A Type-based Diagnoser for CHIP *

Marco Comini, Włodek Drabent,
Jan Małuszyński, Paweł Pietrzak
Linköping University†

Abstract

The paper presents an approach to diagnosis of CLP programs and a diagnoser for CHIP based on this approach. The objective is to achieve partial correctness of the program wrt a specification describing calls and successes of program predicates. The specification language is restricted; parametric regular term grammars with constraints are used to define the meaning of *type terms* as sets. A specification for a program consists of type terms describing sets of calls and sets of successes of program predicates. The diagnoser compares the type specification given by the user with the type information inferred from the program and localizes fragment of the program responsible for the discrepancy between the two. The type specification is not requested a priori. It is developed "by need" querying the user about the types inferred. A type error is often localised by providing only a small portion of type specification.

1 Introduction

We present an approach to diagnosis of CLP programs and a tool supporting it. In a general setting the problem addressed concerns the situation when the program is not partially correct wrt some explicit or implicit specification. We want to localize a fragment of the program responsible for that, called sometimes a program error. The specifications used in our approach concern the form of call and success of every predicate of a CLP program during

*This work has been supported by the ESPRIT 4 Project 22532 DiSCiPl.

†IDA, Linköpings universitet, S – 581 83 Linköping, Sweden, The second author is also affiliated at Institute of Computer Science, Polish Academy of Sciences; e-mail: {g-marco|wlodr|janma|pawpi}@ida.liu.se

any execution of the program starting with a class of goals. They give an approximation of the call-success semantics, described in more detail in this paper. The work is focused on the finite domain subset of CHIP [Cos96], an industrially used CLP language. The approach follows the general principles of assertion-based diagnosis for CLP formulated in [BDD+97]. The paper is an extended and modified version of the material presented at JICSLP'98 workshop on types for CLP [CDP98]

Programming errors appear commonly in program development phase. This work is an attempt to facilitate their location without running a program (which may still be incomplete). The method used by our tool localizes a clause responsible for the program being incorrect wrt a given approximate specification. The sematics under consideration describes the procedure calls and procedure successes occurring in the computations of the program (call-success semantics).

To provide information about the actual semantics of the program at hand and to simplify constructing the specification, a program analyser is used. It produces an approximation of the program's semantics. The user is prompted to construct the specification out of it; she accepts those fragments of the analyzer output which conform to her expectation and modifies the remaining ones. Only a part of the specification needed to localize the error has to be constructed.

Below we survey some main requirements and explain how our tool satisfies them.

- The tool is to be applicable to the *existing* CLP platforms.
- The use of the tool must not require excessive additional work. This is achieved by the use of program analysis techniques for automatic inference of relevant information about the program. The additional effort of the user is to inspect this information and revise it if it does not conform to the expectations. This way of constructing a specification is much simpler than constructing it from scratch.
- The tool is to be stand alone, to be used optionally at development time. This is achieved by using static analysis techniques and static diagnosis techniques. Thus our approach is different from the usual debugging techniques, based on tracing an execution where symptoms of an error have been observed.
- The errors concerned include wrong computed answers and run-time errors caused by improper calls of built-in predicates. Therefore our tool employs the call-success semantics.

- The analysis algorithms should be sufficiently efficient to handle practical programs. This puts a limitation on the kind of information inferred. In the presented tool the information inferred takes a form of regular sets, approximating the call-success semantics of the program.

- The specification language used by the tool in the dialogue with the user should be easy to understand. In our tool the language used is that of parametric types, where ground type terms represent regular sets. Standard types, like $list(X), int$, etc. are easily understood by the user. User can also define her own types by providing (parametric) rules defining regular sets to be connected with new type constants (and constructors). If applicable, the output of the analyzer is expressed in terms of predefined types. Otherwise the analyzer introduces new type definitions and presents the rules for them.

- The tool is to be applicable to incomplete/partially developed programs. This is achieved by a possibility of using type declarations of predicates whose definitions are not included in the program at hand. In particular this concerns built-in predicates whose types a provided by a library.

The work presented is based on/inspired by some well-known concepts and techniques which are adapted to the needs of constraint programming. In particular some most relevant related work includes

- the methods for proving run-time properties of logic programs of [DM88, Dra88, BC89] and their specialisations to directional types [AM94, BM97]. We adapt them to the case of a call-success semantics suitable for CLP. We formulate sufficient conditions for partial correctness. For a given program and a given specification we obtain a (finite) number of verification conditions. Each of them is linked to a clause. If the program is not correct some of them will be violated.

- the theory of regular sets of terms [DZ92, YS91, GS97]. For the specifications defining regular sets the verification conditions are decidable. Thus the clauses violating the conditions can be found automatically.

- the methods for computing regular approximations of logic programs, [GdW92, GdW94]. We adopt them for working with constraints. The inferred regular approximations satisfy the verification conditions.

- the abstract diagnosis method [CLMV98]. In that approach one has to define an *abstract domain* that provides a class of considered specifications and a technique for abstracting actual semantics of programs

to elements of this domain. Given such a specification and a program the abstraction technique makes it possible to find in the program all errors wrt the specification. Our work uses the basic idea of abstract diagnosis to the domain of types. The specification is not a priori given by the user but it is extracted in the dialog process where the type information inferred from the actual program is modified if it does not correspond to user expectations. Our abstract domain does not satisfy the conditions of the abstract diagnosis method, so the method, as it is, is inapplicable.

The paper is organized as follows. In Section 2 we present an overview of the method and illustrate it by an example. Section 3 gives the theoretical setting of the work. Section 4 describes the specification formalism that we use. Section 5 surveys some design decisions of our prototype diagnoser. The remaining sections include discussion and conclusions.

2 Overview

This section outlines informally the diagnosis problem and shows on example how the diagnoser helps to localize error in a CLP program.

2.1 Describing Calls and Successes of Predicates

We consider CLP programs executed with the Prolog computation rule.

At every step of a computation the selected atom A constrained by the actual constraint store constitues a *call* which may eventually succeed. By *success* we mean A constrained by the store in the state when the call succeeds. Thus, a call or a success is a constrained atom, i.e. a pair $c[]A$ where c is a constraint and A is an atom such that each free variable of c occurs in A. The constraint involved defines all potential restrictions which may appear later in the computation. For example, the variable X of a call $X :: 1..3[]p(X)$ may further be restricted to any subset of the set $\{1, 2, 3\}$. If the constraint allows only a unique valuation of the variables of the atom, (e.g. $X{=}1[]p(X)$) or if the atom is variable-free (e.g. $p(1)$) then the constrained atom is said to be *ground*.

The errors considered concern discrepancies between expected and actual calls and successes of computations starting with a class of goals. Notice that calls and successes are constrained atoms. Thus 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 language is presented more formally in the next section. Here we only give some explanations as a background for the introductory example.

The sets specified in the language are represented by terms. To distinguish these terms from the terms in the language we call them *type terms*.

There are some predefined basic types including: int, denoting the set of ground integer constants, nat, denoting the set of nonnegative integer constants, any, denoting the set of all constrained terms, and $anyfd$, which stands for domain variables and their instances. The latter type contains integers and constrained terms $c[]x$ where x is a variable and c is a constraint restricting the possible values of x to a finite set of integers.

Type constructors can be defined in the language as operations on sets and applied for construction of new sets. A standard type constructor $list$ is interpreted in the usual way. Thus the language includes expressions like $list(int), list(list(int))$, etc. representing lists over elements of the indicated sets.

An expression of the form $p(\tau_1, ..., \tau_n)$ where p is a predicate symbol and $\tau_1, ..., \tau_n$ are type terms will be used to describe the set of constrained atoms of the form $c_1, ..., c_n[]p(t_1, ..., t_n)$ where each $c_i[]t_i$, for $i = 1, ..., n$ is in the set described by τ_i. We will use such expressions to describe overapproximations of calls and successes of predicates in the computations starting from a given class of goals. The expressions will be called, respectively, the *call-type* and the *success-type* of p. The collection of call-types and success-types of all predicates of a program is a call-success specification.

We assume without loss of generality that the initial goal is atomic. We will specify the class of initial goals by choosing a predicate and providing its entry type. For example the declaration `:-entry p(nat,any).` identifies all atomic goals of the form $p(n, t)$ where n is non-negative integer and t is an arbitrary term.

If all calls and all successes that appear in any computation of the program starting with an initial goal in the class are included in the sets specified by a call-success specification, the program is said to be *partially correct* wrt to this specification. Such a specification is then said to be *valid* (for the program).

As already mentioned in Section 1, we have a program analyzer that derives a valid call-success type specification for a program augmented with an entry declaration. The specification derived may not conform to user expectations. The diagnoser is then used to localize the source of the discrepancy by querying the user about the intended types.

2.2 Example Diagnosis Session

We now present an example of an erroneous program and illustrate on it the use of our diagnoser. The input is the program augmented with an entry declaration specifying the class of intended initial calls.

The first step is type analysis resulting in a call-success specification. The user may inspect this specification for checking the call-type and success type of any chosen predicate. A diagnosis of a predicate may be requested if the user is not satisfied with the types inferred for it by the analyzer.

The diagnoser presents first the list of all predicates that may influence the type of the diagnosed one. A predicate usually has two entries on the list, annotated respectively as call and as success. The user is expected to provide for each entry the intended type specification. This is done by choosing the entries one-by-one in arbitrary order. For every chosen entry the type inferred by the analyzer is presented by the diagnoser. The user may accept it as an element of the constructed specification or reject it and provide a new type. During this interaction the system uses the already existing parts of the new specification for checking verification conditions of partial correctness. The failure of a verification condition is reported by a warning identifying the clause and the atom involved. A completeness result discussed later guarantees that every type incorrectness error will be reported by a warning (but a warning may not concern a real error).

Example 2.1 Let us consider the following erroneous N-Queens program. The (only) error is the misprint in the recursive definition of `safe` where the erroneous call `safe(T,Y,K1)` appears instead of `safe(X,T,K1)`.

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

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

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

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

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

The `:-entry` declaration says that the predicate `nqueens/2` should be called with an *integer* term as first argument and *any* term as second argument.

This includes the special case of variable as a second argument, which however cannot be stated separately in our specification language.

The call-success analyzer informs that one of the (finite domain) built-ins `#\=` is called with incorrect types. Moreover the inferred types are as follows: (In the notation used, `[]` denotes a singleton set including the empty list, `[t]` denotes the set of all one-element lists of type `t` and | denotes the union of sets.

```
CALL-type:    nqueens(int, any)
SUCCESS-type: nqueens(nat, ([]|[nat]) )

CALL-type:    constraint_queens(list(anyfd))
SUCCESS-type: constraint_queens( ([]|[anyfd]) )

CALL-type:    safe(list(anyfd), list(anyfd), int)
SUCCESS-type: safe(list(anyfd), [], int)

CALL-type:    noattack(list(anyfd), anyfd, int)
SUCCESS-type: noattack(anyfd, anyfd, int)
```

The user will find suspicious the success-type of the "top-level" predicate `nqueens/2` which should instead be `nqueens(nat,list(nat))` [1].

The user will thus call the diagnoser for the subprogram defining the predicate `nqueens/2`. The interaction with the diagnoser may be as follows, where the user may change the order of inspecting the types.

```
Do you like Call-Type constraint_queens(list(anyfd))?  YES

Do you like Call-Type safe(list(anyfd),list(anyfd),int)?   NO
What should it be?  anyfd, list(anyfd), int.

Do you like Succ-Type safe(list(anyfd),[],int)?  NO
What should it be?  anyfd, list(anyfd), int.

Do you like Succ-type constraint_queens( ([]|[anyfd]) )? NO
what should it be?  list(anyfd).

Do you like Succ-Type noattack(anyfd,anyfd,int)?  YES
```

[1] Notice that the inferred type is a subtype of the intended one. The point is that some expected results will not be computed. This does not concern partial correctness but *completeness* of the program.

```
Diagnoser WARNING: Clause

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

suspiciuos because of atom safe(T, Y, K1).
```

Here the diagnoser shows that this clause is wrong since the call to `safe/3` in the body does not respect the intended call-type. Also the expected success type of the head cannot be verified from the intended success type of this body atom. (The latter fact is not reported separately, as the clause has already been pointed out to the user). Indeed in this case the warning localizes the erroneous atom. The user may continue diagnosis session to check existence of other warnings.

```
Do you like Call-Type noattack(list(anyfd),anyfd,int)?  NO
What should it be?  anyfd, anyfd, int.

Do you like Succ-Type nqueens(nat, ([]|[anyfd]) )?  NO
What should it be?  nat, list(nat).

end of predicate nqueens / 2 diagnosis.
```

Thus, we are warned about *only* the erroneous clause which is indeed responsible of all symptoms. It is important to notice that all the other clauses are not reported as erroneous, as it is actually the case, even if the types derived for most of them do not conform to the user intentions. ■

We would like to point out that the diagnosis engine does not require performing any test computations of the diagnosed program. In contrast to the usual debugging techniques, like tracing or even declarative debugging [Sha82, Llo87, Fer87], it is not driven by error symptoms appearing in test computations. The diagnoser has only the access 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 localized by providing only a part of the specification. The role of the program analyzer is auxiliary, the diagnosis may be done without it. The types inferred by it may give abstract symptoms and thus make it possible to focus the diagnosis to a fragment of the program. The part of the analyzer's output that is found by the user to be correct, is re-used as a (part of) the specification she is composing.

Notice that in the example the diagnosis began after the success type of `nqueens` computed by the type analyzer was found to be a proper subset of the expected type. This was an abstract symptom of *incompleteness*, i.e. of the fact that some of the expected answers cannot be computed by the program. An incomplete program can still be partially correct wrt to the specification considered. However an error in the program causes often both incorrectness and incompleteness. Luckily this was the case in the example even if the incorrectness was not easily visible from the inferred types.

3 The Theoretical Setting

This section summarizes the theoretical basis of our diagnoser. We present the semantics used, the concept of incorrectness and we discuss the use of sufficient conditions for partial correctness as a basis for incorrectness diagnosis.

We assume that the reader is familiar with basic concepts of CLP, as explained in [JM94].

3.1 The Call-Success Semantics

This section presents the call-success semantics, which describes the procedure calls and procedure successes that occur during the computations of programs. Prolog selection rule is assumed. In logic programming a procedure call or success is an atom, in CLP it is a constrained atom. Thus we begin with introducing the notion of a constrained expression. We assume that certain *constraint domain* is given and we will denote it $\mathcal{D}$.

Definition 3.1 A *constrained expression* (atom, term, ...) is a pair $c\,[]\,E$ of a constraint c and an expression E such that each free variable of c occurs (freely) in E.

A constrained expression $c'[]E'$ is an *instance* of a constrained expression $c[]E$ if c' is satisfiable in $\mathcal{D}$ and there exists a substitution θ such that $E' = E\theta$ and $\mathcal{D} \models c' \rightarrow c\theta$ ($c\theta$ means here applying θ to the free variables of c, with a standard renaming of the non-free variables of c if a conflict arises).

If $c[]E$ is an instance of $c'[]E'$ and vice versa then $c[]E$ is a *variant* of $c'[]E'$

By the *instance-closure* $cl(E)$ of a constrained expression E we mean the set of all instances of E. For a set S of constrained expressions, its instance-closure $cl(S)$ is defined as $\bigcup_{E \in S} cl(E)$. ■

Note that, in particular, $c\theta[]E\theta$ is an instance of $c[]E$ and that $c'[]E$ is an instance of $c[]E$ whenever $\mathcal{D} \models c' \rightarrow c$. The relation of being an instance is

transitive. (Take an instance $c'[]E\theta$ of $c[]E$ and an instance $c''[]E\theta\sigma$ of $c'[]E\theta$. As $\mathcal{D} \models c'' \rightarrow c'\sigma$ and $\mathcal{D} \models c' \rightarrow c\theta$, we have $\mathcal{D} \models c'' \rightarrow c\theta\sigma$).

Notice also that if c is not satisfiable then $c[]E$ does not have any instance (it is not an instance of itself).

We will often not distinguish E from $true[]E$ and from $c[]E$ where $\mathcal{D} \models \forall c$.

Example 3.2 $a + 7$, $Z + 7$, 1+7 are instances of $X + Y$, but 8 is not.

$f(X)>3[]f(X)+7$ is an instance of $Z>3[]Z+7$, which is an instance of $Z+7$, provided that constraints $f(X)>3$ and $Z>3$, respectively, are satisfiable.

Assume a numerical domain with the standard interpretation of symbols. Then $4 + 7$ is an instance of $X=2+2[]X+7$ (but not vice versa), the latter is an instance of $Z>3[]Z+7$.

Consider CLP(FD) [Hen89]. A domain variable with the domain S, where S is a finite set of natural numbers, can be represented by a constrained variable $x \in S \,[]\, x$ (with the expected meaning of the constraint $x \in S$). ■

If $Vars(c) \not\subseteq Vars(E)$ then $c[]E$ will denote $(\exists_{-Vars(E)} c)[]E$ (where $Vars$ denotes the set of (free) variables of the indicated syntactic object $\exists_{-V}$ stands for quantification over the variables not in V).

From the formal point of view, the computations of a program are LD-derivations. We illustrate the notions of LD-derivation, procedure call and success by an example. When discussing procedure calls and successes we will not distinguish equivalent constrained atoms. Thus we will not distiguish $c[]A$ from $c'[]A$ when c and c' are equivalent.

Example 3.3 Consider a program

$$p(f(Y)) \leftarrow Y > 7,\ q(Y, Z),\ q(Z, Y).$$
$$q(V, W) \leftarrow W > V.$$

in CLP over integers and a goal $p(X)$. The following sequence of goals and substitutions is an LD-derivation of the program (and it is the only LD-derivation with this initial goal).

$p(X)$	ε
$Y>7[]q(Y, Z), q(Z, Y)$	$\{X/f(Y)\}$
$Y>7, Z>Y[]q(Z, Y)$	$\{X/f(Y), V/Y, W/Z\}$

The procedure calls in the derivation are $p(X)$, $Y>7[]q(Y, Z)$ and $Y>7, Z>Y[]q(Z, Y)$. There is one procedure success $Y>7, Z>Y[]q(Y, Z)$, corresponding to the call $Y>7[]q(Y, Z)$. The derivation cannot be extended, as the last procedure call results in an unsatisfiable constraint $Y>7, Z>Y, Y>Z$.

■

In this work we are interested in CLP with syntactic unification. This means that the function symbols outside of constraints are treated as constructors. So parameter passing is done as in Prolog, by computing an mgu. For instance, in CLP over integers a selected atom $p(2+2)$ does not unify with a clause head $p(4)$. An example of a language using syntactic unification is CHIP, an example of a language not using it is CLP(R).

It will be helpful to syntactically distinguish between procedure calls and successes. So for each predicate symbol p we introduce two new symbols ${}^\bullet p$ and $p^\bullet$; we will call them *annotated predicate symbols*. They will be used to represent, respectively, call and success instances of constrained atoms whose predicate symbol is p. Constrained atoms with annotated predicate symbols will be called annotated constrained atoms. For an atom $A = p(\tilde{t})$, we will denote ${}^\bullet p(\tilde{t})$ and $p^\bullet(\tilde{t})$ by ${}^\bullet A$ and $A^\bullet$ respectively. We will use analogous notation for constrained atoms. (If $A = c[]p(\tilde{t})$ then ${}^\bullet A = c[]{}^\bullet p(\tilde{t})$, etc). If M is a set of constrained atoms then ${}^\bullet M$ is $\{{}^\bullet A \mid A \in M\}$ and $M^\bullet$ is $\{A^\bullet \mid A \in M\}$.

Definition 3.4 Let P be a CLP program and $\mathcal{G}$ a set of atomic initial goals. The **call-success semantics** $CS(P,\mathcal{G})$ of P (wrt $\mathcal{G}$) is a set of annotated constrained atoms such that

1. ${}^\bullet A \in CS(P,\mathcal{G})$ iff there exists an LD-derivation for P with the initial goal in $\mathcal{G}$ and in which A is a procedure call;
2. $A^\bullet \in CS(P,\mathcal{G})$ iff there exists an LD-derivation for P with the initial goal in $\mathcal{G}$ and in which A is a procedure success. ■

We skip (rather obvious) formal definitions of an LD-derivation, a procedure call and procedure success [DP98]. Notice that the call-success semantics takes into account all the computations of the program, including failed and infinite ones.

3.2 Partial Correctness and Errors

The call-success semantics $CS(P\mathcal{G})$, characterizes an observable aspect of program behavior as a set of annotated constrained atoms. Ideally the user could independently provide a set S of constrained atoms as a specification of the expected behavior of the program. The following definition specializes the commonly accepted concept of *partial correctness* to the case of call-success semantics.

Definition 3.5 A program P with a class of goals $\mathcal{G}$ is *partially correct* wrt S iff $CS(P,\mathcal{G}) \subseteq S$, it is *complete* wrt to S if $S \subseteq CS(P,\mathcal{G})$. ■

Thus, partial correctness wrt S means that all calls and successes of the program in any computation starting in a goal in $\mathcal{G}$ are included in S. Consequently, a program P (together with its initial goals $\mathcal{G}$) is incorrect wrt a specification S iff $CS(P,\mathcal{G}) \not\subseteq S$. The latter holds if there exists an annotated constrained atom A such that $A \in CS(P,\mathcal{G})$ but $A \notin S$. Such an A is called a *symptom* (of incorrectness). So A corresponds to a procedure call or a procedure success that violates the specification but does occur in some computation of P (starting from an initial goal from $\mathcal{G}$).

Example 3.6 The program of Example 2.1 is incorrect wrt the specification provided by the user in the diagnosis session. In the execution of the program starting with the goal `nqueens(8,L)` the predicate `safe` will have the call `safe([X3,X4,X5,X6,X7,X8],X2,2)` but the annotated atom $^{\bullet}safe([X3,X4,X5,X6,X7,X8],X2,2)$ does not belong to the specification. ■

For the call-succes semantics we are going to formulate a sufficient condition for partial correctness. For this we need the following definition.

Definition 3.7 Let M be an instance-closed set of constrained atoms, Consider an implication $c_0, B_1, \ldots, B_n \rightarrow H$, where c_0 is a constraint, $n \geq 0$, and $B_1, \ldots, B_n, H$ are (non constraint) atoms. We say that this implication **is satisfied** wrt M (this fact will be denoted by $M \mid\!\approx c_0, B_1, \ldots, B_n \rightarrow H$) when, for any substitution θ, if $c_1[]B_1\theta, \ldots, c_n[]B_n\theta \in M$ and constraint $c := (c_0\theta, c_1, \ldots, c_n)$ is satisfiable then we have $c[]H\theta \in M$. ■

This specific concept of satisfaction is related to the fact that we describe semantics of CLP with syntactic unification as sets of constrained atoms.

The following proposition plays a central role in our approach. It provides a sufficient condition for partial correctness of CLP programs wrt specifications of the call-success semantics. It can be seen as adaptation for CLP of a certain instance of the proof method of [DM88] (or as a generalisation for CLP of the verification conditions of the proof method of [BC89]).

Proposition 3.8 Let P be a CLP program, S an instance-closed set of annotated constrained atoms and let $\mathcal{G}$ be a set of initial goals, $^{\bullet}\mathcal{G} \subseteq S$. A sufficient condition for

$$CS(P,\mathcal{G}) \subseteq S$$

(in other words for correctness of P wrt specification S) is that for each clause $H \leftarrow c, B_1, \ldots, B_n$ from P

$$
\begin{array}{l}
S \mid\!\approx c, {}^\bullet H \rightarrow {}^\bullet B_1 \\
\ldots \\
S \mid\!\approx c, {}^\bullet H, B_1^\bullet, \ldots, B_{i-1}^\bullet \rightarrow {}^\bullet B_i \\
\ldots \\
S \mid\!\approx c, {}^\bullet H, B_1^\bullet, \ldots, B_{n-1}^\bullet \rightarrow {}^\bullet B_n \\
S \mid\!\approx c, {}^\bullet H, B_1^\bullet, \ldots, B_n^\bullet \rightarrow H^\bullet.
\end{array}
\qquad \text{(VC)}
$$

Thus, every clause of P gives rise to a number of verification conditions.

Example 3.9 Consider the first clause for `safe` of Example 2.1. It gives rise to the following implications:

$$
\begin{array}{l}
true, {}^\bullet safe(X, [Y|T], K) \rightarrow {}^\bullet noattack(X, Y, 1) \\
true, {}^\bullet safe(X, [Y|T], K), noattack^\bullet(X, Y, 1) \rightarrow K1\, {}^\bullet is\, K + 1 \\
true, {}^\bullet safe(X, [Y|T], K), noattack^\bullet(X, Y, 1), K1\, is^\bullet\, K + 1 \\
\quad \rightarrow {}^\bullet safe(T, Y, K1). \\
true, {}^\bullet safe(X, [Y|T], K), noattack^\bullet(X, Y, 1), K1\, is^\bullet\, K + 1, \\
\quad safe^\bullet(T, Y, K1) \rightarrow safe^\bullet(X, [Y|T], K)
\end{array}
$$

One has to check whether they are satisfied wrt the (instance-closed) set of constrained atoms defined by the call-types and success types of the predicates involved, including the types of the built in `is` which is not specified by the user but provided by the system library.

Consider the types specified by the user in the diagnosis section of Example 2.1 and the first implication above. To check it one has to show that for every substitiution θ if $c_1[]safe(X, [Y|T], K)\theta$ belongs to the set specified as `safe(anyfd, list(anyfd), int)` then $c_1[]noattack(X, Y, K)\theta$ is in the set specified as `noattack(anyfd, anyfd, int)`. This reduces to checking whether $c_1[]Y\theta$ is in `anyfd` provided that $c_1[][Y|T]\theta$ is in `list(anyfd)`, which is the case under the usual definition of lists.

Similarly it can be seen that the check of the third implication will fail. From the assumption that $c_1[]safe(X, [Y|T], K)\theta$ belongs to the set specified by the call-type of `safe` it follows that $c_1[]T$ is in `list(anyfd)`, hence it is not in `anyfd` which would be necessary for the implication to be satisfied wrt the specification.

■

In actual CLP languages, constraints in the clauses are mixed with non constraint atoms. (In a clause $H \leftarrow B_1, \ldots, B_n$, each B_i may be an atomic constraint). Thus in LD-resolution each of the atomic constraints is selected (i.e. added to the constraint store) separately, after the success of the (non

contraint) atoms to the left of it. The last proposition can be generalized to this case. The generalization is rather obvious, to avoid technical details we do not present it here.

3.3 Incorrectness diagnosis

The role of diagnosis is to find a reason of incorrectness. By an *incorrect clause* of P wrt S we mean a clause $C \in P$, for which (some of) the conditions (VC) is not satisfied. From Proposition 3.8 it follows that if a program is incorrect then it contains an incorrect clause[2]. Such an incorrect clause will be considered as the reason of incorrectness.

One may be more specific here stating that the incorrectness is due to those atoms of the clause that occur in the implication of (VC) that is not satisfied.

The incorrectness diagnosis means finding incorrect clauses in a given program. Checking if a clause is correct boils down to checking if implications of the form like in Definition 3.7 are satisfied. Consider such an implication $C = c_0, b_1, \ldots, b_n \rightarrow H$ and a specification S. One may imagine a following procedure of checking whether $S \not\approx C$. Consider the function T_C defined by

$$T_C(S) := \{\ c[]h\theta \mid \begin{array}{l} c_i[]b_i\theta \in S, \text{ for } i = 1, \ldots, n, \\ c := (c_0\theta, c_1, \ldots, c_n) \text{ is satisfiable,} \\ \theta \text{ is a substitution} \quad \} \end{array}$$

Now $T_C(S) \subseteq S$ iff $S \not\approx C$. The problem is that $T_C(S) \subseteq S$ may be undecidable.

For practical use of the proposed approach we have to be more specific about the language of specifications. The expressions of this language have to define decidable instance-closed sets of constrained atoms and terms. In addition we require that the specifications have decidable subset property, i.e. that for arbitrary specifications S_1, S_2 it is decidable whether the set described by S_1 is a subset of the set described by S_2. In the sequel we assume that the specification language has these properties. For a specification S we will sometimes refer by S to the set defined if the meaning is clear from the context.

Assume that we have a computable approximation of the function T_C, i.e. a computable function $T_C^{\mathcal{A}}$ over specifications such that $T_C(S) \subseteq T_C^{\mathcal{A}}(S)$ (for

[2]But not vice versa. A counterexample is easy to construct using a "too weak" specification for some predicates. For instance take $P = \{\ p(x) \leftarrow q(x);\ q(a) \leftarrow\ \}$ and $\mathcal{G} = \{p(x)\}$. Let P be a correct wrt a specification S (so $CS(P, \mathcal{G}) \subseteq S$) and let S contain $q^\bullet(x)$ but not $p^\bullet(x)$. Then the last condition of (VC) does not hold.

all specifications S in our specification langage). As the check for $\subseteq$ for our specifications is decidable, this gives a decision procedure for $T_C^{\mathcal{A}}(S) \subseteq S$.

We will say that implication C is *abstracly satisfied* wrt S if $T_C^{\mathcal{A}}(S) \subseteq S$. Obviously, if C is abstracly satisfied wrt S then $S \models\!\!\approx C$ (as $T_C(S) \subseteq T_C^{\mathcal{A}}(S) \subseteq S$). We will say that conditions (VC) are abstractly satisfied if each their implication is abstractly satisfied wrt S. We obtain immediately:

Proposition 3.10 Consider a program P and a specification S. If for each clause of P the conditions (VC) are abstractly satisfied then (VC) are satisfied and P is partially correct wrt S.

Obviously the reverse may not hold, for a particular $T_C^{\mathcal{A}}$ conditions (VC) may be satisfied but not abstractly satisfied.

The proposition gives an algorithm for error diagnosis. Computing whether $T_C^{\mathcal{A}}(S) \subseteq S$ for every implication of (VC) for each clause of P makes it possible to find the set of clauses for which (VC) are not abstractly satisfied. Each clause of P for which (VC) is not satisfied is in this set. Thus this set contains all the incorrect clauses of P. (On the other hand it may contain correct clauses). If the set is empty then the program is correct wrt S.

Example 3.11 Consider *CLP(FD)* and a specification S that states that the argument of p is constrained to 1..8 at call and to 4..8 at success, while the argument of q is in `anyfd` at call and is constrained to 4..5 at success. Consider a CHIP clause `p(X) :- Y::1..10, X#=Y+4, q(Y)` (where constraint `Y::1..10` constrains variable `Y` to the interval 1..8 and constraint `X#=Y+4` constrains expressions `X` and `Y+4` to have the same numerical value). The reader may check that the clause is correct wrt this specification, and that the verification conditions imply that the argument p at success is bound to 8.

To check abstract satisfiability of (VC) we have to define the function $T_C^{\mathcal{A}}$. $T_C^{\mathcal{A}}(S)$ may state that the argument of p at success is `anyfd`. It is still a correct, though very imprecise approximation. For this choice of $T_C^{\mathcal{A}}$ does not permit to establish correctness of the clause, since the conditions (VC) are not abstractly satisfied.

■

4 The specification formalism

This section describes the specification formalism used in our diagnosis system to describe instance-closed sets of constrained terms. It is used for communication with the user, both to present the results of program analysis and to acquire specifications. It is a simple extension of the well-known

formalism of *regular term grammars* (see e.g. [DZ92] and references therein) for specifying sets of constrained terms.

The expressive power of the formalism is limited to facilitate automatic check of the verification conditions for partial correctness. 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 a program describe approximations of the semantics. We will call them *types* following the terminology used in the descriptive approach to types in logic programming. In this paper we discuss a particular class of types, that is suitable to CLP over finite domains (CLP(FD)). A more general solution is presented in [DP98].

We build our specifications over the alphabet including a set F of function symbols, a set V of variables, a set of type symbols $\mathcal{T}$ (each of certain arity) and type variables $\mathcal{TV}$. $\mathcal{T}_0 = \{ any, anyfd, nat, neg \}$ is a set of distinguished type symbols of arity 0. As we are going to specify sets of constrained terms the alphabet has also to include some constraint predicates, at least $true$ and $\in$ for expressing finite domain constraints of the form $X \in a..b$.

We denote by $Term(S_1, S_2)$ the universe of terms built from function symbols from S_1 and variables from S_2. Elements of $Term(\mathcal{T}, \emptyset)$, are called *ground type terms*. Below we describe a notion of a grammar that from a given ground type term t generates a set of constrained terms over F and V. The instance-closure of the set of all constrained terms derivable from t is the type denoted by t. The grammar provides thus a set interpretation of ground type terms.

A grammatical rule may include type variables. Such a rule is considered a shorthand for a possibly infinite set of its instances, where each occurrence of a type variable is replaced by a ground type term.

In our diagnoser the user refers to types by ground type terms that are defined by the grammatical rules stored in the system. The user has a possibility of defining new types by supplying new grammatical rules.

Definition 4.1 A parametric *regular term grammar* with constraints (in normal form) a finite set $\mathcal{R}$ of rules of the form

$$t(\alpha_1, \ldots, \alpha_n) \rightarrow f(t_1, \ldots, t_k)$$

where $t \in \mathcal{T} \setminus \mathcal{T}_0$, $\alpha_1, \ldots, \alpha_n$ are distinct type variables, $f \in F$ and $t_1, \ldots, t_k \in Term(\mathcal{T}, \{\alpha_1, \ldots, \alpha_n\})$, $n \geq 0$, $k \geq 0$. If $t(\ldots) \rightarrow f(\ldots)$ and $t(\ldots) \rightarrow f'(\ldots)$ are two distinct rules of $\mathcal{R}$ then $f \neq f'$.

A *ground type substitution* is a mapping from type variables to ground type terms. A *ground rule* is the instance of a rule in $\mathcal{R}$ under a ground type substitution. ■

Before explaining the use of the grammatical rules we illustrate the definition by an example.

Example 4.2 Let $[\,], a, b$ be unary function symbols, $|$ a binary function symbol, $t1$ a type constant, *list* a unary type constructor and let α be a type variable. We may now define the following term grammar, where $[\,|\,]$ notation is used for terms constructed with $|$.

$$\begin{aligned} &list(\alpha) \rightarrow [\,] \\ &list(\alpha) \rightarrow [\alpha|list(\alpha)] \\ &t1 \rightarrow a \\ &t1 \rightarrow b \end{aligned}$$

■

We now define a rewrite relation $\Rightarrow$ on the objects of the form $c[]t$ where c is a constraint and $t \in Term((F \cup \mathcal{T}), V)$. Thus t is a term which may include both type symbols and function symbols and ordinary variables but does not include type variables. In particular t may be a type term without type variables. $c[]t_1 \Rightarrow (c', c)[]t_2$ if t_2 is obtained from t_1 by replacing a maximal subterm $t \in Term(\mathcal{T}, \emptyset)$ by t' and adding the new constraint c' in the following way[3]

- if $t \notin \mathcal{T}_0$ then t' is a term such that $t \rightarrow t'$ is a ground rule of the grammar and c' is *true*,
- if t is *any* then t' is a new variable not appearing in t_1 and c' is *true*.
- if $t = anyfd$ then t' is a new variable x not appearing in t_1 and c' is $x \in 0..maxint$ (where *maxint* is the greatest integer permitted to occur in the finite domain constraints of the CLP(FD) language under consideration),
- if t is *nat* or *neg* then t' is a natural number or a negative integer respectively and c' is *true*.

Transitive closure of $\Rightarrow$ is denoted as $\stackrel{*}{\Rightarrow}$.

Definition 4.3 The type denoted by a given ground type term t is the set of constrained terms defined by

$$Type(t) = cl\Big(\{\, c[]t' \mid t \stackrel{*}{\Rightarrow} c[]t',\ t' \in Term(F, V)\,\}\Big)$$

■

[3] We don't distinguish e from $true[]e$.

The following example illustrates this definition.

Example 4.4 For the grammar of Example 4.2 we have

$$list(list(nat)) \Rightarrow [list(nat)|list(list(nat))] \stackrel{*}{\Rightarrow} [[1]|[\,]]$$

Hence $[[1]|[\,]] \in Type(list(list(nat)))$.

Similarly (using Prolog notational convention) $[a, b, a, a] \in Type(list(t1))$, and also $X \in 2..3, Y \in 5..7[][X, Y]$ is in $Type(list(anyfd))$. The latter example illustrates the role of instance closure in the definition of the type denoted by a ground type term. We see that

$$list(anyfd) \stackrel{*}{\Rightarrow} X, Y \in 1..maxint[][X, Y]$$

and the example constrained term belongs to the instance closure of the constrained term $X, Y \in 1..maxint[][X, Y]$.

■

A specification of a program is given by a grammar as above and, for each predicate symbol p, by a pair of expressions of the form $p(t_1, \ldots, t_n)$, where $t \in Term(\mathcal{T}, \emptyset)$ (and n is the arity of p). The expressions specify, respectively, the call and success types of p. The specification contains a constrained atom ${}^{\bullet}A$ (respectively $A^{\bullet}$) iff A is an instance of some A' generated by the grammar from the first (second) expression.

5 The Diagnoser

This section surveys main design decisions of the existing prototype implementation of the diagnoser.

Regular term grammars with constraints are represented as CLP programs of special kind, called RULC programs. This is an extension of well-known technique relating regular term grammars and logic programs of special kind, called RUL programs [YS91].

A RULC program consists of two kinds of clauses. The clauses of the form $t(f(X_1, ..., X_n)) \leftarrow t_1(X_1), ..., t_n(X_n)$ correspond to the grammatical rules of the form $t \rightarrow f(t_1, ..., t_n)$. In addition there may be clauses of the form $t(X) \leftarrow c(X)$ where c is a constraint satisfying some special requirements. In particular, the constraints generated by the term grammars discussed above can be used in such rules. For every predicate t a RULC program P specifies a set t_P of constrained terms.

There is a library of type definitions which may be augmented by the user.

The diagnoser uses the type analyzer of CHIP programs described in [DP98]. The analyzer extends the techniques of [GdW92, GdW94] to infer call-types and success-types of a given CLP program. They are presented to the user either in the form of type terms referring to the stored type definitions or in the form of grammatical rules if an appropriate type definition does not exist.

A program P to be analyzed with a given entry declaration $\mathcal{G}$ is first transformed into the set of verification conditions for the not yet known call-success specification. The set of clauses obtained in that way is a CLP program $M_{P,\mathcal{G}}$ with annotated predicates. Its c-semantics describes calls and successes of each predicate in the original program as a set of constrained atoms with annotated predicates. (A generalization of c-semantics [FLMP89] for CLP is introduced forthis purpose). The next step is to construct a RULC program whose c-semantics is a superset of the c-semantics of $M_{P,\mathcal{G}}$. The construction, described in [DP98] is a (terminating) fixpoint computation. It uses $T_C^{\mathcal{A}}$ functions that approximate the T_C operator of each clause C of $M_{P,\mathcal{G}}$, as described in Section 3.3. The specifications transformed by $T_C^{\mathcal{A}}$ operators are RULC programs.

The inferred types are on request shown to the user, who may decide to start diagnosis, as illustrated by Example 2.1. The implications from the verification conditions (VC) for the diagnosed subprogram are then checked to be correct wrt the type specification S acquired by querying the user. As discussed in Section 3.3, the check consists in computing $T_C^{\mathcal{A}}(S)$ (for an implication C) and comparing it with S. The functions $T_C^{\mathcal{A}}$ in our diagnoser are those already used in the analysis phase. The specification is obtained from the user by asking queries (about the call respectively success types of particular predicates). The user is able to choose the order of answering them, as the system displays a list of "pending" queries from which the user selects the one to answer first. This list is sorted wrt the relevance of the information to the diagnosis, i.e. the number of implications which could be checked having that information at hand.

Whenever the user decides to answer for a pending oracle query, she is first asked if the results of type analysis for calls or successes of this predicate coincide with the intended meaning of the program. If no, the user has to describe the intended semantics of the predicate, by giving types for its arguments. If yes, the relevant fragment of the results of type analysis is used.

Then all the implications that can be checked are checked, the pending list is updated taking into account the new information and unless it is not empty we wait for the user again.

The choice of implications to be checked is guided by abstract symptoms.

An *abstract incorrectness symptom* exists for the call (success) of a predicate p if the call (success) type of p computed by the program analysis is not a subset of the corresponding type given for p by the specification S. The system selects first those implications $\alpha \rightarrow \beta$ for which there exists an abstract symptom for the annotated predicate in β. This strategy results in decreasing the number of queries needed to localize the first incorrect clause. (It does not improve the number of queries needed to find all the implications from (VC) that are not abstractly satisfied).

6 Discussion

We first survey some limitations of our method.

In our approach, types are approximations of the actual and intended semantics of programs. Obviously, a program may be correct wrt its approximate specification and actually incorrect wrt some more precise specification. Incorrect results may have correct types.

Example 6.1 Let us consider the following erroneous *append* program, where we have `[X|Ys]` instead of `[X|Zs]`.

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

app([], Xs, Xs).
app([X|Xs],Ys,[X|Ys]) :- app(Xs,Ys,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 abstractly incorrect clauses. Hence, by Theorem 3.8, the program is correct wrt the intended specification. This kind of error cannot be detected by a type based approach because, informally speaking, the type of the two variables `Ys` and `Zs` is the same. ■

The restricted expressive power of our specification language makes possible effective analysis and diagnosis of programs but permits to detect only the errors violating specifications expressible in this language.

Another limitation of the method concerns the precision of the aproximation of T_C by the operator $T_C^{\mathcal{A}}$ used by the diagnoser. As illustrated by Example 3.11 a warning for a correct clause may be produced due to imprecision of $T_C^{\mathcal{A}}$.

The method presented concerns incorrectness diagnosis. Incompleteness may also be a problem and is a subject of future work. As illustrated in Example 2.1 a misprint in the program may introduce both incompletness and

incorrectness wrt the call-success semantics and can be found by incorrectness diagnosis. The question is how often incompleteness diagnosis is really needed in practice.

The presented approach assumes the Prolog selection rule, while in many CLP languages selection with delays is possible. For example in CHIP the `is` predicate is delayed if the arguments of the arithmetic expression are not sufficiently instantiated.

Example 6.2 The following scalar product program executes correctly in CHIP

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

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

but the diagnoser warns about the call-type of `is`. ■

Extension of the method to handle delays is a subject of future work.

Another case where the left-to-right call-success semantics may not be the best suitable concerns the use of logical variable. 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 [BM97].

Example 6.3 The following CHIP program analyzes a binary tree T with nodes labeled by integers and constructs a binary tree NT of the same shape with all nodes labeled by the maximal label of T. The program includes a type declaration acceptable by our diagnoser.

```
:- typedef tree(A) --> void; t(A,tree(A),tree(A)).
:- entry maxtree(tree(int),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 analyzer infers the following types:

```
CALL-type:    maxtree(tree(int), any)
SUCCESS-type: maxtree(tree(int), tree(any))

CALL-type:    maxt(tree(int), any, any, any)
SUCCESS-type: maxt(tree(int), any, int, tree(any))

CALL-type:    max(int, int, int, any)
SUCCESS-type: max(int, int, int, int)
```

Hence it correctly shows that during the execution some successes of `maxint` are of the type `tree(any)`, since the trees constructed 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[4] or refer to a different semantics and use different proof methods, like the method shown in [BM97]. A type diagnoser based on that method can be constructed by applying similar approximation techniques to the verification conditions of that method. ■

7 Conclusions

In this paper we have shown a diagnosis approach based on descriptive types, which approximate the call-success semantics of the program. We deal with type-free CLP languages, we are mainly interested in CLP(FD) and our implementation concerns the programming language CHIP. Types are expressed by a grammatical formalism (called regular term grammars with constraints). The formalism is equivalent to a restricted class of CLP programs. In our implementation the former are used in the user interface and the latter in the system internally. We present a theory of incorrectness diagnosis and describe a prototype diagnosis tool.

The very diagnosis algorithm uses a description S of types, which is an approximate specification (a superset) of the intended semantics of the program. In contrast to most of debugging tools, it does not refer to any test computations of the program; so it is a "static" approach. The algorithm works without any information about error symptoms. (However information about abstract symptoms can be used to improve its search strategy). It is able to find all the clauses of the program that are responsible for the

[4] We need to express that, at a success of `max(T,M,Max,NT)`, `NT` is a tree with all the nodes labeled by `M` and `Max` is an integer.

program being incorrect wrt the specification. It may however happen that it points out a clause that is abstractly incorrect but (concretely) correct wrt S.

The user does not have to provide the whole specification. Instead she is asked for fragments of it, when they are needed. If the relevant fragments of type analysis results coincide with the intended semantics, they can be used as specification.

Diagnosis of incomplete programs is possible. For this it is necessary to specify types of the predicates not defined in the diagnosed program. This mechanism can also be used for separate analysis/diagnosis of the selected parts of large programs.

The method needs practical evaluation, so the main subject of further work is experiments with debugging "real" CLP programs. The method deals with call-success semantics of CLP with Prolog selection rule. Generalizing it to include delays is a subject of future work. Another topic of future work is incompleteness diagnosis. There are similarities between our method and abstract diagnosis [CLMV98]. A clear relationship between these two techniques should be established.

References

[AM94] K. Apt and E. Marchiori. Reasoning about prolog programs: From modes through types to assertions. *Formal Aspects of Computing*, 3:743–765, 1994.

[BC89] 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.

[BDD+97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Małuszyński, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In M. Kamkar, editor, *Proc. of the AADEBUG'97 (The Third International Workshop on Automated Debugging)*, pages 155–169. Linköping University, 1997.

[BM97] J. Boye and J. Małuszyński. Directional types and the annotation method. *Journal of Logic Programming*, 33(3):179–220, 1997.

[CDP98] M. Comini, W. Drabent, and P. Pietrzak. Diagnosis of CHIP programs using type information. In *proceedings of Types for Constraint Logic Programming, post-conference workshop of JICSLP'98*, 1998.

[CLMV98] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 1998. To appear.

[Cos96] Cosytec SA. *CHIP System Documentation*, 1996.

[DM88] W. Drabent and J. Małuszyński. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59:133–155, 1988.

[DP98] W. Drabent and P. Pietrzak. Type analysis for CHIP. Technical report, IDA, Linköping University, 1998.

[Dra88] W. Drabent. On completeness of the inductive assertion method for logic programs. Unpublished note, Institute of Computer Science, Polish Academy of Sciences, May 1988.

[DZ92] P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

[Fer87] G. Ferrand. Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Method. *Journal of Logic Programming*, 4:177–198, 1987.

[FLMP89] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modelling of the operational behaviour of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[GdW92] J. Gallagher and D. A. de Waal. Regular approximations of logic programs and their uses. Technical Report CSTR-92-06, Department of Computer Science, University of Bristol, 1992.

[GdW94] J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[GS97] 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.

[Hen89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[JM94] J. Jaffar and M. J. Maher. Constraint Logic Programming: a Survey. *Journal of Logic Programming*, 19 & 20:503–581, 1994.

[Llo87] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

[Sha82] E. Y. Shapiro. Algorithmic program debugging. In *Proc. Ninth Annual ACM Symp. on Principles of Programming Languages*, pages 412–531. ACM Press, 1982.

[YS91] E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Journal of Logic Programming*, 10:125–135, 1991.