs (see also [4.18, 4.25]). Declarative diagnosis of CLP programs is studied in Chapter 5 of this volume.

---

\* The authors acknowledge the contribution of Marco Comini who was largely involved in the development of an early version of the diagnosis tool [4.6, 4.7].

In contrast to the above mentioned approaches, we propose to locate errors in a CLP program without searching for symptoms, that is without executing the program. The idea can be linked to methods for proving partial correctness of a program with respect to a specification, such as [4.5, 4.12, 4.13, 4.14, 4.1]. Our tool tries to construct a proof that the program is correct w.r.t. the specification. If the proof is obtained then every execution will be free of symptoms violating the specification. Conversely, if a symptom violating the specification can be observed, a proof does not exist. In an incorrect program our tool finds all the fragments that are responsible for non existence of the proof. To use the tool the user need not be familiar with the underlying program verification techniques.

We wanted to make the diagnoser as easy to use as possible. To achieve this it was necessary:

- to choose a simple specification language easy to understand by the user,
- to minimise the specification effort necessary to locate an error,
- to allow diagnosis of separate fragments of programs,
- to provide a convenient user interface.

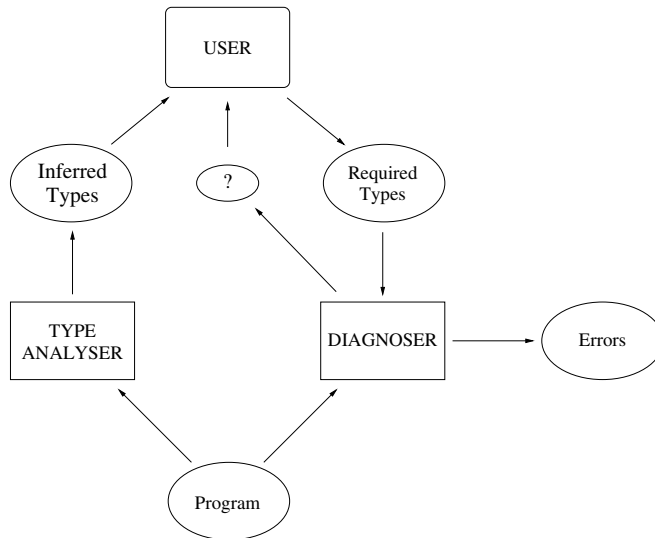
We focus on the question whether incorrect procedure calls and wrong answers may appear in some computations of a given program. The specification provided by the user describes a superset of the expected calls and a superset of the expected answers in computations of the program. The program may or may not satisfy these expectations. The above mentioned sets are described in terms of parametric types such as lists, trees, etc. The language of types is easy to understand and allows efficient checking of the verification conditions. On the other hand, this choice restricts the considered errors to type errors.

We deal with untyped CLP languages and, in contrast to a common practice in typed languages, we do not require types of program constructs to be specified a priori.

Figure 4.1 presents a general overview of the tool. The two main components are the *type analyser* and the *diagnoser*.

The analyser is used to infer types of the predicates in a given program. The role of the diagnoser is to locate the errors. Diagnosis may be requested if the inferred type of a predicate is different from that expected by the user. The diagnoser responds with a list of types on which the diagnosed type depends. In order to locate the error, some of the types in the list (in the worst case all of them) have to be specified by the user. The list of types is sorted in a way that aims at reducing the amount of interaction with the user.

The specification requests are represented by the question mark in Fig. 1. An error locating message is generated by the diagnoser as soon as a sufficient subset of the required types is specified. Providing further specification may result in locating more errors.



**Fig. 4.1.** Overview of the tool

The diagnoser finds incorrect clauses. Even more, it locates an error down to a clause prefix. It is able to locate *all* the errors in the program. In other words, all the reasons that the program behaves incorrectly (with respect to the specification) are within the located clause prefixes. Thus, if no errors are found then the program is correct. An obvious limitation is a restricted class of specifications.

The actual diagnosis is performed without referring to any symptoms. It refers neither to program execution nor to the results of program analysis. So the diagnoser can work without the analyser. The role of the analysis is auxiliary. It helps to discover that the program is possibly incorrect and to suggest a starting point of the diagnosis. The results of the analysis can be used as a draft for the specification; this simplifies the task of constructing the specification by the user.

The rest of this paper is organised as follows. Section 4.2 discusses the concept of partial correctness and the language of types as a basis for formulation of the diagnosis problem. The use of the tool and its functionality is illustrated on an example in Section 4.3. Section 4.4 explains informally the underlying principles of static diagnosis, Section 4.5 describes the treatment of delays in our approach and Section 4.6 discusses some implementation issues. Section 4.7 enumerates some limitations of our approach. Related work is surveyed in Section 4.8. Section 4.9 presents conclusions and future work.

## 4.2 The Specification Language

Intuitively, a program is correct if for any input data it behaves as expected by the user. For automatic support of (in)correctness analysis we need a language for describing both the program behaviour and user expectations. Computations of CLP programs are quite complex. Therefore we focus on selected aspects of computations, namely on calls and successes of program predicates in all computations. This section introduces a simple language for describing the form of predicate calls and successes. The actual behaviour of a program can be approximated by an automatically generated description in this language. The language will also be used for approximate specifications which describe user's expectations. Such specifications are used by our tool for automatic error location.

Notice that the form of procedure calls and successes is among the properties that can be described by assertions discussed in Chapter 1. The properties we deal with can be considered a special case of `calls` and `success` assertions, tagged with `true` or with `check` depending on whether the properties are obtained from program analysis or are a part of a specification. The actual ways of describing sets of constrained atoms, to which assertions refer, are however of secondary importance in Chapter 1. Here we introduce a formalism for describing a particular, suitable for our purposes, class of such sets.

Before introducing our specification language in Section 4.2.2, we explain how the behaviour of a CLP program is characterised in terms of predicate calls and successes. This includes discussing some basics of the chosen CLP semantics and presenting a formal definition of a procedure call and success. Some readers may prefer to skip the latter.

We assume a fixed CLP language (object language) over a fixed constraint domain  $\mathcal{D}$ . In the next subsection  $\mathcal{D}$  is arbitrary, the rest of the paper deals with CLP over finite domains. To simplify the presentation, we first present our approach applied to programs without delays (i.e. executed under the Prolog selection rule)<sup>1</sup>. Treatment of delays is discussed in Section 4.5.

### 4.2.1 Calls and Successes of a CLP Program

In this section we present the semantics of CLP used in this work and introduce the notion of an approximate specification.

The errors we want to locate demonstrate themselves as wrong computed answers or as wrong arguments of predicate calls, where “wrong” refers to user expectations or to a priori given requirements concerning the use of built-ins. These aspects of computations can be captured by associating with

---

<sup>1</sup> We consider constraints as never delayed, assuming that each constraint is passed to the constraint solver as soon as it is selected by the Prolog selection rule. The internals of the solver are outside of the scope of our semantics.

each predicate two sets: one of them describing all calls and the other all successes in the considered class of computations. This section discusses this idea in more detail. For a more formal presentation see [4.29].

We consider CLP programs executed with the Prolog selection rule (LD-resolution) and using syntactic unification in the resolution steps. In CLP with syntactic unification, function symbols occurring outside of constraints are treated as constructors. So, for instance in CLP over integers, the goal  $p(4)$  fails with the program  $\{p(2+2) \leftarrow\}$  (but the goal  $p(X+Y)$  succeeds). Terms 4 and  $2+2$  are treated as not unifiable despite having the same numerical value. Also, a constraint may distinguish such terms. For example in many constraints of CHIP, an argument may be a natural number (or a “domain variable”) but not an arithmetical expression. Resolution based on syntactic unification is used in many CLP implementations, for instance in CHIP and in SICStus [4.30].

Computations of a CLP program involve constraints. Therefore, predicate calls and successes take the form of *constrained atoms*. A constrained expression (atom, term, etc) is a pair of the form  $c \parallel E$  where  $c$  is a constraint and  $E$  is an expression such that each free variable of  $c$  occurs in  $E$ . For example,  $X :: 1..4 \parallel p(X, Y)$  is a constrained atom in CHIP notation<sup>2</sup>. For a  $c$  not satisfying the latter condition,  $c \parallel E$  will be an abbreviation for  $(\exists \dots c) \parallel E$  where the quantification is over all variables not occurring in  $E$ . A constrained expression  $true \parallel t$  may be represented as  $t$ .

We are interested in *calls* and *successes* of program predicates in computations of the program. Both calls and successes are constrained atoms. A precise definition is given below taking a natural generalisation of LD-derivation as a model of computation.

An *LD-derivation* is a sequence  $G_0, C_1, \theta_1, G_1, \dots$  of goals, input clauses and mgus (similarly to [4.26]). A goal is of the form  $c \parallel A_1, \dots, A_n$ , where  $c$  is a constraint and  $A_1, \dots, A_n$  are atomic formulae (including atomic constraints). For a goal  $G_{i-1} = c \parallel A_1, \dots, A_n$ , where  $A_1$  is not a constraint, and a clause  $C_i = H \leftarrow B_1, \dots, B_m$ , the next goal in the derivation is  $G_i = (c \parallel B_1, \dots, B_m, A_2, \dots, A_n) \theta_i$  provided that  $\theta_i$  is an mgu of  $A_1$  and  $H$ ,  $c\theta$  is satisfiable and  $G_{i-1}$  and  $C_i$  do not have common variables. If  $A_1$  is a constraint then  $G_i = c, A_1 \parallel A_2, \dots, A_n$  ( $\theta_i = \epsilon$  and  $C_i$  is empty) provided that  $c, A_1$  is satisfiable.

For a goal  $G_{i-1}$  as above we say that  $c \parallel A_1$  is a *call* (of the derivation). The call succeeds in the first goal of the form  $G_k = c' \parallel (A_2, \dots, A_n) \rho$  (where  $k \geq i$ ) of the derivation. So  $\rho = \theta_i \dots \theta_k$ . The *success* corresponding (in the derivation) to the call above is  $c' \parallel A_1 \rho$ . For example,  $X :: 1..4 \parallel p(X, Y)$  and  $X :: [1, 2, 4] \parallel p(X, 7)$  is a possible pair of a call and a success for  $p$  defined by  $p(X, 7) \leftarrow X \neq 3$ .

<sup>2</sup>  $X :: 1..4$  and  $X :: [1, 2, 3, 4]$  are two alternative ways for describing  $X \in \{1, 2, 3, 4\}$  in CHIP notation.

Notice that in this terminology constraints succeed immediately. If  $A$  is a constraint then the success of call  $c \parallel A$  is  $c, A \parallel A$ , provided  $c, A$  is satisfiable. So we do not treat constraints as delayed; we abstract from internal actions of the constraint solver.

The *call-success semantics* of a program  $P$ , for a set of initial goals  $\mathcal{G}$ , is a pair  $CS(P, \mathcal{G}) = (C, S)$  of sets of constrained atoms: the set of calls and the set of successes that occur in the LD-derivations starting from goals in  $\mathcal{G}$ . We assume without loss of generality that the initial goals are atomic.

So the call-success semantics describes precisely the calls and the successes in the considered class of computations of a given program. The question is whether this set includes “wrong” elements, unexpected by the user. To require a precise description of user expectations is usually not realistic. On the other hand, it may not be difficult to provide an approximate description  $Spec = (C', S')$  where  $C'$  and  $S'$  are sets of constrained atoms such that every expected call is in  $C'$  and every expected success is in  $S'$ . We say that  $P$  with initial goals  $\mathcal{G}$  is *partially correct* w.r.t.  $Spec$  iff  $C \subseteq C'$  and  $S \subseteq S'$ . We will usually omit the word “partially”.

Our tool uses a fixed specification language for writing descriptions of calls and successes. The precision of specification, and the errors which can be discovered are thus a priori restricted by the language. On the other hand, the simplicity of the language allows efficient automatic location of errors if the program is not correct w.r.t. a given specification. Moreover, the intelligibility of the language helps the user in taking right decisions during an interaction with the diagnoser.

#### 4.2.2 Describing Sets of Constrained Atoms

The program errors considered in this work concern discrepancies between expected and actual calls and successes, in other words, between the intended and the actual call-success semantics of the program. Thus, in order to formulate a program specification (or to present results of the static analysis to the user) we need a language for describing sets of constrained atoms and constrained terms. This section presents the specification language used in our tool. In this presentation we do not distinguish between function symbols and predicate symbols (and between terms and atoms).

For the purposes of program analysis and diagnosis, we need to compute certain operations on sets: set intersection and union (possibly approximated), inclusion and emptiness, and operations of construction and deconstruction which are explained in Section 4.4.2. The expressive power of the formalism is limited to facilitate effective and efficient computation of these operations. Due to this limitation, the call-success semantics of a CLP program is usually not expressible in the specification language and the specifications provided for programs describe approximations of the semantics. The approximations will be called *types* following the terminology used in the descriptive approach to types in logic programming.

Our formalism is a generalisation and adaptation to CLP of the ideas of [4.11]. In particular, we add a possibility to distinguish sets of ground (constrained) terms from those containing also non-ground terms.

Types will be denoted by *type terms* built of *type constructors*. Nullary type constructors are called *type constants*.

Some standard type constants are *nat*, *neg*, *any* and *anyfd*:

- *nat* denotes the set  $\{0, 1, 2, \dots\}$  of natural numbers,
- *neg* denotes the set  $\{-1, -2, \dots\}$  of negative integers,
- *any* denotes the set of all constrained terms with satisfiable constraints,
- *anyfd* denotes the set of constrained terms of the form  $c \sqcap x$ , where either  $x$  is a variable and  $c$  is a constraint describing a finite set of natural numbers, or  $x$  is a natural number. (Remember that we do not distinguish between  $true \sqcap x$  and  $x$ ). This type represents domain variables of CLP(FD) together with their instances.

The remaining types are defined by grammatical rules. For example, the rules

$$\begin{aligned} p &\rightarrow 0 \\ p &\rightarrow s(p) \end{aligned}$$

mean that the type constant  $p$  denotes the set of terms  $\{0, s(0), s(s(0)), \dots\}$ . The meaning of  $p$  is formally defined as the set of all terms not containing type symbols that can be derived from  $p$  by applying the grammatical rules. For a precise definition the reader is referred to [4.17].

Type *int* of integer numbers may be expressed as union of *neg* and *nat*:

$$\begin{aligned} int &\rightarrow neg \\ int &\rightarrow nat \end{aligned}$$

Parametric rules with non-ground type terms are also allowed. To apply such a rule, one has to substitute ground type terms for type variables. For example, lists are defined by the following parametric rules:

$$\begin{aligned} list(\alpha) &\rightarrow [] \\ list(\alpha) &\rightarrow [\alpha | list(\alpha)] \end{aligned}$$

Substituting *int* for  $\alpha$  gives grammatical rules defining the type  $list(int)$ , of integer lists. Substituting *anyfd* for  $\alpha$  gives rules defining the type  $list(anyfd)$  of lists with elements whose type is *anyfd*. The following example shows such a list and illustrates how grammar rules are applied to constrained terms. From  $list(anyfd)$  one can derive  $[anyfd | list(anyfd)]$  by applying the second rule once. According to the former definition, from the standard symbol *anyfd* one can derive, for instance,  $X \in \{1, 5, 7\} \sqcap X$ . Hence from  $[anyfd | list(anyfd)]$  one can obtain  $X \in \{1, 5, 7\} \sqcap [X | list(anyfd)]$  and, in further five steps,  $X \in \{1, 5, 7\}, Y \in \{2..6\} \sqcap [X, 3, Y]$ . The latter constrained term belongs to type  $list(anyfd)$ , as it has been generated from  $list(anyfd)$  and does not contain type symbols.

More precisely, a grammatical rule defining an  $n$ -ary type constructor  $t$  ( $n \geq 0$ ) is an expression of the form:

$$t(\alpha_1, \dots, \alpha_n) \rightarrow f(\tau_1, \dots, \tau_k)$$

where  $\alpha_1, \dots, \alpha_n$  are distinct type variables,  $f$  is a standard type constant ( $k = 0$ ) or  $f$  is a function symbol of the object language ( $k \geq 0$ ), and each  $\tau_i$  ( $1 \leq i \leq k$ ) is:

- a type constant or
- a type variable from the set  $\{\alpha_1, \dots, \alpha_n\}$ , or
- type terms of the form  $t_i(\alpha_{i_1}, \dots, \alpha_{i_l})$ , where  $t_i$  is a  $l$ -ary type constructor and  $\{\alpha_{i_1}, \dots, \alpha_{i_l}\} \subseteq \{\alpha_1, \dots, \alpha_n\}$ .<sup>3</sup>

In addition, it is required that no function symbol is a principal symbol of two distinct grammar rules defining the same type constructor. Here by a principal symbol of a rule we mean such  $f$  that  $\dots \sqcap f(\dots)$  can be generated from the right hand side of the rule. A finite set of grammar rules satisfying the latter condition will be called a (parametric) *term grammar*.<sup>4</sup>

The types are communicated between the user and the tool in the form of type terms. They refer to a fixed library of grammar rules and to additional rules declared by the user. These rules describe, respectively, the standard types of the tool and the types the user expects to be useful. The analyser generates new rules, if the present ones are insufficient to describe the results of the analysis.

In our approach, the call-success semantics of the program is approximated by providing for each predicate a *call type* and a *success type*; they are supersets of, respectively, the set of calls and the set of successes of this predicate.

### 4.3 An Example Diagnosis Session

This section gives an informal introduction to our diagnosis technique by demonstrating the use of our diagnoser on an example.

The input of our tool is a CHIP program augmented with an entry declaration specifying a class of initial goals. The result of an interactive diagnosis session is a specification describing the intended types of program predicates and error messages locating clauses responsible for incorrectness of the program w.r.t. this specification.

<sup>3</sup> This restriction on the form of  $\tau_i$  is added for technical reasons. It implies, informally speaking, that no type is defined in terms of an infinite set of types. For a more general form of this condition see [4.17]. Similar restrictions appear in other approaches to type analysis (e.g. [4.11], the restriction formulated informally).

<sup>4</sup> This extends a traditional notion of regular term grammar, see e.g. [4.11], by using type constants that can introduce constrained terms.

We will demonstrate the use of our diagnoser on the following erroneous  $n$ -queens program. The problem being solved by the program is to place  $n$  chess queens on an  $n \times n$  chess board, so that they do not attack each other. A solution to the problem is represented as a list of length  $n$ , where the value  $j$  at the  $i$ -th position means that the queen on column  $i$  is placed on row  $j$ . The error is the miss-print in the recursive definition of `safe/3` where the erroneous call `safe(T,X,K1)` appears instead of `safe(X,T,K1)`.

```
:-entry nqueens(nat,any).

nqueens(N,List):-
    length(List,N),
    List::1..N,
    constrain_queens(List),
    labeling(List,0,most_constrained,indomain).

constrain_queens([X|Y]):-
    safe(X,Y,1),
    constrain_queens(Y).
constrain_queens([]).

safe(X,[Y|T],K):-
    noattack(X,Y,K),
    K1 is K+1,
    safe(T,X,K1).
safe(_,[],_).

noattack(X,Y,K):-
    X #\= Y,
    Y #\= X + K,
    X #\= Y + K.
```

The `:-entry` declaration indicates that the predicate `nqueens/2` should be called with a natural number as the first argument (size of the chess board) and `any` term as the second argument. This includes the special case of a variable as the second argument, which however cannot be stated separately in our specification language.

The tool starts its work by computing call- and success-types of every predicate in the program. While doing this, the system informs us that the finite domain constraint `#\=` will be possibly called with incorrect types:

```
Illegal call-type of: X #\= Y
in clause (lines: 19 - 23)

noattack(X,Y,K) :-
    X #\= Y,
```

$$Y \# \backslash = X + K,$$

$$X \# \backslash = Y + K.$$

This kind of warning often coincides with a run-time error. The computed type of the first call of  $\# \backslash =$  is

```
Call-Type: t41 #\= anyfd
t41 --> anyfd
t41 --> []
t41 --> [anyfd|list(anyfd)]
```

and is not a subset of the specified call-type of  $\# \backslash =$  (where the type of the first argument is `anyfd`). Any calls of  $\# \backslash =$  outside of this type result in a run-time error.

In CHIP, the built-in predicate `is/2` is subject to delay until its second argument is ground. In our example the analyser finds that `K` is an integer at the call of `K1 is K+1`, hence this call is not delayed. The treatment of delays in our system is discussed in Section 4.5.

The inferred types can be now inspected by the user. The diagnosis should be started if some of the computed types do not correspond to the user's expectations. In our example, the inspection of the main predicate shows:

```
Call-Type: nqueens(nat,any)
Succ-Type: nqueens(nat,t67)
t50-->[]
t67-->[nat|t50]
```

The call type comes from the entry declaration and the computed success type is described by a term grammar. The intended result should be a placement of  $n$  queens on the  $n \times n$  chess board, represented by a list of natural numbers. So the expected success type of the second argument is `list(nat)` (as defined by the grammar rules in Section 4.2.2 with  $\alpha$  set to `nat`) and not the type constructed, which denotes a singleton list of natural numbers. We request diagnosis of the inspected predicate. In response, the diagnoser finds all predicates which may influence the types of this predicate, and asks the user about their intended call- and success-types.

In our example, the diagnoser will request specification of the following types, where `C` stands for “call-type” and `S` for “success-type”:

```
(C)constrain_queens/1, (C)safe/3, (S)safe/3,
(S)constrain_queens/1, (S)noattack/3, (C)noattack/3,
(S)nqueens/2.
```

The types are to be specified one-by-one in arbitrary order. The diagnoser uses this input for generating an error message locating an erroneous fragment of the program. The diagnosis procedure will be explained and justified in the next Section. As already mentioned, the error message is generated as soon as the set of already specified types makes it possible. It may not be necessary to specify all requested types.

A type may be specified either by accepting as a specification the corresponding type constructed by the analyser or by providing a specification different from the latter.

In our example session we follow the specification order suggested by the diagnoser, and we provide one-by-one the following specifications that reflect the rationale behind the program. For instance, the predicate `safe(Q,Qs,D)` describes a relation between a queen placed on a column `Q` (an argument of type *anyfd*) and queens that occupy columns `Qs` (an argument of type *list(anyfd)*) situated on the right of `Q`. The parameter `D` (of type *int*) is the distance between `Q` and the first column of `Qs`. The predicate `safe/3` only sets constraints, and therefore its call and success types are the same.

```

- (C)constrain_queens/1: accept the computed call type
  constrain_queens(list(anyfd))
- (C)safe/3: new specification
  safe(anyfd, list(anyfd), int)
- (S)safe/3: new specification
  safe(anyfd, list(anyfd), int)
- (S)constrain_queens/1: new specification
  constrain_queens(list(anyfd))
- (S)noattack/3: accept the computed success type
  noattack(anyfd, anyfd, int)

```

After providing the last specification we obtain the message telling that the clause

```

safe(X, [Y|T], K) :-
    noattack(X, Y, K),
    K1 is K+1,
    safe(T, X, K1).

```

violates the specification (see also Fig. 4.3).

Intuitively, from the specification of the success type of `noattack/3` we get the type *anyfd* for the variable `X`, whereas on the call of `safe/3` this variable is expected to be of the type *list(anyfd)*. A similar clash appears in the case of variable `T`.

Figures 4.2 and 4.3 show the graphical user interface of the diagnoser. It has three information windows and a display. After completing the analysis phase the leftmost window shows all predicates of the program. The computed call- and success- types of a predicate can be displayed by clicking a predicate in this window. If they do not conform to the user's expectation, diagnosis may be started by pressing the button "Diagnose". In response, the diagnoser generates the list of all types which may be needed to be specified for completing the diagnosis and displays them in the "Ask" window. At each step of the diagnosis session this window shows which types remain to be specified. The "User" window shows which types have been already specified during the session. In the "Ask" window one selects a type to be

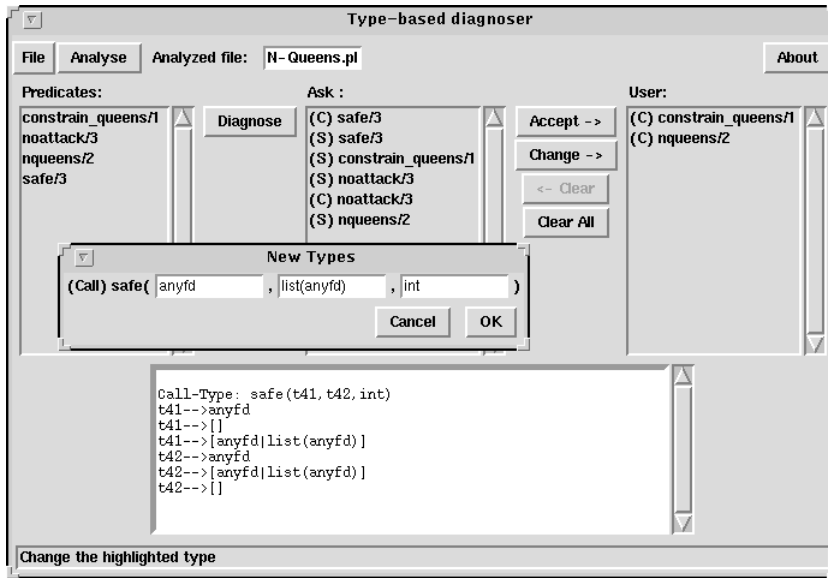


Fig. 4.2. Specifying a new call type for `safe/3`.

specified. The corresponding type inferred by the analyser is shown in the display at the bottom. The user may accept it as a part of the specification, by pressing the “Accept” button. Otherwise, clicking the “Change” button will trigger a pop-up window for writing a new specification for the active entry of the “Ask” window, as illustrated in Figure 4.2. The already specified types shown in the “User” window may be inspected and withdrawn (by pressing the button “Clear” or “Clear All”), if the user decides to specify them differently. Figure 4.3 shows the warning generated by the diagnoser.

The number of the requested types to be specified by the user in a diagnosis session depends on the predicate for which the diagnosis was initiated. This number is often rather small. However, in our example the request concerned the main predicate so the initial “Ask” list included the types of all the predicates in the program (except for built-ins, whose types have been retrieved from a library, and the call-type of the main predicate).

Observe that the diagnosis engine is a compile time tool. It does not require performing any test computations of the diagnosed program. In contrast to the usual debugging techniques, like tracing or even declarative debugging, it is not driven by error symptoms appearing in test computations. The diagnoser has access only to the diagnosed program and to its specification constructed interactively by the user with the help of the results of static analysis. For finding the error it uses partial correctness proof techniques. As the example shows, an error may be sometimes located by providing only part of the specification. The role of the program analyser is auxiliary and

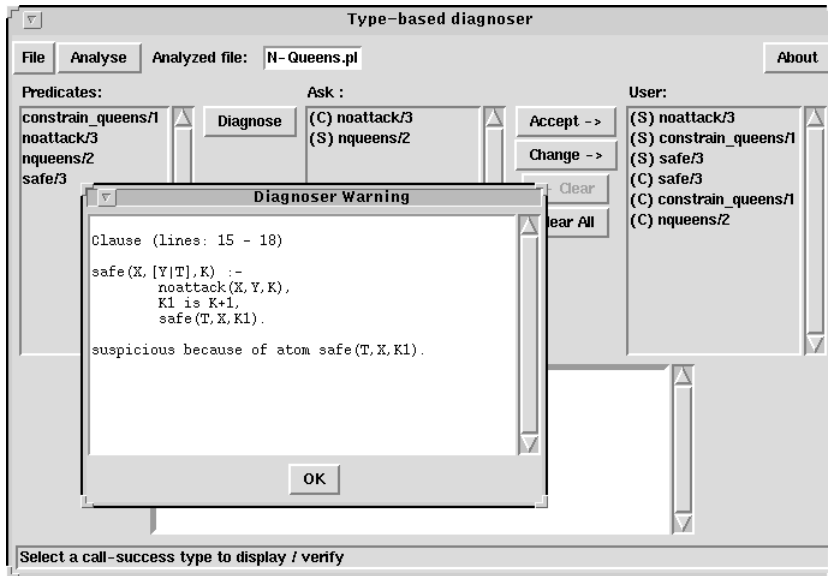


Fig. 4.3. A warning given by the diagnoser.

in principle diagnosis can be done without it. However, in our tool this option has not been implemented and the analyser is invoked at the beginning of each session. Knowing a difference between a type found by the analyser and that expected by the user, makes it possible to focus the diagnosis to a fragment of the program. The part of the analyser's output that is found by the user to be correct, is re-used as a (part of) the specification.

In our example, the diagnosis was started because the success type of `nqueens/2` computed by the analyser did not agree with the user's expectations. Notice that it was a proper subset of the expected type. This suggests *incompleteness*, i.e. the fact that some of the expected answers cannot be computed by the program. An incomplete program can still be partially correct w.r.t. to the specification considered. However, an error in a program often causes both incorrectness and incompleteness. Luckily, this was the case in the example even though the incorrectness was not visible in the type of the main predicate found by the analysis.

#### 4.4 The Diagnosis Method

Now we describe the principles of our diagnosis approach. As already mentioned, to simplify the presentation we assume here that programs are executed under the Prolog selection rule (i.e. without delays).

#### 4.4.1 Correct and Incorrect Clauses

The purpose of diagnosis is to locate in a program all the fragments responsible for its behaviour being incorrect w.r.t. a given specification. They should be as small as possible. The erroneous fragments located in our approach are program clauses, more precisely *clause prefixes*. A prefix of a clause  $H \leftarrow B_1, \dots, B_n$  is any formula  $H \leftarrow B_1, \dots, B_k$ , where  $k \leq n$ .

We have to define precisely what an incorrect clause (prefix) is. It has to be *the* reason that some call or success in the computations violates the specification. The idea is that a clause is considered incorrect if it leads to an incorrect call or success, despite the rest of the program behaving correctly.

*Example 4.4.1.* We show an example of a clause that leads to an incorrect call even if the rest of the program behaves correctly. Consider the clause located by our diagnoser in the example session of Section 4.3.

```
safe(X, [Y|T], K) :-
    noattack(X, Y, K),
    K1 is K + 1,
    safe(T, X, K1).
```

Assume that the rest of the program behaves correctly: each success of a predicate is in its specified success-type. With this assumption we examine the clause. The specification is that given to the system in the example diagnosis session of Section 4.3. Additionally, the call type of `noattack/3` is the same as its success type.

1. Assume that the clause is resolved with a call that belongs to the specified call-type `safe(anyfd, list(anyfd), int)`. Hence, the subsequent call to `noattack/3` must belong to `noattack(anyfd, anyfd, int)`; this follows from the structure of arguments in the head of the clause and from the definition of the type `list(anyfd)`. This conforms to the specified call-type of `noattack/3`, the obtained type is a subset of the specified call-type of `noattack/3`.
2. Assume that the call to `noattack/3` succeeds and the success of this call is in the specified success-type `noattack(anyfd, anyfd, int)`. Hence the built-in `is/2` is called according to its specification (which requires that the second argument is a ground arithmetic expression).
3. Consequently the success of `is/2` is in its specified success type (`K1` is bound to an integer). Hence the subsequent call to `safe/3` is in the type `safe(list(anyfd), anyfd, int)`. This is in conflict with the specified call-type of `safe`, which is `safe(anyfd, list(anyfd), int)`. The former is not a subset of the later.

Thus, even if the rest of the program behaves correctly, the clause leads to an incorrect call of `safe`. This illustrates our idea of incorrect clause. An incorrect prefix is the fragment of the clause beginning with its head and ending with the atom, of which an incorrect call was found.

Similarly, a clause may lead to a success that is not in the success-type of the head predicate. For example, the clause  $p(X) :- q(Y)$  with the specification (C)  $p(\text{any})$ , (S)  $p(\text{int})$ , (C)  $q(\text{any})$ , (S)  $q(\text{int})$  is incorrect since even if the rest of the program behaves as specified this clause may lead to a success that is not in the specified success type  $p(\text{int})$ .

Now we describe the concept of incorrect clause in a more formal way. It is defined w.r.t. to a fixed specification (which includes call- and success types of constraint predicates and built-ins). Below, by a correct call or success we mean a call or success correct w.r.t. this specification. Consider a clause

$$C = p_0(\mathbf{u}_0) \leftarrow p_1(\mathbf{u}_1), \dots, p_n(\mathbf{u}_n).$$

Assume that at some step of derivation a correct call  $c \parallel p_0(\mathbf{t})$  appears and clause  $C$  is used. Assume that then (some instances of)  $p_1(\mathbf{u}_1), \dots, p_k(\mathbf{u}_k)$  become current calls ( $0 \leq k \leq n$ ). Let us denote these calls by  $A_1, \dots, A_k$ . Assume that the corresponding successes  $A'_1, \dots, A'_{k-1}$  are correct. If the call  $A_k$  is incorrect (for some choice of a correct call  $c \parallel p_0(\mathbf{t})$  and correct successes  $A'_1, \dots, A'_{k-1}$ ) then  $C$  is considered an *incorrect clause*. Moreover the prefix  $p_0(\mathbf{u}_0) \leftarrow p_1(\mathbf{u}_1), \dots, p_k(\mathbf{u}_k)$  of  $C$  is said to be incorrect.

Similarly, if  $k = n$  and all the calls  $A_1, \dots, A_n$  succeed then the first call  $c \parallel p_0(\mathbf{t})$  also succeeds. If the latter success is incorrect, for some choice of a correct call  $c \parallel p_0(\mathbf{t})$  and correct successes  $A'_1, \dots, A'_{k-1}$ , then  $C$  is an *incorrect clause*. (We also say that  $C$  is an incorrect prefix).

It can be proved that if the program  $P$  with a class of initial atomic goals  $\mathcal{G}$  is incorrect then it contains an incorrect clause, provided the goals in  $\mathcal{G}$  are correct. (Remember that we assume a fixed specification). Thus showing that there are no incorrect clauses means proving that the program is correct.

The reverse of this property does not hold. A correct program may contain a clause which is incorrect according to the definition above. Roughly speaking, the reason is too weak a specification. There may exist incorrect clauses in the program despite the program (together with a set of correct goals) being correct. This is related to the fact that the specification for a correct program is an over-approximation of the real call-success behaviour. The notion of incorrect clause refers to the specification, and the calls and successes obtained by the incorrect clause from the specification may not appear in the computation of the program. Notice however that it is still justified to call such a clause incorrect. When placed in another program, with the same specification for the common predicates, the clause can be a reason of program incorrectness.

*Example 4.4.2.* Here we show a correct program containing an incorrect clause. The reason is that the specification is too weak. Moreover, a sufficiently strong specification does not exist (in the considered class of specifications).

Consider a clause

```
p( X ) :- prime( N ), q( N, X ).
```

Assume that `prime` succeeds always with a prime number and that `q(N,X)` succeeds with `X` bound to a natural number if `N` is prime, and to a negative integer if `N` is not prime. Assume that the specification requires that `p` succeeds with a natural number.

The set of prime numbers cannot be expressed as a type. The best specification we can have for the success-type of the argument of `prime`, and for the call-type of the first argument of `q`, is the set of natural numbers `nat`. For the clause to be correct, the success-type of the second argument of `q` has to be `nat` (or its subset). With such a specification however, we obtain incorrectness of the part of the program defining `q`.

So we can construct specifications such that the success type of `p` is `p(nat)` and the program is correct. However for each such specification, either the clause above or some other clause of the program is incorrect.

*Example 4.4.3.* Here we show that a problem of an incorrect clause occurring in a correct program can be caused by different usages of a procedure in the program.

Consider the example from Section 4.3 and a (corrected) clause

```
safe(X, [Y|T], K) :-
    noattack(X, Y, K),
    K1 is K+1,
    safe(X, T, K1).
```

This clause is correct w.r.t. the specification discussed in that section. Let us change the specification, replacing `int` by `nat` in the call and success types of `safe`. So both these types are now `safe(anyfd, list(anyfd), nat)`. The program is still correct for this specification, the last argument of `safe` is never a negative integer. However, the clause becomes incorrect. The success type of the first argument of `is` is specified to be `int`. It cannot be any smaller type. (Imagine that `is` appears somewhere else in the program and there negative integers may result too). But now the clause is incorrect, as the call type of the third argument of `safe(X,T,K1)` we obtain `int`, which is not a subset of `nat`.

To solve the problem illustrated by the last example, one would need different type specifications for different occurrences of a predicate.

Summarising, our diagnosis method consists in finding all the incorrect clause prefixes (in a given program, w.r.t. a given specification). This means locating all the errors in the program. Maybe some located prefixes are incorrect only due to a weakness of the specification, as discussed above, and are not reasons for actual program incorrectness. However *all* the actual reasons for the program being incorrect are within the located prefixes. In particular, when no incorrect clause is found then the program is correct, provided the initial goals are correct.

### 4.4.2 Incorrectness Diagnosis

Incorrectness diagnosis is performed by automatic identification of clauses that are incorrect w.r.t. to a given specification. The core of the diagnoser is thus an algorithm that automatically checks correctness of prefixes of a clause. The idea is to use the specification for computing a superset of all possible calls of the last atom of a given prefix (and of all successes of the head if the prefix is the complete clause). Correctness of the prefix is established by checking inclusion of the result in the set defined by the call specification of this atom (or by the success specification of the head). The correctness check may be described in terms of some primitive operations. We describe them referring to the example:

- *Deconstruction.* This operation determines the type of a subterm of a given typed term. Consider the clause of Example 4.4.1. The call type of its head predicate is `safe(anyfd,list(anyfd),int)`. The head is `safe(X,[Y|T],K)`. Using the deconstruction operation we obtain types of the terms bound to the variables in the head in any correct call. For `Y` and `T` it yields types `anyfd` and `list(anyfd)`, as the type of `[Y|T]` is `list(anyfd)`. (The types of `X` and `K` are trivially `anyfd` and `int`). The reader may check that deconstruction is easy to compute using the rules defining types.
- *Type intersection.* Suppose that, by means of deconstruction, types are found for some occurrences of a variable. Then its possible values are determined by the intersection of the types. Assume that the example clause has been called as specified and that the execution reached the call of `is/2`. The variable `K` at this point is bound to a constrained term that is in the intersection of the call-type of the third argument `safe/3` and the success type of the third argument of `noattack/3`. Both types are `int`, so the intersection is trivial in this case. The algorithm used in our tool for computing intersection of types is described in [4.15, 4.29].
- *Construction.* This operation determines the type of a term from types of its subterms. Usually, new grammar rules have to be constructed to describe it. In our example, the type, say `t10`, of `K+1` can be computed knowing the type of `K` (`nat`) and `1`:

```
t10 --> nat + t11
t11 --> 1
```

- *Type inclusion.* To establish (in)correctness of a clause we have to check whether the clause leads to (in)correct calls or successes. For example, whenever the rest of the program behaves as specified, the first argument of the last body atom will be bound at call to a term in `list(anyfd)`. This has been established by deconstruction of the initial call. The specified call-type for this argument is `anyfd`. Incorrectness is established by checking that the former type is not a subset of the latter. The algorithm used in our tool for checking type inclusion is described in [4.15, 4.29]

The algorithm checking whether a clause can generate incorrect calls can now be outlined as follows:

For each body atom  $B = p(\dots)$ ,  
 for each variable  $X$  in  $B$ ,  
   for each occurrence  $i$  of  $X$  in the preceding atoms,  
     compute its type  $t_i$   
     (by type deconstruction, from the specified  
     call type of the head or, respectively,  
     success type of a body atom).  
   The type of  $X$  at  $B$  is the intersection of all  $t_i$ 's  
   (it is “any” if there are no such occurrences).  
 $type(B)$  is determined by type construction  
 from the types of its variables.  
 If  $type(B)$  is not a subset of the call type of  $p$   
 then the clause is incorrect.

The computed  $type(B)$  is the set of all calls of  $B$  that would appear in the computations starting by correct calls of the clause, provided that the success set of any body atom coincides with the specified success type of its predicate.

A similar algorithm checks whether a clause can generate incorrect successes. One just takes the clause head as  $B$  and considers all the occurrences of  $X$  in the whole clause. The obtained type of  $B$  is then checked to be a subset of the success type of  $p$ .

If the specification at hand describes all predicates of the program the above algorithms can be used to find all incorrect clauses. If some type specifications are missing, we can use the following technique. We describe it for the case where  $B = p(\dots)$  is a body atom (and under assumption that the specification of the call type of  $B$  is not missing).

First replace all the missing types by the most general type *any*. Now the computed  $type(B)$  includes all the calls of  $B$ , for any possible specification of the missing types. If  $type(B)$  is a subset of the specified call type of  $p$  then no incorrectness related to  $B$  may occur. The last condition of the algorithm is independent from the missing types. Conversely, suppose that after replacing the missing types by the empty type the computed call-type of  $B$  is not a subset of the specified one. Then the clause is incorrect.

## 4.5 Delays

Modern Prolog and CLP implementations use Prolog selection rule with delays. This means that, for some predicates, an atom can be selected only if its arguments are sufficiently instantiated. The requirements on the form of selectable goals are given by delay declarations. We assume that the set of

selectable goals can be described in our type formalism. In this section we explain handling of delays by our system. The presentation is informal. The predicates subject to delays will be called *delayed* predicates. For simplicity, we deal only with built in delayed predicates.

The approach is based on a simple but rather imprecise approximation of the semantics of delays. The main idea is to deduce, whenever possible, that a predicate call will not be blocked (i.e. will be selected by the Prolog selection rule). Then we treat it as described in the previous section. Otherwise we do not conclude anything about when the call will be actually selected.

In the context of delays we have an additional notion of an initial procedure call. By an *initial call* (of procedure  $p$ ) we mean a constrained atom  $c \llbracket p(\dots) \rrbracket$  such that  $p(\dots)$  is selected by the Prolog selection rule in a goal with constraints  $c$ . If the atom is selectable then it is actually selected. Otherwise it is *blocked*. By an (*actual*) *call* we mean a constrained atom  $c \llbracket p(\dots) \rrbracket$  such that  $p(\dots)$  is actually selected (in a goal with constraints  $c$ ).

Also the notion of success is different. It is convenient to treat a (sub)goal as succeeding also when some blocked atomic goals remain unresolved. To describe this notion of success more precisely, assume that all the unifications in the resolution are variable renamings. Consider a derivation starting from a goal  $G_0 = A_1, \dots, A_n$ . Assume that the first atom of  $G_0$  is actually selected. Consider the first goal in the derivation containing atoms  $A_2, \dots, A_n$  and possibly some blocked, non selectable atoms. Let  $c$  be the constraint of this goal. Then  $c \llbracket A_1 \rrbracket$  is the *partial success* of  $A_1$  in this derivation. So for computations without delays partial successes coincide with successes. If a constrained atom is blocked then it itself is its partial success. Now we assume that a success-type specifies (a superset of) the set of partial successes of the selectable calls of the respective predicate.

As discussed in Section 4.2.1, we do not treat constraint predicates as delayed. Whenever an atomic constraint is initially selected, it is added to the constraint store. Thus it immediately succeeds or fails depending on the result of the constraint consistency check.

For each delayed predicate, instead of its call-type, we maintain its selection-type describing the set of selectable calls. If an initial call is in this set then it is immediately selected (i.e. not delayed)<sup>5</sup>. For instance, the selection-type for the built-in predicate `is/2` of CHIP is the set of constrained atoms  $c \llbracket (u \text{ is } w) \rrbracket$  where  $w$  is a ground arithmetic expression (and  $c, u$  are arbitrary).

Now we present an algorithm for checking correctness of a clause in the context of derivations with delays. The algorithm is a modification of that shown in Section 4.4.2. Given a clause  $H \leftarrow B_1, \dots, B_n$ , the algorithm approximates the set of initial calls of each  $B_i$ . If the result is a subset of the

---

<sup>5</sup> So if we want to approximate the set of the selectable calls, the selection-type has to be a subset of this set, not a superset.

selection-type of  $B_i$  then we are sure that  $B_i$  is not blocked (even if its predicate is a delayed one).

```

Mark { as not blocked } the head  $H$  and the body atoms with non delayed
predicates.
For  $i = 1, \dots, n$ 
  { Compute what happens before the initial call of  $B_i$  }
  For each variable  $X$  occurring in  $B_i$ ,
    for each occurrence  $x$  of  $X$  in a marked atom from  $H, B_1, \dots, B_{i-1}$ 
      compute its type  $t_x$  using type deconstruction operation
      (from the specified call type of  $H$  or, respectively,
      the success type of  $B_j$ ).
    Compute the intersection  $t_X$  of all  $t_x$ 's.
    { Type  $t_X$  is an approximation of the set of values of  $X$  at the
    initial call of  $B_i$  }
  Compute an approximation  $type(B_i)$  of the set of initial calls of  $B_i$ ,
  from the types  $t_X$  of its variables, by the type construction operations.
  If  $type(B_i)$  is a subset of the call/selection-type of  $B_i$  then
    {  $B_i$  is not blocked and the clause prefix  $H \leftarrow B_0, \dots, B_i$  is correct }
    mark  $B_i$ .
  Else
    if  $B_i$  is a call of a non delayed predicate then
      { the prefix may be incorrect }
      signal a warning,
    else
      {  $B_i$  may be blocked. }
      If the intersection of  $type(B_i)$  and the call/selection-type of  $B_i$  is
      empty then
        {  $B_i$  will never be selected }
        signal a warning,
      else
        { the clause prefix  $H \leftarrow B_0, \dots, B_i$  is correct }

```

A similar, second part of the algorithm checks that the clause cannot generate incorrect successes of  $H$ .

If the algorithm does not find any incorrect clause prefix in a program  $P$  then  $P$  is correct in the following sense. In any computation of  $P$  starting from an atomic goal which is in its call/selection-type,

- any (actual) call is in its call/selection-type,
- if an atom has been marked as not blocked then any its call is not blocked (any its initial call is an actual call),
- any partial success of a non blocked call is in its success type.

The same property holds if the only warnings issued by the algorithm are related to the last check in the algorithm ( $B_i$  never selected). Such a warning may correspond to a run-time error. As an example take  $B_i$  to be `...is...` and suppose that no atom in  $type(B_i)$  has an arithmetical expression as the second argument, so that the intersection considered in the check is empty.

Then execution of the program by CHIP will stop with a run-time error (unless  $B_i$  is never initially selected).

Starting from the algorithm described above it is rather obvious how to generalize our type analysis algorithm for programs with delays.

*Example 4.5.1.* In the scalar product program below, each call of `is/2` is blocked, as the second argument is not ground.

```
:- entry sp(any,list(int),list(int)).

sp(N1,[P|T],[Q|R]) :- N1 is P*Q+N, sp(N,T,R).
sp(0,[],[]).
```

This program is found correct when both the call- and success-type of `sp` are specified to be `sp(any,list(int),list(int))`. Also program analysis finds these types. Actually, the program is correct also for the success-type `sp(int,list(int),list(int))`, as the blocked calls of `is` are eventually selected and `sp` succeeds with ground first argument. Our algorithm is however too weak to determine the correctness of  $P$  for this stronger specification.

The described approach is applicable to any kind of delayed predicates, not only to built-ins, but in the current version of the tool delay declarations are not yet supported.

An advantage of the approach is its simplicity and ease of augmenting the system to handle delays. Its generalizations are a subject of future work. Existing static analysis techniques of delays (see e.g. [4.22] and references therein) are potentially interesting in this context. Applying them for our purposes may however require specifications which describe more than just call- and success-types. We expect that a major improvement in our approach can be achieved by discovering the cases when a blocked body atom is selected before “the clause succeeds”. It seems that it is possible to extend for this purpose the dependency technique of directional types discussed in [4.2].

## 4.6 The Diagnosis Tool

This section surveys the main design decisions of the existing prototype implementation of our tool. Its main components are the analyser that computes types (which approximate the actual semantics of a given program) and the diagnoser locating erroneous clauses. They have some common parts. We begin the section by explaining the usage of parametric and parameterless grammars by the tool. Then we describe its two main parts.

Both the analysis and diagnosis algorithms work with types represented as parameterless regular term grammars. Equivalently, such grammars can be seen as a restricted class of CLP programs [4.15]. Parametric grammars are employed only in the user interface. There is a library of type definitions

which may be augmented by the user. It contains for instance a parametric grammar defining type  $list(\alpha)$  (Cf. Section 4.2.2). Whenever possible, the types computed by the system are presented to the user in terms of those defined in the library or declared by the user. In this way the user faces familiar and meaningful type names instead of artificial ones. For instance, assume that the system has to display a type  $t77$  together with grammar rules  $t77 \rightarrow []$ ,  $t77 \rightarrow [t78|t77]$ . It finds that they are an instance of the rules defining  $list(\alpha)$  and displays  $list(t78)$  instead. Similarly, the user's input can refer to the types defined in the library.

The principles of the analysis are described in a separate paper [4.16]. The analyser and the diagnoser share some basic components. The algorithm is implemented in SICStus Prolog [4.30], the implementation is based on that by [4.20]. We made several lower level improvements to the original implementation, like introducing more efficient data structures (AVL trees instead of lists). They resulted in substantial improvement in analysis efficiency. Running the analyser on a number of examples, we have observed speedup with an approximate factor 4.

The type analysis algorithm constructs call and success types of the predicates defined by program clauses, thus computing an approximation of their call-success semantics. To be able to deal with real programs, it uses a library of type specifications of built-in predicates. Similarly it is able to deal with fragments of programs (for instance with programs under development). In the latter case the user is required to provide type descriptions for the undefined predicates. Such descriptions correspond to `trust` assertions of Chapter 1.

The types constructed by the analyser are on request shown to the user, who may decide to start diagnosis, as illustrated in Section 4.3. The diagnosis relies on the type specification provided incrementally by the user. As discussed in Section 4.3, the specification process is supported by the possibility to accept some types constructed in the analysis phase as specified ones. It is also restricted to the predicates relevant for the diagnosed predicate. Moreover, a heuristics is used to suggest to the user the order of specifying types. The suggestion is reflected by the order of requests in the "ask" window. Following this order often results in fewer type specifications needed to locate an error. The user may stop the diagnosis with the first error message, which is often obtained without specifying all requested types. The diagnosis process may be continued by specifying all requested types. In this case, the tool will locate all incorrect clause prefixes in the fragment of the program relevant for the diagnosed predicate.

An error message contains an incorrect clause. Its incorrect prefix is indicated by referring to the atom whose type computed by the method of Section 4.4 is not a subset of the respective specified type. This atom is the head of  $C$  or the last atom of an incorrect prefix.

The specification provided by the user is stored by the diagnoser and may be re-used during further diagnosis sessions.

## 4.7 Limitations of the Approach

Our approach can be characterized by several design decisions that we have made, such as the semantics considered, the specification language and the diagnosis method. As always in such cases, there is a trade-off between efficiency and precision. Therefore, some limitations of the framework were inevitable. This section surveys them.

We discuss the restricted expressive power of type specifications, a need for incompleteness diagnosis and a need to have more than one pair of types for a predicate. We also show an example where the call-success semantics is inappropriate to express programmer intuitions.

The restricted expressive power of our specification language makes it possible to perform effective analysis and diagnosis of programs but permits to detect only the errors that violate specifications expressible in this language. This is illustrated by the following example.

*Example 4.7.1.* Let us consider the following erroneous *append* program, where variables *Xs* and *Ys* are swapped in the clause body.

```
:- entry app(list(int),list(int),any).

app([], Xs, Xs).
app([X|Xs], Ys, [X|Zs]) :-
    app(Ys, Xs, Zs).
```

What we obtain if we invoke the call-success diagnoser with *intended* success-type `app(list(int),list(int),list(int))` is that there are no incorrect clauses. Hence, the program is correct w.r.t. the intended specification. This kind of error cannot be detected by a type based approach because, informally speaking, the type of the two variables *Xs* and *Ys* is the same.

For some built-in predicates, the restricted expressive power of types makes it impossible to express the form of the allowed calls. Thus our approach is not able to discover some of the errors of an incorrect call of a built-in.

We illustrate the problem of lack of the incompleteness diagnosis in our tool by the following example.

*Example 4.7.2.* Consider an erroneous version of a program for “closing” open (or partial) lists.

```
:-entry close_list(any).

close_list([]).
```

```
close_list([_|Xs]):-
  close_list(xs).
```

The recursive call of `close_list/1` has the constant `xs` as its argument instead of the variable `Xs`. The type inferred by the analyser is:

```
Succ-Type: close_list(t15)
t15-->[]
```

It is obvious that some values (i.e. all the non-empty lists) cannot be computed by the program. That means an incompleteness symptom. Assume now that the programmer has completed the specification with the success type `close_list(list(any))`. Observe that, as the second clause always fails, the only success of `close_list/1` (i.e. `close_list([])`), is contained in the type `close_list(list(any))` and therefore the diagnoser will give us no warning. Thus the program is correct w.r.t. the specification, although it is incomplete.

Next, we show that having only one call and one success type per predicate is actually a restriction.

*Example 4.7.3.* The program below deletes an integer number from a given list of integers, by means of the `app/3` predicate.

```
:-entry del(int,list(int),any).

del(E,L,L0):-
  app(L1,[E|L2],L),
  app(L1,L2,L0).

app([],Xs,Xs).
app([X|Xs],Ys,[X|Zs]):-
  app(Xs,Ys,Zs).
```

Observe, that in the clause defining `del/2` the predicate `app/3` is used in two ways: first to decompose the list `L` and then to concatenate `L1` and `L2`. The types inferred by the analyser are:

```
Call-Type: del(int,list(int),any)
Succ-Type: del(int,list(int),any)

Call-Type: app(any,any,any)
Succ-Type: app(list(any),any,any)
```

The call type of `app/3` is so general because it has been computed taking into account all three usages of this predicate in the program. Call types originating from these three program points have been merged, by means of the upper bound operation. Notice, that no specification (with an entry declaration as above) for which the program remains correct, can contain more accurate call type for `app/3`, as we can specify only one call type per predicate.

Analysing various calls of the same predicate separately would obviously bring more precise results and would require employing a *polyvariant* analysis method, as that of [4.24].

Essentially the same limitation affects the treatment of built-ins. Their success types are described taking into account all their possible usages. As pointed out in Section 4.4 this may make the diagnoser generate undesired warnings (as a clause of a correct program may turn out to be incorrect, due to a too general success type of a built-in).

The call-success semantics may not be suitable, as concerning the use of logical variables. The user may not be interested in the actual calls but rather in the successes related to initial calls. We illustrate this by the example originating from [4.2].

*Example 4.7.4.* The following program analyses a binary tree  $T$  with nodes labeled with natural numbers and constructs a binary tree  $NT$  of the same shape with all nodes labeled with the maximal label of  $T$ . The program includes a type declaration defining a parametric type `tree(A)` by two grammar rules (conf. Section 4.2.2).

```
:- typedef tree(A) --> void; t(A,tree(A),tree(A)).
:- entry maxtree(tree(nat),any).

maxtree(T,NT) :- maxt(T,Max,Max,NT).

maxt(void,_,0,void).
maxt(t(N,L,R), Max, MaxSoFar, t(Max,NewL,NewR)) :-
    maxt(L,Max,MaxL,NewL),
    maxt(R,Max,MaxR,NewR),
    max(N,MaxL,MaxR,MaxSoFar).

max(A,B,C,A) :- A >= B, A >= C.
max(A,B,C,B) :- B >= A, B >= C.
max(A,B,C,C) :- C >= A, C >= B.
```

The call-success analyser infers the following types:

```
Call-Type: maxtree(tree(nat), any)
Succ-Type: maxtree(tree(nat), tree(any))

Call-Type: maxt(tree(nat), any, any, any)
Succ-Type: maxt(tree(nat), any, nat, tree(any))

Call-Type: max(nat, nat, nat, any)
Succ-Type: max(nat, nat, nat, nat)
```

Hence, it correctly shows that during the execution some successes of `maxt` have an argument of the type `tree(any)`, since the constructed trees have nodes labeled by variables. To show that in the final result both arguments of `maxtree` are of the type `tree(int)`, one has to use a richer class of specifications<sup>6</sup> or refer to a different semantics and use different proof methods, like the method shown in [4.2]. A type diagnoser based on that method can be constructed by applying similar approximation techniques to the verification conditions of that method.

## 4.8 Related Work

Our approach and the design of our tool is a novel contribution, but its components and principles are based on well-known ideas and techniques of logic programming, which, however, require extension and adaptation to CLP.

The question of what the reason of an incorrect behaviour of a program is has been investigated in the framework of declarative diagnosis [4.31, 4.25, 4.18]. The concept of incorrectness error originating from that work can be related to the program not being partially correct w.r.t. a specification. Eventually this can be linked to violation of a verification condition in a proof method for partial correctness of run-time properties. Several such methods has been discussed in the literature, e.g. [4.14, 4.1, 4.16]. Our concept of incorrect clause can be linked to such a verification condition.

The general idea of using semantic approximations for program verification and for locating errors was discussed in our previous work [4.3]. That paper was the main inspiration of this work.

The static diagnosis technique presented here is similar to the abstract diagnosis of [4.8]. The latter is essentially a method for verifying a program against a specification that describes a program property. It is required that the space of possible properties forms a *Galois insertion* with the semantics of the programming language. Our type domain does not fit into the Galois insertion framework due to non existence of an abstraction function [4.15, 4.29]. Unlike our approach, the work of [4.8] is not focused on implementing a tool, but rather on theoretical aspects of the diagnosis problem. It aims at finding both incorrectness and incompleteness errors.

The decision to use term grammars as a specification language made it possible to extend for our purposes the well-known results and techniques on regular sets and regular term grammars and the techniques of constructing regular descriptive types for logic programs. There is a vast literature about it (see e.g. the survey in [4.27]). More specifically, our term grammars

---

<sup>6</sup> We need to express that, at a success of `maxt(T,M,Max,NT)`, `NT` is a tree with all the nodes labeled by `M` and `Max` is an integer.

can be directly linked to regular term grammars (see e.g. [4.11] and references therein) and to deterministic root-to-frontier tree automata [4.21]. Our grammars provide “ad hoc” extension of the grammars of [4.11], for dealing with CLP over finite domains. In [4.29] we present a more systematic way for extending regular term grammars with constraints, which is however not directly used in our tool. There have been many proposals for constructing descriptive regular types for logic programs, e.g. [4.28, 4.19, 4.24, 4.4]. We adopted for extension to CLP the technique of [4.20]. One of the reasons was that in that case we were able to re-use part of the analyser code for diagnosis.

The paper [4.3] was also a starting point for developing two tools that are strongly related to ours, both are described in this volume.

The first one is a framework of assertion-based debugging (Chapter 2, [4.23]). The assertion language used in that framework is a superset of our specification language. It not only allows to express call- and success-types but also richer properties of calls and successes and several other properties such as determinacy, non-failure, cost, etc. Abstract interpretation is used to infer some properties of the program, including types. (The type inference program has been ported for that purpose from our tool). The user may provide a priori a partial specification by stating some assertions. These are automatically compared with the inferred assertions. The comparison may confirm that the latter imply the former. A failure in verifying this may be due to program error or to incompleteness of the checking method. In the case of such failure the system generates a warning and allows to incorporate run-time checks for non-verified assertions. A special warning is issued when the comparison shows that the inferred assertion and the specified one do not intersect. This implies that the latter will be violated by each computation of the program (or the control will not reach the corresponding program points).

The tool reports all abstract symptoms that can be found with given assertions. It does not explicitly locate the erroneous clauses.

The assertion tool of Prolog IV, described in Chapter 3, uses verification techniques similar to ours for locating erroneous clauses at compile time. However, the assertion language is different from ours. The assertions are built from a fixed number of primitives, mostly some predefined constraints of Prolog IV. This gives less flexibility than our types, but allows direct use of constraint solver for verification of assertions. If an assertion cannot be proved statically then a run-time test is generated, like in the method of Chapter 1 and [4.23]. In contrast to our approach, the specification has to be given a priori and the specification process is not supported by static analysis.

Our work was initially described in [4.6, 4.7]. Since then, the treatment of delays was added and the presentation of the method has been substantially changed and improved. The type description formalism was changed, a graphical user interface was designed and implemented and the efficiency of the analysis has been improved by modification of the analysis algorithm.

## 4.9 Conclusions and Future Work

By extending to CLP well-known techniques of LP and combining them in an innovative way we constructed an interactive tool that facilitates location of errors in CHIP programs. The errors dealt with are incorrect calls and successes of predicates. The incorrectness is considered with respect to a restricted class of specifications, namely type specifications. The principles of our approach can be summarized as (1) automatic synthesis of types that approximate a program's semantics and are easy to understand by the user, (2) automatic location of the errors and (3) minimizing and facilitating the specification effort. As a side effect of a diagnosis session a specification of the program is obtained which may be used in future diagnosis and for documentation purposes.

In contrast to most of debugging approaches, our tool does not refer to any test computations of the program. Also, the algorithm is able to work without any information about error symptoms. As the class of considered specifications is restricted, many errors are outside the scope of the method. On the other hand, the diagnosis algorithm locates exactly those clauses (and clause prefixes) that are incorrect w.r.t. the specification.

Our approach can be seen as a kind of type checking. However, it does not impose any type discipline on the program, it does not require providing type declarations in advance and often only a part of these declarations is sufficient to locate an error.

The role of static typing in discovering errors at compile time is well known. We believe that the presented tool adds to untyped CLP languages some important advantages of statically typed languages.

As stated in Section 4.8, relevant techniques adopted concern:

- representing and manipulating regular sets [4.11],
- construction of regular approximations of logic programs [4.20],
- methods for proving properties of logic programs [4.14, 4.1, 4.13, 4.16]

We had to extend them to handle constrained terms. The efficiency of type construction by our tool was acceptable on a benchmark consisting of the CHIP demonstration programs, but we hope to be able to improve it. For instance optimizations similar to those suggested in [4.9] should be possible, as our type analysis algorithm is equivalent to bottom-up abstract interpretation of the magic transformation of the source program. Developing a different algorithm for construction of types may be another way of attacking this problem. Extension of the tool for other constraint domains would require addition of new standard types. The topics of future research include: better handling of delays, diagnosis of incompleteness, improving the heuristics of query ordering and introducing parametric polymorphism into specifications<sup>7</sup>.

<sup>7</sup> We already can specify parametric types by using type variables in grammatical rules. However non-ground type terms are not allowed in the specifications. Such

## References

- 4.1 A. Bossi and N. Cocco. Verifying correctness of logic programs. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development TAPSOFT '89, vol. 2*, pages 96–110. Springer-Verlag, 1989. Lecture Notes in Computer Science.
- 4.2 J. Boye and J. Maluszyński. Directional types and the annotation method. *Journal of Logic Programming*, 33(3):179–220, 1997.
- 4.3 F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszyński, and G. Puebla. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In M. Kamkar, editor, *Proceedings of the AADEBUG'97 (The Third International Workshop on Automated Debugging)*, pages 155–169. Linköping University, 1997.
- 4.4 W. Charatonik and A. Podelski. Directional type inference for logic programs. In G. Levi, editor, *Proc. of SAS'98*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
- 4.5 K. L. Clark. Predicate logic as computational formalism. Technical Report 79/59, Imperial College, London, December 1979.
- 4.6 M. Comini, W. Drabent, J. Maluszyński, and P. Pietrzak. A type-based diagnoser for CHIP. ESPRIT DiSCiPl deliverable, September 1998.
- 4.7 M. Comini, W. Drabent, and P. Pietrzak. Diagnosis of CHIP programs using type information. In Proceedings of APPIA-GULP-PRODE'99 – 1999 Joint Conference on Declarative Programming, L'Aquila, Italy, 1999. (Preliminary version appeared in Proceedings of Types for Constraint Logic Programming, post-conference workshop of JICSLP'98, Manchester, 1998).
- 4.8 M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–95, 1999.
- 4.9 M. Codish. Efficient goal directed bottom-up evaluation of logic programs. *Journal of Logic Programming*, 38(3):355–370, 1999.
- 4.10 Cosytec SA. *CHIP System Documentation*, 1998.
- 4.11 P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. The MIT Press, 1992.
- 4.12 P. Deransart. Proof method of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.
- 4.13 P. Deransart and J. Maluszyński. A Grammatical View of Logic Programming. The MIT Press, 1993.
- 4.14 W. Drabent and J. Maluszyński. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59(1):133–155, 1988.
- 4.15 W. Drabent and P. Pietrzak. Inferring call and success types for CLP programs. ESPRIT DiSCiPl deliverable, September 1998.
- 4.16 W. Drabent and P. Pietrzak. Type Analysis for CHIP. In A.M. Haeberer, editor, *Proc. of the Seventh International Conference on Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of LNCS, pages 389–405. Springer-Verlag, 1999.
- 4.17 W. Drabent, J. Maluszyński, and P. Pietrzak. Type-based Diagnosis of CLP Programs. *Electronic Notes in Theoretical Computer Science*, 30(4), 2000.
- 4.18 G. Ferrand. Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Method. *Journal of Logic Programming*, 4:177–198, 1987.

---

a specification would correspond to an infinite family of specifications in the present approach.

- 4.19 T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic Programs as Types for Logic Programs. In G. Kahn, editor, *Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 300–309. IEEE Computer Society Press, 1991. (Corrected version available from <http://WWW.pst.informatik.uni-muenchen.de/~fruehwir>)
- 4.20 J. Gallagher and D. A. de Waal. Fast and Precise Regular Approximations of Logic Programs. In P. Van Hentenryck, editor, *Proc. of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.
- 4.21 F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, Beyond Words. Springer-Verlag, 1997.
- 4.22 A. Heaton, P. Hill, and A. King. Analysis of Logic Programs with Delay. In *LOPSTR'97, Logic Program Synthesis and Transformation*, volume 1463 of *LNCS*, pages 148–167. Springer-Verlag, 1998.
- 4.23 M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, LNAI, pages 161–192. Springer-Verlag, 1999.
- 4.24 G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2 & 3):205–258, 1992.
- 4.25 J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.
- 4.26 J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second, extended edition, 1987.
- 4.27 P. Mildner. *Type Domains for Abstract Interpretation, A Critical Study*. PhD thesis, Uppsala University, 1999.
- 4.28 P. Mishra. Towards a theory of types in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 289–298, 1984.
- 4.29 P. Pietrzak. *Static Incorrectness Diagnosis of CLP(FD)*. Linköping Studies in Science and Technology, Lic. Thesis no. 742, Linköping University, 1998.
- 4.30 *SICStus Prolog User's Manual*. Intelligent Systems Laboratory, Swedish Institute of Computer Science, 1998.
- 4.31 E. Y. Shapiro. Algorithmic program debugging. In *Proc. Ninth Annual ACM Symp. on Principles of Programming Languages*, pages 412–531. ACM Press, 1982.