Static Incorrectness Diagnosis of CLP(FD)

Paweł Pietrzak
Department of Computer and Information Science
Linköping University

Acknowledgments

First and foremost, I would like to express my gratitude to my advisor Jan Małuszyński. His constant encouragement and cooperation during writing this thesis have been invaluable.

I am very grateful to Włodzimierz Drabent who considerably contributed to the work. Most of the theoretical problems have been solved with his efficient effort.

Various ideas presented in the thesis have come up after many discussions with people involved in the DiSCiPl project. Therefore I would like to thank Helmut Simonis, Abder Aggoun, Pierre Deransart, Manuel Hermenegildo, German Puebla, Francisco Bueno and John Lloyd.

John Gallagher has explained some of details of the regular approximation algorithm and has provided the source code of the tool computing regular approximations of logic programs. This code is a basis of our type inference system.

Marco Comini has implemented a prototype of the diagnoser described in Chapter 5. Discussions with him helped me in writing more clearly some fragments of the thesis.

I would like to thank Henrik Nilsson for the comments on the draft of the thesis and for the stimulating discussions on types.

Furthermore, I would like to thank Ulf Nilsson and the rest of the members of the Logic Programming Group for the creative atmosphere and much help I have received.

I am also very grateful to Lillemor Wallgren for her engagement and her efficient help in solving administrative problems. Without her it would never have been possible to finish the thesis on time.

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

Contents

Chapter 1

Introduction

1.1 Motivation and Objectives

This thesis addresses the problem of locating certain kind of errors in constraint logic programs. Constraint logic programming (CLP) [JM94, JMMS98] emerged in the eighties as an extension of logic programming and is recently gaining a growing industrial interest. Due to their expressive power the CLP languages make it often possible to provide concise programs describing complex problems in an intuitively clear declarative way. On the other hand, locating errors in a CLP program may be rather difficult because the control flow and the data flow are not explicitly described by the program and quite involved. The motivation for the work presented in this thesis is to facilitate this task.

The industrial applications of CLP concern often combinatorial optimization problems where the problem variables range over finite domains. CLP programs using only the constraints over the finite domains are called CLP(FD) programs. We are particularly interested in CLP(FD) and in the programming language CHIP [Cos96] which provides a commercial implementation of a framework of CLP(FD) [Hen89]. Our objective is to provide a tool which facilitates locating errors in CHIP programs. [1]

Locating errors in a program is a major issue in software development. An erroneous program will on some data behave differently than expected by the user. An observed discrepancy is a *symptom* of an error. By locating an error we mean finding a part of the program responsible for a symptom in some computation. Errors can be classified according to the kind of symptoms. In particular the following symptoms may be considered:

- the result computed is wrong (i.e. does not correspond to user's expec-

[1] However the presented framework is applicable to any CLP language.

tation),

- a procedure of the program is called with wrong arguments,
- the computation does not terminate,
- the time needed for computing results is too long.

This thesis discusses only the first two kinds of the errors mentioned above. We are aware of the fact that other errors may be even more difficult to locate. In particular this concerns the question what are the reasons of poor efficiency of the program, addressed by *performance debugging*. We believe that different kinds of errors require different methods and a progress on each of the topics is needed.

1.2 The Approach

Static Diagnosis

This section explains our approach to location of errors.

In a traditional setting the process of locating an error starts with a symptom observed when running the program on a test input data. We note here that a systematic testing technique may be needed to find symptoms.

A rather ad hoc approach to locating an error is tracing the execution which shows a symptom. In the case of declarative languages it 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 [Sha82] for logic programs (see also [Fer87, Llo87a]). More recently declarative debugging was extended for lazy functional programming [NF94] and constraint logic programming [Tes96]. Declarative debugging allows to abstract from details of the program execution and focuses on the declarative semantics. In this approach the execution which shows a symptom of an error is analyzed interactively by a semi-automatic system. The user is queried about elements of relations or functions defined by the program. The questions are automatically constructed from the record of the program execution and from the previous answers of the user. The session ends with a message locating a fragment of the program which causes the symptom.

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. Verification of logic programs has been discussed by many authors, see e.g. [Cla79, Der93, DM88, BC89]. Assume that the user expectations have been described by a specification and that the program was proved correct wrt it. 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. A proof of partial correctness for given program and specification consists usually of the proofs of a finite number of verification conditions linked to the constructs of the verified program. For an incorrect program some of the verification conditions cannot be proved and the reason of the incorrectness can only be linked to the corresponding constructs. We elaborate on this idea to define a static diagnosis method for CLP programs.

We note that introducing *types* to programming languages (see e.g. [Car97, CW85]) has similar motivation. In many typed languages providing type declarations is obligatory. Then either static or dynamic type checking (or combination of both) assures that the program variables can only take values of the declared types. As stated in [Car97]:

> The fundamental purpose of a **type system** is to prevent the occurrence of *execution errors* during the running of a program.

The Problems

We now outline a number of problems connected with the proposed approach.

- How to define the underlying semantics and the verification method suitable for it? The verification methods mentioned above are not directly applicable to CLP programs. We have to choose an appropriate semantics that captures the properties we want to express and we have to define a partial correctness proof method based on this semantics. As stated above, we focus on the computed answers of a CLP program and on the form of the intermediate calls during the execution.

- How to choose the specification language? It should be expressive enough to be useful during diagnosis process. On the other hand it should be decidable and suitable for efficient automatic verification of practical programs.

- How to facilitate writing of the specifications? We want to minimize the additional effort of the user needed to locate errors. Verification of a program is done with respect to a specification. The methods mentioned above require that a specification is given for every predicate of the logic program to be verified. In the case of realistic programs this requires an excessive effort.

The Design Decisions

The above stated problems are solved as follows. We define a *call-success* semantics and a verification method for CLP. It is an adaptation to CLP of the verification method for logic programs described in [DM88, BC89].

We propose to express specifications as *regular types* (see Chapter 3) and to use type inference techniques for facilitating the task of specification. As explained below we also adapt for that purpose some techniques originating from logic programming.

Following Pfenning [Pfe92a] the attempts to introduce types in logic programming can be classified as *prescriptive* and *descriptive*.

Prescriptively typed logic programs can be seen as formulae of typed logic. A syntactic condition of well-typedness is provided and the semantics is only defined for well typed programs. Early examples of this approach can be found in [MO84] and [Smo89]. [HT92] is an introductory material about prescriptive types. There are several logic programming languages that follow this approach [HL94, SHC96, LR91]. A proposal of introducing prescriptive types to CLP has been presented in [FP98].

Descriptive types are approximations of the denotation of a program (see e.g. [Mis84]) and are usually inferred by static analysis techniques, see e.g. [GdW92, GdW94, JB92, HCC95].

As our method has to be applicable to *existing* CLP languages which are untyped we cannot rely on prescriptive types. On the other hand an extension to CLP of the descriptive types used in [GdW92, GdW94] makes it possible to adapt for our purposes their inference techniques with the objective to support the process of writing specifications.

An Overview of the Diagnosis Tool

Figure 1.1 presents a general overview of our tool. The two main components are the **type inferencer** and the **diagnoser**. An input for type inferencer are:

1. Program **P** with an entry point **E**

2. Definition of types of external (not defined in **P**) predicates. They may be built-in predicates or predicates not yet implemented in the actual state of the program development.

The type inferencer analyzes the program **P** and generates types **T** which describe an over-approximation of the call-success semantics of **P**. When types are inferred, they are presented to the user, who decides whether they

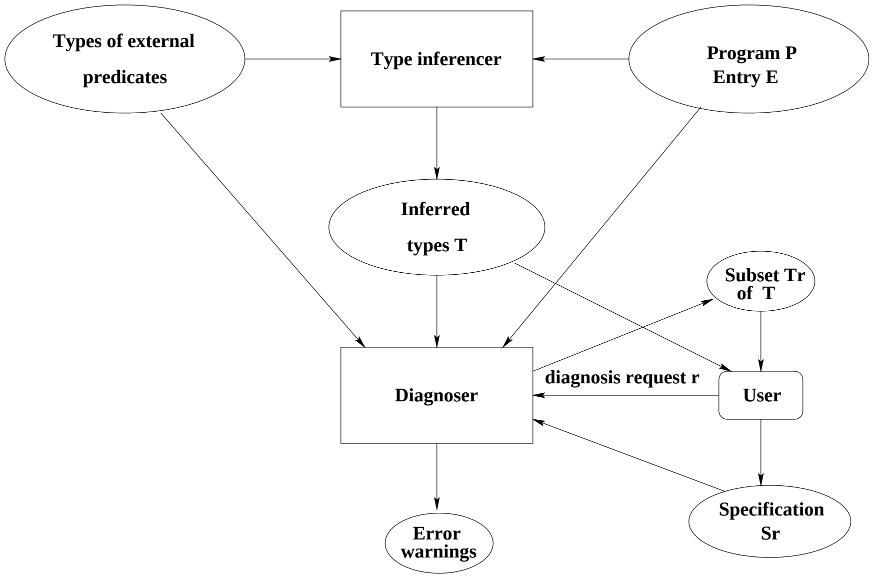

Figure 1.1: Overview of the tool

fit to her expectation or not. Then the user, after having found an unexpected type, may require (**r**) performing diagnosis of related predicate. The diagnoser selects a subset **Tr** of **T**, that is relevant for **r**. A part of the diagnosis is a dialogue, during which the user inspects and possibly modifies **Tr**. This results in a specification **Sr**, which is relevant for **r**. Taking this as input, together with the program **P**, the entry point **E** and the types for external predicates, the diagnoser generates error warnings.

1.3 The Contributions

The main contributions of the thesis are:

- A method for incorrectness diagnosis of CLP programs combining static analysis and verification.

- A prototype tool supporting this method.

- Adaptation to CLP of some concepts and techniques originating from logic programming. In particular the thesis defines:

  - The call-success semantics for CLP programs with syntactic unification.

  - An extension to CLP of the concept of regular type.

  - An extension of the method for type inference of [GdW92, GdW94] to CLP programs.

The material of this thesis was partly published in [DP98, CDMP98, CDP98, DP99]. An early attempt of type inference for CLP(FD) has been presented in [Pie97].

The thesis is organized as follows. Chapter 2 contains preliminaries on CLP and defines the semantics used in our diagnosis method. Chapter 3 gives the detailed description of the type language. In Chapter 4 the type inference algorithm is described. The diagnosis method is a subject of Chapter 6. Chapter 7 surveys some implementation issues. Final discussion is contained in Chapter 8. Appendix shows a sample diagnosis session.

Chapter 2

Constraint Logic Programming

This section presents the CLP semantics used in our work. We are concerned with correctness of CLP programs and to be precise about the concept of correctness we have to be precise about the underlying semantics.

For the general overview of CLP see [JM94, JMMS98].

2.1 CLP programs

CLP programs operate on constraint domains. A *constraint domain* is defined by providing a finite signature and a structure $\mathcal{D}$ over this signature. Predicate symbols of the signature are divided into *constraint predicates* and *non-constraint predicates*. The former have a fixed interpretation in $\mathcal{D}$, the interpretation of the latter is defined by programs. Similarly, the function symbols are divided into interpreted function symbols and constructors. All the function symbols have a fixed interpretation. It is assumed that the interpretations of constructors are bijections with disjoint co-domains. So the elements of structure $\mathcal{D}$ can be seen as (finite) terms built from some elementary values by means of constructors. That is why we will often call them $\mathcal{D}$-terms[1]. Interpretations satisfying the conditions above will be called $\mathcal{D}$-interpretations.

Notice that in many CLP languages function symbols for some their arguments play the role of constructors. For instance, the interpretation of $2+3$ may be a number, while the interpretation of $a+3$ (where a is a 0-ary constructor) is a $\mathcal{D}$-term with the main symbol $+$.

A *primitive constraint* is an atomic formula with a constraint predicate symbol. Throughout this thesis by a *constraint* we will mean a primitive

[1] Sometimes we will slightly abuse the notation and use $\mathcal{D}$ to denote the set of $\mathcal{D}$-terms.

constraint or $c_1 \wedge c_2$ or $c_1 \vee c_2$ or $\exists x c_1$, where c_1 and c_2 are constraints and x is a variable.

We write $\mathcal{D} \models c$ iff c is satisfied in every $\mathcal{D}$-interpretation and every valuation.

A CLP clause is of the form: $h \leftarrow c, b_1, \ldots, b_n$ where $h, b_1, \ldots, b_n$ are atoms (i.e. atomic formulae built up from non-constraint predicate symbols) and c is a conjunction of primitive constraints. A CLP program is a finite set of CLP clauses.

A *$\mathcal{D}$-model* for a CLP program P is $\mathcal{D}$-interpretation which is a model for P. Every program has a *least $\mathcal{D}$-model* which is the most conservative $\mathcal{D}$-model.

In many CLP applications variables range over finite sets. To express this it is convenient to introduce primitive constraint of the form $x \in S$ where x is a variable and S is an expression specifying a finite set. The constraint is satisfied if the domain of the variable x is restricted to the finite set denoted by S. CLP languages using such constraints are called CLP(FD) languages [Hen89]. The abbreviation FD stands for "finite domains" to indicate that the variables used range over finite sets. In most CLP(FD) languages the finite sets considered can only consist of non-negative integers.

2.2 CLP with syntactic unification

We want to point out an important semantic feature of CLP languages. There exist two substantially different variants of their semantics. Many actual implementations use syntactic unification: function symbols occurring outside of constraints are treated as constructors. This results in a semantics different from the standard one that is given by the least $\mathcal{D}$-models of programs. For instance, consider $\mathcal{D}$ being integers and take the program

$$\begin{array}{l} p(X) \leftarrow q(X), r(X). \\ q(2+3). \\ r(6-1). \end{array}$$

$p(5)$ is true in the least $\mathcal{D}$-model of the program. However in languages like CHIP, query $p(X)$ fails, as $2+3$ and $6-1$ are not unifiable. We will refer to CLP with the second kind of semantics as to *CLP with syntactic unification.* To characterize this semantics by the least $\mathcal{D}$-model of a program one has to use a Herbrand domain as $\mathcal{D}$. (No element of the carrier of such a domain is a value of two distinct ground terms).

In this thesis we are interested in CLP with syntactic unification.

2.3 Declarative semantics

The standard least $\mathcal{D}$-model semantics is insufficient for our purposes. We are interested in the actual form of computed answers. Two programs with the same least $\mathcal{D}$-model semantics may have different sets of computed answers. For instance take the following two CLP(FD) programs

$$P_1 = \{\, p(1).;\ p(2).\,\} \qquad P_2 = \{\, p(x) \leftarrow x \in \{1,2\}.\,\}$$

and a goal $p(x)$. Constraint $x{\in}\{1,2\}$ is an answer for P_2 but not for P_1, for which we have two answers: $x = 1$ and $x = 2$. In order to describe such differences, we generalize the c-semantics [Cla79, FLMP89]. For logic programs, this semantics is given by the set of (possibly non ground) atomic logical consequences of a program. The c-semantics for CLP will be expressed by means of constrained atoms.

Definition 2.3.1 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.

If ν is a valuation such that $\mathcal{D} \models \nu(c)$ then $\nu(E)$ is called an *$\mathcal{D}$-instance* of $c[]E$.

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 $\mathcal{D} \models c' \rightarrow c$, provided that $c\theta$ and, respectively, c' are satisfiable. 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 $\mathcal{D} \models \forall c$. Similarly, we will also not distinguish $c[]E$ from $c'[]E$ when c and c' are equivalent constraints ($\mathcal{D} \models c \leftrightarrow c'$).

Example 2.3.2 $a+7$, $Z+7$, $1{+}7$ 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 $x{\in}S$). □

Throughout the thesis we will write $Vars(E)$ to denote a set of free variables occurring in E. The notation $\exists_{-V}E$ will mean quantification over $Vars(E) \setminus V$.

If $Vars(c) \not\subseteq Vars(E)$ then $c[]E$ will denote $(\exists_{-\,Vars(E)}c)[]E$.

Two notions of groundness arise naturally for constrained expressions. $c[]E$ is *syntactically ground* when E contains no variables. $c[]E$ is *semantically ground* if it has exactly one $\mathcal{D}$-instance.

Now we define the c-semantics for CLP with syntactic unification. Let us first recall the definition of the *ordinal powers* of a function F over a complete lattice X:

$$F\uparrow\alpha = \begin{cases} \bot_X & \text{if } \alpha = 0 \\ F(F\uparrow(\alpha-1)) & \text{if } \alpha \text{ is a successor ordinal} \\ lub\{F\uparrow\beta \mid \beta<\alpha\} & \text{if } \alpha \text{ is a limit ordinal} \end{cases}$$

In the next definition we apply substitutions to program clauses. So let us define $\downarrow\!P$ as $\{\,C\theta \mid C\in P,\ \theta \text{ is a substitution}\,\}$.

Definition 2.3.3 [Immediate consequence operator for c-semantics] Let P be a CLP program. $T_P^{\mathcal{C}}$ is a mapping over sets of constrained atoms, defined by

$$\begin{aligned} T_P^{\mathcal{C}}(I) = \{\ c[]h \mid\ & (h{\leftarrow}c', b_1,\ldots,b_n) \in {\downarrow}P,\ n\geq 0, \\ & c_i[]b_i \in I,\ \ \text{for } i=1,\ldots,n, \\ & c = \exists_{-\,Vars(h)}(c', c_1,\ldots,c_n), \\ & \mathcal{D} \models \exists c \quad \} \end{aligned}$$

□

Notice that in the definition syntactic unification is used for parameter passing, but terms occurring in constraints are interpreted w.r.t. $\mathcal{D}$.

It follows from the definition that, $T_P^{\mathcal{C}}$ is continuous w.r.t. $\subseteq$. From Kleene's fixpoint theorem we know that $T_P^{\mathcal{C}}$ has the least fixpoint $lfp(T_P^{\mathcal{C}}) =$

$T_P^{\mathcal{C}} \uparrow \omega$. By the *declarative semantics* (or *c-semantics*) $M(P)$ of P we mean the instance-closure of the least fixpoint of $T_P^{\mathcal{C}}$:

$$M(P) = cl(T_P^{\mathcal{C}}\uparrow\omega).$$

Speaking informally, cl is used here only to add new constraints but not new (non-constraint) atoms: As $T_P^{\mathcal{C}}\uparrow\omega$ is closed under substitution, for every $c[]u \in M(P)$ there exists a $c'[]u \in T_P^{\mathcal{C}}\uparrow\omega$ such that $\mathcal{D} \models c \rightarrow c'$.

Example 2.3.4 Consider programs P_1 and P_2 from the beginning of this section. $M(P_1) = \{p(1), p(2)\}$. $T_{P_2}^{\mathcal{C}}\uparrow\omega$ contains $p(1)$, $p(2)$ and $x{\in}\{1,2\}[]p(x)$. $M(P_2)$ contains additionally all the instances of $x{\in}\{1,2\}[]p(x)$, like $y{=}1[]p(y)$, $z \in \{1,2\}[]p(z)$. □

The traditional least $\mathcal{D}$-model semantics and the c-semantics are related by the fact that the set of $\mathcal{D}$-instances of the elements of $M(P)$ is a subset of the least $\mathcal{D}$-model of P. If we take a least $\mathcal{D}$-model semantics for CLP with syntactic unification (where $\mathcal{D}$ is a Herbrand domain) then the set of $\mathcal{D}$-instances of the elements of $M(P)$ and the least $\mathcal{D}$-model of P coincide. We expect that c-semantics for CLP without syntactic unification can be described in a similar way. This topic is however outside of the scope of this thesis.

2.4 Call-success semantics of CLP

We are interested in the actual form of procedure calls and successes that occur during the execution of a program. We assume the Prolog selection rule. Such semantics will be called the *call-success semantics*. In this section we will also refer to a more detailed operational semantics, given by LD-resolution (SLD-resolution with the Prolog selection rule).

Without lack of generality we can restrict ourselves to atomic initial goals. Given a program and a class of initial goals, we want to provide two sets of constrained atoms corresponding to the calls and to the successes. For technical reasons that will become clear later, it is convenient to have just one set. 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 atoms whose predicate symbol is p. 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 $\{A^\bullet \mid A \in M\}$.

We assume a natural generalization of LD-resolution, with constrained goals of the form $c[]A_1, \ldots, A_n$ (where A_i are atoms) and with derivations that are sequences of constrained goals, mgu's and input clauses (similarly to [Llo87b]). For a constrained goal $G_i = c[]A_1, \ldots, A_n$ and a clause $C_{i+1} = H \leftarrow c', B_1, \ldots, B_m$ the next goal in an LD-derivation is $G_{i+1} = c''[](B_1, \ldots, B_m, A_2, \ldots, A_n)\theta$, provided that θ is an mgu of A_1 and H, constraint c'' is equivalent to $(c \wedge c')\theta$ and is satisfiable and $Vars(G_i) \cap Vars(C_{i+1}) = \emptyset$.

We adapt the definition of procedure call and success from [DM88]. If $G_i = c[]A_1, \ldots, A_n$ is a goal then $c[]A_1$ is the corresponding *procedure call.* (Remember that $c[]A_1$ is an abbreviation for $\exists_{-Vars(A_1)} c[]A_1$). If in an LD-derivation $G_0, G_1, \ldots$ with the mgu's $\theta_1, \theta_2, \ldots$ we have $G_i = c[]A_1, \ldots, A_n$ and j is the least number such that $j > i$ and $G_j = c'[](A_2, \ldots, A_n)\theta$, where $\theta = \theta_{i+1} \cdots \theta_j$, then $c'[]A_1\theta$ is the *procedure success* corresponding to the procedure call in G_i (in this LD-derivation).

Definition 2.4.1 Let P be a CLP program and $\mathcal{G}$ a set of constrained atoms. Their *call-success semantics* $CS(P, \mathcal{G})$ is a set of constrained atoms (with annotated predicate symbols) such that

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

We will characterize the call-success semantics of a program P as the declarative semantics of some other program P^{CS}. In logic programming this approach is often called "magic transformation". Program P^{CS} can also be viewed as the verification conditions of the proof method of [BC89] or an instance of the verification conditions of the proof method of [DM88].

Proposition 2.4.2 Let P be a CLP program and $\mathcal{G}$ a set of constrained atoms. Then

$$cl(CS(P, \mathcal{G})) = cl\left((T^{\mathcal{C}}_{P^{CS}})^{\omega}(\mathcal{G})\right)$$

where P^{CS} is a program that for each clause $H \leftarrow c, B_1, \ldots, B_n$ from P contains clauses:

$$\begin{array}{l}
c, {}^{\bullet}H \rightarrow {}^{\bullet}B_1 \\
\ldots \\
c, {}^{\bullet}H, B_1^{\bullet}, \ldots, B_{i-1}^{\bullet} \rightarrow {}^{\bullet}B_i \\
\ldots \\
c, {}^{\bullet}H, B_1^{\bullet}, \ldots, B_{n-1}^{\bullet} \rightarrow {}^{\bullet}B_n \\
c, {}^{\bullet}H, B_1^{\bullet}, \ldots, B_n^{\bullet} \rightarrow H^{\bullet}
\end{array}$$

PROOF (outline) One shows that all the procedure calls and successes occurring in (a prefix of) an SLD-derivation of length j are in $(T^{\mathcal{C}}_{P^{CS}})^j(\mathcal{G})$. Conversely, for any member of $(T^{\mathcal{C}}_{P^{CS}})^j(\mathcal{G})$ the corresponding call/success occurs in a derivation. Both proofs are by induction on j. □

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 constraint) 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.

Assume that the set of initial constrained goals is characterized by a CLP program P': $\mathcal{G} = \{\, A \mid {}^{\bullet}\!A \in M(P') \,\}$. Assume that no predicate $p^{\bullet}$ occurs in P'. From the last proposition it follows that the declarative semantics of $P^{CS} \cup P'$ describes the call-success semantics of P:

$$cl(CS(P, \mathcal{G})) = M(P^{CS} \cup P') \cap \mathcal{A}$$

where $\mathcal{A}$ is the set of all constrained atoms with annotated predicate symbols. (The role of the intersection with $\mathcal{A}$ is to remove auxiliary predicates that may originate from P').

Chapter 3

The specification language

We are interested in computing approximations of the call-success semantics of programs. The semantics of a program is an instance closed set of constrained atoms, an approximation is its superset. The approximations are to be computed by an analysis algorithm and communicated to the user.

We need a suitable class of approximations and a language to specify them. We extend for that purpose the formalism of regular unary logic programs [YS91] used in LP to describe regular sets of terms/atoms[1]. Following the terminology of LP we call such sets regular (constraint) types. So we use (a restricted class of) CLP programs and their declarative c-semantics to describe approximations of the call-success semantics of CLP programs.

For communication with the user we use a more terse formalism of regular term grammars with constraints (Section 3.6). The formalism provides additionally a parametric mechanism. One can define a family of types, like $list(t)$; such a definition describes the set of lists with elements of any type t.

Section 3.1 describes the class of types we employ and a way of specifying them by a certain class of CLP programs. The next section discusses some technical properties of the constraints used in these programs. These properties are applied in Section 3.3, where we present algorithms for operations on types. The operations are needed for computing semantic approximations of programs. Then we discuss our class of types from the point of view of abstract interpretation. In Section 3.5 we apply the ideas of the former sections to construct a system of types for CLP(FD). Finally we generalize the notion of regular term grammars [DZ92] to the case of constrained terms.

[1] The formalism is equivalent to deterministic root-to-frontier tree automata [GS97] and to (non parametric) regular term grammars (see e.g. [DZ92] and references therein).

3.1 Regular unary programs

Our approach to defining types is a generalization of canonical regular unary logic (RUL) programs [YS91]. We use (a restricted class of) CLP programs to approximate the semantics of (arbitrary) CLP programs.

We begin with presenting RUL programs. Then we introduce our generalization, called RULC programs. We conclude this section with several examples.

Our types will be defined by unary CLP programs. The (unary) predicates of such a program will be used as type symbols. A type denoted by a predicate is the set of constrained terms defined by the following definition which refers to the declarative semantics of the program.

Definition 3.1.1 Let P be a CLP program. Let p be a unary predicate. Then $[\![p]\!]_P := \{ \, c[]u \mid c[]p(u) \in M(P) \, \}$. □

Definition 3.1.2 A (canonical) regular unary logic program (**RUL program**) is a finite set of clauses of the form:

$$t_0(f(x_1, \ldots, x_n)) \leftarrow t_1(x_1), \ldots, t_n(x_n).$$

(where $n \geq 0$ and $x_1, \ldots, x_n$ are distinct variables) such that no two clause heads have a common instance. □

Notice that the types defined by a RUL program are sets of ground terms. (For such programs there is no difference between the c-semantics and the least Herbrand model semantics).

RUL programs were introduced in [YS91]. In [FSVY91] they are called reduced regular unary-predicate programs. The formalism defines tuple distributive [Mis84, YS91] sets of terms. So if $f(u_1, u_2)$ and $f(u'_1, u'_2)$ are members of such a set then also $f(u_1, u'_2)$ and $f(u'_1, u_2)$ are. (For exact definitions the reader is referred to [Mis84, YS91]).

Before introducing a CLP generalization of RUL programs we need some definitions. A clause with predicate symbol p in its head will be called a clause *defining* p. A predicate p in a program P *depends* on a predicate q if $p = q$ or in the r.h.s. of a clause defining p there occurs a predicate p' which depends on q. (Formally, relation "depends on" is the least relation satisfying these conditions). A clause is *relevant* for p if it defines q and p depends on q. We will write $F[x_1, \ldots, x_n]$ to stress that F is a formula such that $Vars(F) \subseteq \{x_1, \ldots, x_n\}$. $F[u_1, \ldots, u_n]$ will denote F with each x_i replaced by the term u_i.

Definition 3.1.3 A constraint $c[x]$ in a constraint domain $\mathcal{D}'$ will be called a **regular constraint** if there exists a RUL program R and a predicate symbol t such that for any ground term u, $\mathcal{D}' \models c[u]$ iff $u \in [\![t]\!]_R$. Constraint c will be called the **corresponding constraint** for t and R. Conversely, program R will be called a **corresponding program** for c and t, provided that all the clauses of R are relevant for t. □

Notice that if $\mathcal{D}'$ is a Herbrand domain then the corresponding constraint for a RUL program is regular. This may not be the case for a non Herbrand domain. For instance consider domain $\mathcal{D}'$ of integers, where $+$ is an interpreted function symbol. So terms $1+3$ and 4 denote the same value in $\mathcal{D}'$. Take a program $R = \{t(4).\}$. The corresponding constraint should be satisfied by $1+3$ and by $3+1$ but not by $3+3$. So the set of terms for which it is satisfied cannot be described by a RUL program.

The next definition provides a CLP generalization of RUL programs. From now on we assume that the constraint domain $\mathcal{D}$ contains the regular constraints.

Definition 3.1.4 By an *instance of the head* of a clause $h \leftarrow c, b_1, \ldots, b_n$ (where c is a constraint and $b_1, \ldots, b_n$ are non constraint atoms) we mean an instance of $c[]h$.

A regular unary constraint logic program (**RULC program**) is a finite set of clauses of the form:

$$t_0(f(x_1, \ldots, x_n)) \leftarrow t_1(x_1), \ldots, t_n(x_n). \tag{3.1}$$

(where $n \geq 0$, $x_1, \ldots, x_n$ are distinct variables) or of the form

$$t_0(x) \leftarrow c[x]. \tag{3.2}$$

(where $c[x]$ is a regular constraint) such that no two clause heads have a common instance.

□

Example 3.1.5

The type t described by the RUL program $\{\, t(2).,\ t(3).,\ t(4).\,\}$ is the set $\{2, 3, 4\}$ of ground terms.

Consider CLP(FD) [Hen89]. Let S be a finite set of integers. Assume that we want to describe a type containing a domain variable with S as its domain. (It is natural that the type also contains all possible instantiations of the variable). To do this, we use a regular constraint $x{\in}S$ in a RULC program $R' = \{\, t'(x) \leftarrow x{\in}S \,\}$. Indeed, $[\![t']\!]_{R'} = cl(x{\in}S \,[]\, x)$ contains the constants corresponding to the elements of S and the constrained terms of the form $x{\in}S' \,[]\, x$, where $S' \subseteq S$. □

Example 3.1.6 A type of lists with (possibly non-ground) elements satisfying a constraint c can be expressed by the following RULC program R:

$$\begin{array}{l} list([\,]) \leftarrow . \\ list([x|xs]) \leftarrow elem(x), list(xs). \\ elem(x) \leftarrow c[x]. \end{array}$$

The c-semantics of this program is
$M(R) = cl\left(\{\, c[x_1], \ldots, c[x_n] \,[]\, list([x_1, \ldots, x_n]) \mid n \geq 0 \,\} \cup \{\, c[x][]elem(x) \,\}\right)$.

Let Q be a corresponding (RUL) program for $c[x]$ and $elem$. Replacing in R the last clause by (the clauses of) Q results in a RUL program R' describing the set of ground lists from the previous type.

Let $c_{list}[x]$ be the corresponding constraint for $list$ and R'. A type of possibly non-ground lists with elements of the type $elem$ can be defined by a one clause RULC program R''

$$list(x) \leftarrow c_{list}[x].$$

The c-semantics of this program is $M(R'') = cl(\, c_{list}[x][]list(x)\,)$. Notice that the type $list$ (i.e. the set $[\![list]\!]_{R''} = cl(c_{list}[x][]x))$ contains unbound variables whose further bindings are restricted to be lists.[2] Thus our approach makes it possible to express prescriptive types like those of programming language Gödel [HL94].

Comparing the three list types presented here, we obtain $[\![list]\!]_{R'} \subseteq [\![list]\!]_{R} \subseteq [\![list]\!]_{R''}$.

□

Example 3.1.7 The type of all ground terms (over the given signature) is defined by predicate $ground$ and a (RUL) program containing the clause $ground(f(x_1, \ldots, x_n)) \leftarrow ground(x_1), \ldots, ground(x_n)$ for each function symbol f of arity $n \geq 0$.

The type of all constrained terms is defined by predicate any and program $\{\, any(x) \leftarrow true \,\}$. Notice that true is a regular constraint as it corresponds to predicate ground in the previous program.

□

3.2 On regular constraints

In this section we discuss some properties of regular constraints and corresponding RUL programs, which will be used later.

[2] It also contains, for instance, open lists with such variables as their tails (i.e. terms of the form $c_{list}[y][][u_1, \ldots, u_n|y]$ where y is a variable and $u_1, \ldots, u_n$ are possibly non-ground terms of type $elem$).

Let $c[x]$ be a regular constraint and R be a corresponding program for c and t. Consider the constraint $c[f(x_1,\ldots,x_n)]$ (obtained by replacing in c the variable x by the term $f(x_1,\ldots,x_n)$), where $x_1,\ldots,x_n$ are distinct variables. Assume that $c[f(x_1,\ldots,x_n)]$ is satisfiable. So there must exist a clause $t(f(x_1,\ldots,x_n)) \leftarrow t_1(x_1),\ldots,t_n(x_n)$ in R, and the clause is unique. Consider constraints

$$c_i[x_i] := \exists_{-\{x_i\}}\, c[f(x_1,\ldots,x_n)]$$

($1 \leq i \leq n$). We have $u_1 \in [\![t_1]\!]_R$ iff $f(u_1,\ldots,u_n) \in [\![t]\!]_R$ for some $u_2,\ldots,u_n$ iff $\mathcal{D} \models c[f(u_1,\ldots,u_n)]$ for some $u_2,\ldots,u_n$ iff $\mathcal{D} \models c_1[u_1]$. (Here $u_1,\ldots,u_n$ are ground terms). The same reasoning holds for $u_2,\ldots,u_n$. Thus c_i is the corresponding constraint for t_i and R, for $i = 1,\ldots,n$.

Now consider an arbitrary term u and the constraint $c[u]$. Assume that $c[u]$ is satisfiable and that $y \in \mathit{Vars}(u)$. From the previous property, by induction on the depth of u, we obtain that

$$\exists_{-\{y\}}\, c[u]$$

is the corresponding constraint for some predicate t' in R. So for a given y, the class of constraints of this form modulo equivalence is finite. (Constraints c_1 and c_2 are equivalent iff $\mathcal{D} \models c_1 \leftrightarrow c_2$). Moreover, the corresponding constraint for any t' in R is (equivalent to one) of this form.

3.3 Operations on RULC programs

Now we discuss basic algorithms for types described by RULC programs. We discuss membership check, checks for type emptiness and type inclusion, and computing the intersection and (an approximation of) the union of types. These operations will be employed in type analysis in section 4.

Let R be an RULC program. A method of checking whether a ground term u is in a type $[\![t]\!]_R$ is obvious (and is linear time w.r.t. the size of the term), provided that an algorithm to check whether a ground term satisfies a regular constraint is given (and is linear time).

Checking if a constrained term is in $[\![t]\!]_R$ can be based on the following property. A constrained variable $c[]x \in [\![t]\!]_R$ iff there exists a clause $t(y) \leftarrow c'[y]$ in R such that $\mathcal{D} \models c \to c'[x]$. For non variable constrained terms, $c[]f(u_1,\ldots,u_n) \in [\![t]\!]_R$ iff there exists a clause $t(f(x_1,\ldots,x_n)) \leftarrow t_1(x_1),\ldots,t_n(x_n)$ in R and $c[]u_i \in [\![t_i]\!]_R$ for $i = 1,\ldots,n$, or there exists $t(y) \leftarrow c'[y]$ in R such that $\mathcal{D} \models c \to c'[f(u_1,\ldots,u_n)]$. The resulting algorithm is also linear time provided that the check for $\mathcal{D} \models c \to c'$ is linear time.

The set of empty types in an RULC program R can be computed as follows. Mark as empty every predicate t which contains no clauses in its definition or contains only clauses with unsatisfiable constraints. Then mark as empty each predicate t such that all clauses defining t contain at least one predicate marked as empty; until no new marks can be added. Now a predicate t is marked iff $[\![t]\!]_R = \emptyset$.

Consider a RULC program R. We show how to construct a RUL program $ground(R)$ that defines the ground subsets of the types defined by R.

Definition 3.3.1 Let R be a RULC program. Let $R_c = \{ t_i(x) \leftarrow c_i[x] \mid i = 1, \dots, n \}$ be those clauses of R that contain constraints. Let (for $i = 1, \dots, n$) R_i be a corresponding program for c_i and t_i, such that any predicate symbol distinct from $t_1, \dots, t_n$ occurs in at most one of the programs $R, R_1, \dots, R_n$ (and that t_i does not occur in R_j, for $i \neq j$). Then

$$ground(R) := (R \setminus R_c) \cup R_1 \cup \dots \cup R_n.$$

□

It is easy to see that $ground(R)$ is a RUL program and that $[\![t]\!]_{ground(R)}$ is the set of ground terms from $[\![t]\!]_R$, for any predicate t.

Definition 3.3.2 [Inclusion] Let R_1, R_2 be RULC programs (not necessarily distinct). Let $\sqsubseteq$ be a relation on $PP \times RR$ where RR is a finite set of programs, containing $R_1, R_2, ground(R_1), ground(R_2)$ and the programs (corresponding to the constraints of R_1, R_2) used in the construction of $ground(R_1), ground(R_2)$. PP is the set of predicates occurring in RR. The relation $\sqsubseteq$ is defined as the greatest relation such that for any predicate symbols t_1, t_2 from respectively R_1, R_2, $(t_1, R_1) \sqsubseteq (t_2, R_2)$ iff

1. for every clause $t_1(f(x_1, \dots, x_n)) \leftarrow r_1(x_1), \dots, r_n(x_n) \in R_1$, $n \geq 0$,
   (a) there is a clause $t_2(f(x_1, \dots, x_n)) \leftarrow s_1(x_1), \dots, s_n(x_n) \in R_2$ such that $(r_i, R_1) \sqsubseteq (s_i, R_2)$ for $1 \leq i \leq n$, or
   (b) there is a clause $t_2(x) \leftarrow c[x] \in R_2$ and a clause $t_2(f(x_1, \dots, x_n)) \leftarrow s_1(x_1), \dots, s_n(x_n) \in R_c$, where R_c is a corresponding program for $c[x]$ and t_2, and $(r_i, ground(R_1)) \sqsubseteq (s_i, R_c)$ for $1 \leq i \leq n$.
2. For every clause of the form $t_1(x) \leftarrow c_1[x] \in R_1$ there is a clause $t_2(x) \leftarrow c_2[x] \in R_2$ such that $(t_1, R_{c_1}) \sqsubseteq (t_2, R_{c_2})$, where R_{c_j} is a corresponding program for c_j and t_j ($j = 1, 2$).

□

We will sometimes abbreviate $(t_1, R_1) \sqsubseteq (t_2, R_2)$ to $t_1 \sqsubseteq t_2$. The next proposition shows that relation $\sqsubseteq$ indeed corresponds to relation $\subseteq$ between types:

Proposition 3.3.3 For RULC programs R, R' and predicates t, t', if $(t, R) \sqsubseteq (t', R')$ then $[\![t]\!]_R \subseteq [\![t']\!]_{R'}$ (i.e. t is a subtype of t').

If $[\![t]\!]_R \subseteq [\![t']\!]_{R'}$ and t does not depend on an empty type then $(t, R) \sqsubseteq (t', R')$. □

The main idea of the definition 3.3.2 is that for every ground element in $[\![t_1]\!]_{R_1}$ there is ground or non-ground corresponding element in $[\![t_2]\!]_{R_2}$. If $[\![t_1]\!]_{R_1}$ contains non-ground elements, i.e. there is a clause $t_1(x) \leftarrow c[x]$ in R_1 then there must be a corresponding clause in R_2.

Example 3.3.4 Consider programs P_1

$$\begin{array}{l} list_1([\,]). \\ list_1([x_1|x_2]) \leftarrow t(x_1), list_1(x_2). \\ t(a). \end{array}$$

and P_2

$$\begin{array}{l} list_2([\,]). \\ list_2([x_1|x_2]) \leftarrow r(x_1), list_2(x_2). \\ r(x) \leftarrow c[x]. \end{array}$$

and the program corresponding to c

$$s(a).$$

The type $list_1$ describes ground lists of as, whereas list of the type $list_2$ may contain ground elements and variables (constrained to a). The reader may check that $(list_1, P_1) \sqsubseteq (list_2, P_2)$. □

Definition 3.3.5 [Intersection] Let R_1, R_2 be RULC programs. We construct a RULC program R where for each pair t_1, t_2 of predicates of respectively R_1, R_2 a new predicate $t_1 \sqcap t_2$ is defined.

Take a variable y. For each constraint $c[x]$ occurring in R_i $(i = 1, 2)$ and any term u such that $y \in \mathit{Vars}(u)$ consider the constraint $\exists_{-\{y\}}\, c[u]$. As shown in Section 3.2, the set of such constraints is finite (when we do not distinguish between equivalent constraints). For each such constraint add to R_i a clause $t_u(y) \leftarrow \exists_{-\{y\}}\, c[u]$, where t_u is a new predicate symbol, obtaining a RULC program R'_i.

1. if $t_1(f(x_1, \ldots, x_n)) \leftarrow r_1(x_1), \ldots, r_n(x_n) \in R_1$ and $t_2(f(x_1, \ldots, x_n)) \leftarrow s_1(x_1), \ldots, s_n(x_n) \in R_2$ then R contains a clause $(t_1 \sqcap t_2)(f(x_1, \ldots, x_n)) \leftarrow (r_1 \sqcap s_1)(x_1), \ldots, (r_n \sqcap s_n)(x_n)$.

2. If
$$\begin{array}{l} t_i(f(x_1,\ldots,x_n)) \leftarrow r_1(x_1),\ldots,r_n(x_n) \in R_i \\ t_j(x) \leftarrow c[x] \in R'_j \end{array}$$
(where $\{i,j\} = \{1,2\}$) and if $c[f(x_1,\ldots,x_n)]$ is satisfiable then R contains clause
$$(t_1 \sqcap t_2)(f(x_1,\ldots,x_n)) \leftarrow (r_1 \sqcap s_1)(x_1),\ldots,(r_n \sqcap s_n)(x_n).$$
where clauses
$$\begin{array}{l} s_1(x_1) \leftarrow \exists_{-\{x_1\}}\, c[f(x_1,\ldots,x_n)]. \\ \ldots \\ s_n(x_n) \leftarrow \exists_{-\{x_n\}}\, c[f(x_1,\ldots,x_n)]. \end{array}$$
occur in $R'_j \setminus R_j$.

3. If
$$\begin{array}{l} t_1(x) \leftarrow c_1[x] \in R'_1 \\ t_2(x) \leftarrow c_2[x] \in R'_2 \end{array}$$
then R contains a clause
$$(t_1 \sqcap t_2)(x) \leftarrow c_1[x], c_2[x].$$

□

The obtained program is a RULC program, as the conjunction $c_1[x], c_2[x]$ of regular constraints is a regular constraint. (This follows for instance from applying the following proposition to RUL programs).

Proposition 3.3.6 For R_1, R_2, R, t_1, t_2 and $t_1 \sqcap t_2$ as in the last definition,
$$[\![t_1 \sqcap t_2]\!]_R = [\![t_1]\!]_{R_1} \cap [\![t_2]\!]_{R_2}$$

□

The construction of an intersection of two RULC is rather straightforward. The interesting case is when a clause of the form $t_1(f(x_1,\ldots,x_n)) \leftarrow r_1(x_1),\ldots,r_n(x_n)$ meets a clause with a constraint i.e. $t_2(x) \leftarrow c[x]$ (case 2). Then each element of $[\![t_1 \sqcap t_2]\!]_R$ originating from this case is of the form $c'[]f(u_1,\ldots,u_n)$, where $\mathcal{D} \models c' \rightarrow c[f(u_1,\ldots,u_n)]$.

Example 3.3.7 Consider RULC programs P_1

$$\begin{array}{l} list_1([\,]). \\ list_1([x_1|x_2]) \leftarrow t(x_1), list_1(x_2). \\ t(a). \\ t(b). \end{array}$$

and P_2

$$\begin{array}{l} list_2([\,]). \\ list_2([x_1|x_2]) \leftarrow r(x_1), list_2(x_2). \\ r(x) \leftarrow c[x]. \end{array}$$

and the program corresponding to c

$$s(a).$$

The intersection $list_1 \sqcap list_2$ is defined by the following RULC program

$$\begin{array}{l} (list_1 \sqcap list_2)([\,]). \\ (list_1 \sqcap list_2)([x_1|x_2]) \leftarrow (t \sqcap r)(x_1), (list_1 \sqcap list_2)(x_2). \\ (t \sqcap r)(a). \end{array}$$

□

Definition 3.3.8 [Upper bound] Let R_1, R_2 be RULC programs. We construct a new RULC program R where, for each pair t_1, t_2 of predicates defined in R_1 and R_2 respectively, a new predicate $t_1 \sqcup t_2$ is defined. We will sometimes write $(t_1 \sqcup t_2, R) = (t_1, R_1) \sqcup (t_2, R_2)$ to make it explicit to which program each of the predicates belongs.

We say that RULC clauses C, C' (defining t and t' respectively) *overlap* if, for some term w, $t(w)$ and $t'(w)$ are instances of the heads of C and C' respectively, in the sense of Definition 3.1.4.

1. If

$$t_i(u) \leftarrow B \in R_i$$

   and this clause does not overlap with any clause defining t_j (where $\{i, j\} = \{1, 2\}$) then

$$(t_1 \sqcup t_2)(u) \leftarrow B \in R$$

   and each clause of R_i relevant for a predicate occurring in B is in R.

2. If
$$t_1(f(x_1,\ldots,x_n)) \leftarrow q_1(x_1),\ldots,q_n(x_n) \in R_1$$
$$t_2(f(x_1,\ldots,x_n)) \leftarrow r_1(x_1),\ldots,r_n(x_n) \in R_2$$
then
$$(t_1 \sqcup t_2)(f(x_1,\ldots,x_n)) \leftarrow (q_1 \sqcup r_1)(x_1),\ldots,(q_n \sqcup r_n)(x_n) \in R.$$

3.
- Collect in $R'_1 \subseteq R_1$ and $R'_2 \subseteq R_2$ all the remaining clauses defining respectively t_1 and t_2 (i.e. those that do not satisfy the conditions of case 1 and 2 above). Thus for each clause $C \in R'_1$ there exits at least one clause $C' \in R'_2$ (and vice versa) such that C and C' overlap and at least one of C and C' is a clause with a constraint.
- for each predicate q occurring in a clause body in R'_i add to R'_i the clauses of R_i relevant for q, obtaining R''_i $(i = 1, 2)$,
- compute $(t_1 \sqcup t_2, R'') = (t_1, ground(R''_1)) \sqcup (t_2, ground(R''_2))$ and let c be the constraint corresponding to $(t_1 \sqcup t_2)$ in R'' (notice that it is a RUL program),
- then
$$(t_1 \sqcup t_2)(x) \leftarrow c[x] \in R.$$

□

Proposition 3.3.9 If $(t_1 \sqcup t_2, R) = (t_1, R_1) \sqcup (t_2, R_2)$ then
$$[\![t_1]\!]_{R_1} \cup [\![t_2]\!]_{R_2} \subseteq [\![t_1 \sqcup t_2]\!]_R$$

□

Notice that the operation $\sqcup$ may add non-ground elements which are not present in the denotations of the involved programs. These new elements may be created by case 3 and by case 2 of the definition. For example, the upper bound of $\{f(a,b)\}$ and $\{f(c,d)\}$ which is $\{f(a,b), f(a,d), f(c,b), f(c,d)\}$., is tuple-distributive due to the case 3 of the definition.

Example 3.3.10 Consider program P_1 and P_2 from the Example 3.3.7. The upper bound $list_1 \sqcup list_2$ is given by the following RULC program

$$(list_1 \sqcup list_2)([\,]).$$
$$(list_1 \sqcup list_2)([x_1|x_2]) \leftarrow (t \sqcup r)(x_1), (list_1 \sqcup list_2)(x_2).$$
$$(t \sqcup r)(b).$$
$$(t \sqcup r)(x) \leftarrow c[x].$$

The program RULC corresponding to c has the following form
$$st(a).$$

□

3.4 Regular programs as an abstract domain

In this section we present how RULC programs can be treated as approximations of CLP programs. This is a straightforward extension of the ideas of [GdW92, GdW94]. It seems natural to view such an approach as abstract interpretation. This is however a rather unusual case of abstract interpretation. We show that most of the usually required conditions [CC92a] are not satisfied (neither by the approach of [GdW92, GdW94] nor by our generalization). This contradicts some claims of [GdW92, GdW94]. In particular, the abstract domain is not partially ordered, the abstraction function does not exist and the abstract semantics function is not monotonic, hence (although it has a fixpoint) a least fixpoint may not exist.

In our approach, the concrete domain $\mathbf{C}$ is that used by the c-semantics of programs. So $\mathbf{C}$ is the set of sets of constrained atoms over the given language. (We do not need to make the domain more sophisticated by removing from $\mathbf{C}$ those elements that are not the meaning of any program). $(\mathbf{C}, \subseteq)$ is a complete lattice.

For a given CLP program we want to approximate its semantics by a RULC program. So we have to relate somehow the semantics of both programs. Following [GdW92, GdW94] we introduce a distinguished (unary) predicate symbol *approx*. The set (unary relation) corresponding to *approx* is understood as the set of constrained atoms specified by the RULC program.

Definition 3.4.1 Let P be a CLP program and R a RULC program. Let I be a set of constrained atoms. Then R is a *regular approximation* of I if $I \subseteq [\![approx]\!]_R$. □

Notice that the arguments of *approx* are treated both as atoms and as terms, we use here the ambivalent syntax [AB96].

Example 3.4.2 Let P be the following CLP(R) program

$$rev([\,], Y, Y).$$
$$rev([f(V,X)|T], Y, Z) \leftarrow V{*}V{+}X{*}X < 9,\ rev(T, Y, [f(V,X)|Z]).$$

Then the following program is a regular approximation of $M(P)$.

$$approx(rev(X,Y,Z)) \leftarrow t1(X), any(Y), any(Z).$$
$$t1([\,]).$$
$$t1([X|Xs]) \leftarrow t2(X),\ t1(Xs).$$
$$t2(f(X,Y)) \leftarrow t3(X),\ t3(Y).$$
$$t3(X) \leftarrow -3 < X,\ X < 3.$$

□

So the abstract domain $\mathbf{A}$ is the set of RULC programs (over the given language). The concretization function $\gamma : \mathbf{A} \to \mathbf{C}$ is defined as the meaning of *approx*:

$$\gamma(R) = [\![approx]\!]_R.$$

This and the ordering of the concrete domain induces the relation $\preceq$ on $\mathbf{A}$:

$$R \preceq R' \text{ iff } \gamma(R) \subseteq \gamma(R').$$

$\preceq$ is a pre-order but not a partial order. A pre-order generates in a standard way an equivalence relation $\cong$ and a partial ordering of the quotient set. However, even taking quotient set $\mathbf{A}/_{\cong}$ as the abstract domain we do not avoid the next problem. Namely, an abstraction function *does not exist*. (This holds already in the case of logic programs and the approach of [GdW92, GdW94]).

In the example below we show that there does not exist an abstraction function $\alpha : \mathbf{C} \to \mathbf{A}$, which is monotonic and $\alpha\gamma(Q) = Q$, for any Q. It is sufficient to consider logic programs.

Example 3.4.3 Consider a program P.

$$\begin{array}{l} p(s(0), 0). \\ p(s(s(N)), NM) \leftarrow p(N, M), plus(N, M, NM). \\ plus(0, Y, Y). \\ plus(s(X), Y, s(Z)) \leftarrow plus(X, Y, Z). \end{array}$$

The semantics of p in P is $[\![p]\!]_P = \{\, p(s^{2i+1}(0), s^{i^2}(0)) \mid i \geq 0 \,\}$. So the type of the second argument of p is not regular. There exists an infinite sequence $Q_1 \succeq Q_2 \succeq \cdots$ of pairwise non-equivalent regular approximations of $M(P)$, where Q_i is:

$$\begin{array}{l} approx(p(X, Y)) \leftarrow t(X), u(Y). \\ approx(plus(X, Y, Z)) \leftarrow \ldots. \\ \ldots \end{array}$$

such that the semantics of u is $[\![u]\!]_{Q_i} = \{\, 0, s^{1^2}(0), \ldots, s^{i^2}(0) \,\} \cup \{\, s^j(r) \mid j > i^2 \,\}$ So there does not exist a best regular approximation of $M(P)$, hence the abstraction function does not exists.

More precisely: Assume that there exists a monotonic abstraction function $\alpha : \mathbf{C} \to \mathbf{A}$ such that $\alpha\gamma(Q) = Q$, for any Q. Let $\alpha(M(P)) = R$. Program R specifies a regular set of terms as the type of the second argument of p. Thus this set contains a term r, which is not of the form $s^{i^2}(0)$. On the other hand, there exists a k such that $r \notin [\![u_k]\!]_{Q_k}$. So there exists a regular program R' such that $\gamma(R') = \gamma(R) \cap \gamma(Q_k)$, moreover $R' \preceq R$ and $R' \not\cong R$. Now, as $M(P) \subseteq \gamma(R')$, we have $R = \alpha(M(P)) \preceq \alpha\gamma(R') = R'$, contradiction. $\square$

We want to mention that the abstract immediate consequence function $T_P^{\mathcal{A}}$, defined later on and used in type inference, may be not monotonic and that $\cong$ may be not a congruence w.r.t. $T_P^{\mathcal{A}}$.[3] The same holds for the analogical function of [GdW92, GdW94]. Also, using another natural pre-order on $\mathbf{C}$ ($R \sqsubseteq R'$ iff $M(R) \subseteq M(R')$) does not improve the properties discussed in this section.

3.5 Types for CLP(FD)

In our type analysis for CHIP we use some types that correspond to restrictions on the form of arguments of finite domain constraint predicates. We need the type of natural numbers, the type of integers, the type of finite domains (the l.u.b. of the types of the form $cl(x{\in}S[\,]x))$, the type of arithmetic expressions and its subset of so called linear terms.

Defining the first three of them by a RULC program requires a constant *maxint*, which is the maximal integer value[4].

We introduce unary predicate symbols *nat*, *neg* and *anyfd*. The type *nat* is defined by the following RULC program:

$$\begin{array}{l} nat(0).\\ nat(1).\\ \vdots\\ nat(maxint). \end{array}$$

So $[\![nat]\!]$ is the set of all non-negative integer constants. Similarly $[\![neg]\!]$ is the set of all negative integer constants, as it is defined by the following RULC program:

$$\begin{array}{l} neg(-1).\\ neg(-2).\\ \vdots\\ neg(-maxint). \end{array}$$

The type *anyfd* is defined as follows:

$$anyfd(x) \leftarrow c_{nat}[x].$$

where c_{nat} is a constraint corresponding to *nat*.

Notice that in according to the definition 3.3.5 $nat \sqcap anyfd = nat$. This is not surprising as $[\![nat]\!]$ is a set of constants, and $[\![anyfd]\!]$ contains both constants and domain variables.

[3] Relation $\cong$ is a congruence w.r.t. a function f iff $f(x) \cong f(y)$ whenever $x \cong y$.

[4] So we assume that type of integers if finite.

The type *int* of integers and the type of arithmetic expressions are defined by means of the above types by a RULC program. The type of linear terms cannot be defined by a RULC program. (For instance, for domain variables X, Y and a natural number n, it contains $X * n$ and $n * Y$ but not $X * Y$). So we use a RULC description of a superset of it.

3.6 Regular term grammars

The formalism of RULC programs is not very convenient as a tool of communication between the user and the analysis system. The formalism of regular term grammars of [DZ92] is much better for that purpose but it can only describe sets of terms. In this section we define its extension that can describe sets of constrained terms. In addition we equip it with parameters so that a kind of parametric types can be defined. The user may provide to the system some (possibly parametric) type declarations. Whenever possible, the system uses the declared types in its output. For instance assume that the user has described a type $list(\alpha)$, with the expected meaning. (The details are given in Example 3.6.2 below). Assume also that the system derives a type t with the corresponding fragment of a RULC program

$$\begin{array}{l} t([\,]). \\ t([x|y]) \leftarrow nat(x), t(y). \end{array}$$

Then, instead of displaying the RULC clauses (or actually a corresponding grammar) the system informs that the type is $list(nat)$.

In this section we describe a restricted class of regular term grammars with constraints. The class is suitable for CLP(FD) and is used in our analysis system. It corresponds to the class of RULC programs described in the previous section.

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 $\mathcal{TV}$. $\mathcal{T}_0 = \{\, any, anyfd, nat, neg \,\} \subseteq \mathcal{T}$ is a set of distinguished type symbols of arity 0. As we are going to specify sets of constrained terms the alphabet includes also constraint predicate $\in$.

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.

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.

Definition 3.6.1 We consider *rules* of the form

$$t(\alpha_1, \ldots, \alpha_n) \to f(t_1, \ldots, t_k)$$
$$t(\alpha_1, \ldots, \alpha_n) \to c[x]$$

where $t \in \mathcal{T} \setminus \mathcal{T}_0$ and is of arity $n \geq 0$, $\alpha_1, \ldots, \alpha_n$ are distinct type variables, $f \in F$ and is of arity $k \geq 0$ and $t_1, \ldots, t_k \in Term(\mathcal{T}, \{\alpha_1, \ldots, \alpha_n\})$.

A *ground type substitution* is a mapping from type variables to ground type terms. A *ground rule* is the instance of a rule under a ground type substitution.

The RULC clause *corresponding* to ground rule $s \to f(s_1, \ldots, s_k)$ is $s(f(x_1, \ldots, x_k)) \leftarrow s_1(x_1), \ldots, s_k(x_k)$. The RULC clause *corresponding* to ground rule $s \to c[x]$ is $s(x) \leftarrow c[x]$. (Ground type terms play here the role of predicate symbols).

Let R be a finite set of rules. Let R' be the set of ground rules that are instances of those from R. Consider the set Q of RULC clauses corresponding to the rules of R'. R is a parametric **regular term grammar** with constraints (in normal form) if

1. any finite subset of Q is a RULC program and
2. for any ground type term s the set Q_s of those clauses of Q that are relevant for s is finite.

The set (of constrained terms over $Term(F, V)$) *generated* by a ground type term s in the grammar R is the set $[\![s]\!]_R := [\![s]\!]_{Q_s}$. □

The first of the two conditions means that for no two ground instances of rules of R the corresponding RULC clauses have a common instance of their heads. The second condition means that any s depends on a finite set of predicates of Q. We may point out similarity of the second condition to the "reflexive condition" used by [SHK97] (in the context of prescriptive types). The sets generated by the grammar may be also defined in a natural way using a notion of a derivation. This is however outside of the scope of the present thesis.

Example 3.6.2

Let $t1$ be a type constant, *list* a unary type constructor and let α be a type variable. Let $a, b, [\,], | \in F$. We may define the following term grammar (where we use the standard Prolog list notation).

$$\begin{array}{l} list(\alpha) \to [\,] \\ list(\alpha) \to [\alpha|list(\alpha)] \\ t1 \to a \\ t1 \to b \end{array}$$

The RULC clauses corresponding to the last two rules are $t1(a)\leftarrow$ and $t1(b)\leftarrow$. The clauses corresponding to (the ground instances of) the first two rules are of the form $list(s)([\,])\leftarrow$ and $list(s)([x|y]) \leftarrow s(x),\, list(s)(y)$, where s is an arbitrary ground type term. □

3.7 Discussion

A notion of regular types has been introduced to logic programming by Mishra [Mis84]. The basic idea is to describe a (possibly infinite) set of ground terms using finite representation. Regular types have been used by many authors later. See for instance [YS91, FSVY91, DZ92, PR89, Zob87].

Regular types can be defined by (non-parametric) regular term grammars [DZ92] or by restricted class of logic programs [FSVY91, YS91, GdW92, GdW94]. They are both equivalent to deterministic root-to-frontier tree automata [GS97].

In most cases regular types obey *tuple distributivity* restriction. The system of types which may not be tuple distributive, described by Dart and Zobel [DZ92] is more expressive. The recent paper by Lu and Cleary [LC98] shows incorrectness of the algorithm for testing type inclusion, presented in [DZ92][5]. Nevertheless, it seems that tuple distributive regular types are expressive enough to describe commonly used data structures.

Another issue is how regular sets of terms are used in the context of typing in logic programming. Often they are used to denote a superset of declarative semantics of the program [YS91, FSVY91, DZ92, Zob87]. They are also used to describe modes [RNP92] and directional types [Boy96].

[5] Problem of correctness and completeness of this algorithm is left in [DZ92] as an open question.

Chapter 4

Constructing regular approximations of CLP programs

Inferring descriptive types means computing an approximation of the semantics of a given program. In our approach the approximations are expressed as RULC programs. We are mainly interested in the call-success semantics. However we use its characterization by means of the declarative semantics of another program (Section 2.4). So the core of the method is computing a regular approximation of the declarative semantics.

Computing a regular approximation can be seen as a bottom-up abstract interpretation which for a given program P gives an over-approximation of $M(P)$. Our approach is based on [GdW92, GdW94]. We use a function $T_P^{\mathcal{A}} : \mathbf{A} \to \mathbf{A}$, which approximates the immediate consequence operator $T_P^{\mathcal{C}}$. The program semantics $M(P)$ is approximated by a fixpoint of $T_P^{\mathcal{A}}$. A technique similar to widening [CC92a] is applied to assure that a fixpoint is reached in a finite number of steps. (It is called normalization and shortening in [GdW92] and [GdW94] respectively).

To describe the algorithm, we first present approximating a constraint, then approximating a clause and then computing $T_P^{\mathcal{A}}$.

4.1 Approximating a constraint

Suppose that we want to approximate a constraint $c[x_1, \ldots, x_n]$ (with the free variables $\{x_1, \ldots, x_n\}$) in a program clause:

$$h \leftarrow c[x_1, \ldots, x_n], b_1, \ldots, b_m.$$

Assume that some type information about the predicates occurring in $b_1, \ldots, b_m$ has been already inferred and it is given by a RULC program R. Let $r_1, \ldots, r_n$ be types defined by R. The intention is that the values of $x_1, \ldots, x_n$ are restricted to types $r_1, \ldots, r_n$ respectively. Assume that $c_{r_1}[x_1], \ldots, c_{r_n}[x_n]$ are regular constraints corresponding to $r_1, \ldots, r_n$ in R (formally, in $ground(R)$).

If $c_1[x_1], \ldots, c_n[x_n]$ are regular constraints such that

$$\mathcal{D} \models c[x_1, \ldots, x_n], c_{r_1}[x_1], \ldots, c_{r_n}[x_n] \rightarrow c_1[x_1], \ldots, c_n[x_n]$$

then $t_1(x_1), \ldots, t_n(x_n)$, together with a RULC program $\{\, t_i(x_i) \leftarrow c_i[x_i] \mid i = 1, \ldots, n \,\}$, is called a **regular approximation** of $c[x_1, \ldots, x_n]$ with types $r_1, \ldots, r_n$ for variables $x_1, \ldots, x_n$.

Computing such an approximation depends on a constraint domain and on its implementation. In general a constraint solver may be employed to compute it.

Example 4.1.1 Consider CLP(FD) and a clause:

$$p(x) \leftarrow x{<}y, q(x), r(y).$$

Assume that types for x and y inferred so far (by analyzing q and r) are given by the following RULC program:

$$\begin{array}{l} t_1(x) \leftarrow x{\in}\{1..maxint\}. \\ t_2(y) \leftarrow y{\in}\{0..6\}. \end{array}$$

where t_1 gives a type for x and t_2 for y. Let $c_1[x]$ be $x{\in}\{1..maxint\}$ and let $c_2[y]$ be $y{\in}\{0..6\}$. A CLP(FD) solver finds that $x{<}y, c_1[x], c_2[y] \rightarrow x{\in}\{1..5\}, y{\in}\{2..6\}$. Types s_1, s_2 defined by a RULC program

$$\begin{array}{l} s_1(x) \leftarrow x{\in}\{1..5\}. \\ s_2(y) \leftarrow y{\in}\{2..6\}. \end{array}$$

may be used to approximate $x{<}y, t_1(x), t_2(y)$ by the conjunction $s_1(x), s_2(y)$. $\square$

4.2 Approximating a clause

Now we give an algorithm that approximates the semantics $T^{\mathcal{C}}_{\{C\}}$ of a clause C wrt a RULC program R. A function realized by the algorithm is called *solve*. Its arguments are a CLP clause and a RULC program and its value is a RULC program. A required property of *solve* is:

$$T^{\mathcal{C}}_{\{C\}}(\gamma(R)) \subseteq \gamma(solve(C, R)).$$

Let C be $h \leftarrow c_0, b_1, \ldots, b_n$. Then $solve(C, R)$ is computed in the following steps. The data structures of the algorithm are a clause D and a RULC program Q. An invariant

$$T^{\mathcal{C}}_{\{C\}}(\gamma(R)) \subseteq T^{\mathcal{C}}_{\{D\}}(M(Q))$$

is maintained.

1. Set Q as R and D as

$$h \leftarrow c_0, approx(b_1), \ldots, approx(b_n)$$

   Unfold clause D w.r.t. Q, by iteratively applying the following until all the body atoms are of the form $t(x)$ where x is a variable.

   Choose an atom $t(f(u_1, \ldots, u_l))$ from the body ($l \geq 0$).

   i. If there is a clause $t(f(y_1, \ldots, y_l)) \leftarrow \vec{b}$ in Q then replace $t(f(u_1, \ldots, u_l))$ by $\vec{b}\theta$, where θ is the substitution $\{\, y_i/u_i \mid i = 1, \ldots l \,\}$.

   ii. If there is a clause $t(x) \leftarrow c'[x]$ in Q such that $c'[f(u_1, \ldots, u_l)]$ is satisfiable then let $\{z_1, \ldots, z_k\} := Vars(f(u_1, \ldots, u_l))$. Take new predicate symbols $s_1, \ldots, s_k$ and add to Q the clauses $\{\, s_i(z_i) \leftarrow \exists_{-\{z_i\}} c'[f(u_1, \ldots, u_l)] \mid i = 1, \ldots, k \,\}$. Replace $t(f(u_1, \ldots, u_l))$ in the body of D by $s_1(z_1), \ldots, s_k(z_k)$.

   iii. Otherwise halt with $solve(C, R) = \emptyset$.

   Remove from Q all the clauses containing *approx* in the heads.

   To show that the invariant is preserved, first notice that there is at most one clause in Q which satisfies the conditions of steps (i) and (ii) above. Unfolding (i) does not change $T^{\mathcal{C}}_{\{D\}}(M(Q))$. In (ii) we have $\models c'[f(u_1, \ldots, u_l)] \rightarrow \bigwedge_{i=1}^{k} \exists_{-\{z_i\}} c'[f(u_1, \ldots, u_l)]$. So step (ii) replaces $T^{\mathcal{C}}_{\{D\}}(M(Q))$ by its superset.

2. If some variable y occurs more than once in the body of D, say as an argument of s and s', then compute $t = s \sqcap s'$, together with a program R_t, which is predicate disjoint from Q. Delete $s(y)$ and $s'(y)$ from the body of D, and put $t(y)$ instead. Add to Q the clauses of R_t. Repeat this step until each variable occurs exactly once in the body. If some intersection is empty then halt with $solve(C, R) = \emptyset$.

   Computing intersections of types is exact (see Proposition 3.3.6), hence the invariant is obviously preserved.

3. For each variable z in D that does not occur in a body atom of the form $t(z)$ (so it occurs only in the head or the constraint) add the atom $any(z)$ to the body (and clause $any(x) \leftarrow true$ to Q, if it is not there).

   The invariant is obviously preserved. Clause D is now of the form $h \leftarrow c_0, s(z_1), \ldots, s_k(z_k)$, where $Vars(D) = \{z_1, \ldots, z_k\}$.

4. Let $x_1, \ldots, x_m$ be the free variables of c_0. For $i = 1, \ldots, m$ let x_i occurs in the atom $t_i(x_i)$ in the body of D. Compute a regular approximation $t'_1(x_1), \ldots, t'_m(x_m)$, in a RULC program R', of c_0 with types $t_1, \ldots, t_m$ for variables $x_1, \ldots, x_m$ (cf. Sect. 4.1). Set D as

   $$h \leftarrow t'_1(x_1), \ldots, t'_m(x_m), B$$

   and Q as $Q \cup R'$.

   Since $t'_1(x_1), \ldots, t'_m(x_m)$ approximates c_0 the invariant is maintained in this step.

5. If some variable z occurs more than once in the head of D, where $t(z)$ occurs in the body, then replace one occurrence of z in the head by a fresh variable y, and add $t(y)$ to the body. Repeat this operation until each variable in the head occurs once.

   The current $T^{\mathcal{C}}_{\{D\}}(M(Q))$ is a superset of the previous one, therefore the invariant also holds. Now D is of the form $h' \leftarrow s_1(y_1), \ldots, s_l(y_l)$, where each y_i occurs exactly once in h' and exactly once in the body.

6. Repeat the following step until impossible.

   Choose a non variable proper subterm u of h', where $u = f(\vec{z})$, the arity of f is ≥ 0 and $\vec{z}$ is a tuple of variables. Let s be a new predicate symbol and y a new variable. Replace the clause $D = h'[u] \leftarrow B_{\vec{z}}, B$ (where $B_{\vec{z}}$ contains those atoms from the body which contain variables $\vec{z}$) by $h'[y] \leftarrow s(y), B$. Add clause $s(u) \leftarrow B_{\vec{z}}$ to Q.

   After this step the set of constrained atoms generated by the clause D and the program Q (augmented with new clauses) remains the same as after the previous step.

The result is $solve(C, R) = \{approx(h') \leftarrow B\} \cup Q$, where $D = h' \leftarrow B$.

The invariant implies the required property $T^{\mathcal{C}}_{\{C\}}(\gamma(R)) \subseteq \gamma(solve(C, R))$, because $\gamma(solve(C, R)) = [\![approx]\!]_{solve(C,R)} = T^{\mathcal{C}}_{\{D\}}(M(Q))$ (as $approx$ occurs only once in $solve(C, R)$).

4.3 Approximating a program

Definition 4.3.1 Let R_1 and R_2 be RULC programs and let each of them contain a definition of *approx*. We define $R_1 \amalg R_2$ as a RULC program such that $(approx, R_1) \sqcup (approx, R_2) = (approx, R_1 \amalg R_2)$. □

Notice that $\gamma(R_1) \cup \gamma(R_2) \subseteq \gamma(R_1 \amalg R_2)$.

The function $T_P^{\mathcal{A}} : \mathbf{A} \to \mathbf{A}$, which approximates the function $T_P^{\mathcal{C}}$ characterizing the c-semantics of program P, is defined by

$$T_P^{\mathcal{A}}(R) = norm\left(R \amalg \coprod_{C \in P} solve(C, R) \right).$$

The role of function *norm* which is defined below, is to guarantee termination of the fixpoint computation. The approach is similar to the widening of Cousot [CC92a, CC92b] and will be also called widening. The function is not monotonic but it has a property that $R \preceq norm(R)$ (see the next section). Hence $R \preceq T_P^{\mathcal{A}}(R)$.

As $T_{\{C\}}^{\mathcal{C}}(\gamma(R)) \subseteq \gamma(solve(C, R))$, we have that $T_P^{\mathcal{A}}$ indeed approximates $T_P^{\mathcal{C}}$:

$$T_P^{\mathcal{C}}(\gamma(R)) \subseteq \gamma(T_P^{\mathcal{A}}(R)).$$

Hence $\forall n \; T_P^{\mathcal{C}} \uparrow n \subseteq \gamma(T_P^{\mathcal{A}} \uparrow n)$. Due to widening, a fixed point of $T_P^{\mathcal{A}}$ is found in a finite number of iterations. More precisely, it is a fixpoint up to equivalence $\cong$. There exists an n such that $T_P^{\mathcal{A}} \uparrow (i+1) \cong T_P^{\mathcal{A}} \uparrow i$ for all $i > n$. We call this fixpoint the *computed fixpoint* and denote it by $T_P^{\mathcal{A}} \uparrow \omega$.

Function $T_P^{\mathcal{A}}$ is in general not monotonic w.r.t. $\preceq$ (as *norm* is not monotonic). Thus we cannot claim that the computed fixpoint is the least fixpoint[1].

The computed fixpoint of $T_P^{\mathcal{A}}$ approximates the c-semantics of P, as

$$M(P) = cl(T_P^{\mathcal{C}} \uparrow \omega) \subseteq \gamma(T_P^{\mathcal{A}} \uparrow \omega).$$

4.4 Avoiding infinite loops

This section presents the widening function used in our approach to assure termination. We follow Gallagher and de Waal, however we apply the widening function used in their implementation of regular approximation tool, not the one described in [GdW92, GdW94][2].

[1]This is always the case in techniques that use widening.

[2]This operation is called *shortening* in [GdW94].

Let $Fun(p)$ denote the set all function symbols occurring in the heads of clauses defining a predicate p.

Definition 4.4.1 [Relation $D(t,s)$] Let R be an RULC program containing predicates t and s $(t \neq s)$. $D(t,s)$ is true if t depends on s and $Fun(t) = Fun(s)$. □

Definition 4.4.2 [Widening function] Let R be an RULC program and let t and s be predicates defined in R s.t. $D(t,s)$ and $s \sqsubseteq t$ holds.

Then the program $N(R)$ is obtained from R by replacing all the occurrences of s by t in the bodies of all clauses relevant for t.

Operation N is repeated until inapplicable (i.e. no predicates t and s such that $D(t,s)$ and $s \sqsubseteq t$ occur in $N^n(R)$). Then $norm(R) = N^n(R)$. □

The following example illustrates how the widening is used during the analysis.

Example 4.4.3 Let P be the following program:

$even(0).$
$even(s(s(X))) \leftarrow even(X).$

First iteration of analysis results in the RULC program:

$approx(even(X1)) \leftarrow t1(X1).$
$t1(0).$

Second iteration gives:

$approx(even(X1)) \leftarrow t2(X1).$
$t2(0).$
$t2(s(X1)) \leftarrow t3(X1).$
$t3(s(X1)) \leftarrow t1(X1).$
$t1(0).$

Next iteration produces:

$approx(even(X1)) \leftarrow t4(X1).$
$t4(0).$
$t4(s(X1)) \leftarrow t5(X1).$
$t5(s(X1)) \leftarrow t2(X1).$
$t2(0).$
$t2(s(X1)) \leftarrow t3(X1).$
$t3(s(X1)) \leftarrow t1(X1).$
$t1(0).$

and after applying *norm* (as $D(t4, t2)$ and $t2 \sqsubseteq t4$) we get:

$approx(even(X1)) \leftarrow t4(X1).$
$t4(0).$
$t4(s(X1)) \leftarrow t5(X1).$
$t5(s(X1)) \leftarrow t4(X1).$

Now observe, that to detect the fixpoint we need yet another iteration:

$approx(even(X1)) \leftarrow t6(X1).$
$t6(0).$
$t6(s(X1)) \leftarrow t7(X1).$
$t7(s(X1)) \leftarrow t4(X1).$
$t4(0).$
$t4(s(X1)) \leftarrow t5(X1).$
$t5(s(X1)) \leftarrow t4(X1).$

Since $t6 \sqsubseteq t4$ (and $t4 \sqsubseteq t6$) widening is applied again and finally we obtain:

$approx(even(X1)) \leftarrow t6(X1).$
$t6(0).$
$t6(s(X1)) \leftarrow t7(X1).$
$t7(s(X1)) \leftarrow t6(X1).$

□

Proposition 4.4.4 Let R be an RULC program. Then $R \preceq norm(R)$.

PROOF: We present a proof for the case of RUL. It suffices to show that $R \preceq N(R)$. Let t and s be those predicates of R that are selected in computing $N(R)$. So $D(t, s)$ and $s \sqsubseteq t$ in R. It is sufficient to show that if $p(u) \in M(R)$ then $p(u) \in M(N(R))$ for any predicate p and any ground term u. The proof is by induction on depth m of u. If $m = 1$ then the property trivially holds. Let depth of u be m and let the result hold for each term of depth less than m. Let u be of the form $f(u_1, \ldots, u_n)$. Assume that $p(f(u_1, \ldots, u_n)) \in M(R)$. Then there exists in R a clause $C = p(f(x_1, \ldots, x_n)) \leftarrow p_1(x_1), \ldots, p_n(x_n)$ and $p_i(u_i) \in M(R)$, for $1 \le i \le n$. If $p_i = s$ then $t(u_i) \in M(R)$. By the inductive assumption, each $p_i(u_i) \in M(N(R))$ and if $p_i = s$ then $t(u_i) \in M(N(R))$. As $N(R)$ contains the clause obtained from C by replacing s in the r.h.s. by t, we have $p(f(u_1, \ldots, u_n)) \in M(N(R))$. □

Now we provide informal justification that *norm* makes the iteration of $T_P^{\mathcal{A}}{\uparrow}i$, $i = 0, 1, \ldots$, finite. Let us begin with introducing a notion of a position. A *position* is a sequence of the form $(p, i_0), (f_1, i_1), \ldots, (f_k, i_k)$, where p is a predicate name (from the analyzed program P), i_0 a number of its argument, f_j is a function symbol (from P) and i_j a number of its argument (for $j = 1, \ldots, k$). A position describes a path from a predicate (of P) to a type of a subterm of one its arguments. In other words it describes a path in a RULC program, from *approx* to a type. Any position corresponds to exactly one type (and any type relevant for *approx* has a corresponding position, possibly not unique).

Assume that we use function $T_P^{\mathcal{A}}$ without *norm*, let us call it $T'^{\mathcal{A}}_P$. Notice that programs $Q_n := T'^{\mathcal{A}}_P \uparrow n$ do not contain recursion (no predicate depends on itself). Recursion in P leads to infinite iteration of $T'^{\mathcal{A}}_P$: each Q_n is not equivalent to Q_{n+1} (for $n = 1, 2, \ldots$).

Recursion in P results in existing in Q_n (for a sufficiently large n) of types with the following property. A type t at a position π in Q_n depends on a type t' at position π in Q_m, $m < n$. This means that there is a type s in Q_n (at a position π') such that $[\![s]\!]_{Q_n} = [\![t']\!]_{Q_m}$ and t depends on s. As $Q_1 \preceq Q_2 \preceq \ldots$, we have $[\![t']\!]_{Q_m} \subseteq [\![t]\!]_{Q_n}$. Such pairs of types can be seen as the reason of the infinite iteration, at some further $Q_{n'}$, $n < n'$, there appears a type r which depends on a type equivalent to t (which depends on a type equivalent to s), etc.

Function *norm* destroys all such infinite chains, replacing such dependency chains of increasing length by recursion. Notice that for a sufficiently large n $Fun(t_n) = Fun(s_n)$, where t_n and s_n are the types at the positions π and π' respectively in Q_n. So at some stage N becomes applicable to t_n and s_n.

We conclude this section by showing that *norm* is not monotonic (and therefore $T_P^{\mathcal{A}}$ is not).

Example 4.4.5 Consider CLP over the Herbrand domain and RUL programs R and R':

$$
\begin{array}{ll}
R = \{ & approx(a). \\
& approx(f(X)) \leftarrow s(X). \\
& s(a). \\
& s(f(X)) \leftarrow t(X). \\
& t(a). \ \}
\end{array}
\qquad
\begin{array}{ll}
R' = \{ & approx(a). \\
& approx(f(X)) \leftarrow s(X). \\
& approx(b). \\
& s(a). \\
& s(f(X)) \leftarrow t(X). \\
& t(a). \ \}
\end{array}
$$

We have $R \preceq R'$. Applying *norm* to R results in

$$\begin{array}{rl} norm(R) = \{ & approx(a). \\ & approx(f(X)) \leftarrow approx(X). \\ & s(a). \\ & s(f(X)) \leftarrow t(X). \\ & t(a). \;\; \} \end{array}$$

while $norm(R') = R'$. Now $norm(R) \not\preceq norm(R')$ (as $\gamma(norm(R))$ is infinite, while $\gamma(norm(R'))$ is finite). So *norm* is not monotonic. □

4.5 An example

A prototype of the type analyzer has been implemented. The analyzer treats all the finite domains in a uniform way, namely as *anyfd* (the types of the form $cl(x \in S[]x)$ are not yet implemented).

The program below solves the well-known N-queens problem.

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

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

labeling([]).
labeling([X|Y]):-
    indomain(X),
    labeling(Y).

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

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

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

The `entry` declaration indicates a call pattern of the top predicate for the call-success analysis. Types inferred by the system are presented below. The actual implementation of the analyzer provides a more user-friendly syntax for types, namely regular term grammars (see Section 3.6).

```
CALL-type    : nqueens(nat,any)
SUCCESS-type : nqueens(nat,list(nat))
-------------------------
CALL-type    : labeling(list(anyfd))
SUCCESS-type : labeling(list(nat))
-------------------------
CALL-type    : constrain_queens(list(anyfd))
SUCCESS-type : constrain_queens(list(anyfd))
-------------------------
CALL-type    : safe(anyfd,list(anyfd),int)
SUCCESS-type : safe(anyfd,list(anyfd),int)
-------------------------
CALL-type    : noattack(anyfd,anyfd,int)
SUCCESS-type : noattack(anyfd,anyfd,int)
```

4.6 Discussion

A variety of approaches to (descriptive) type inference has been proposed in the literature. The first one is based on solving a set of recursive equations extracted from a program. The solutions describe success types. This line was initiated by Mishra [Mis84] and then continued by [HJ90, FSVY91]. Zobel [Zob87] and later Pyo and Reddy [PR89] extended Mishra's technique to parametric polymorphism. The recent paper by Charatonik and Podelski [CP98], however being focused on directional types, is strongly related to this framework.

Another approach uses bottom-up abstract interpretation [CC92a] and was applied for the first time by Yardeni and Shapiro [YS91]. Gallagher and de Waal [GdW92, GdW94] gave the fast algorithm for computing regular types (therein called "regular appoximation") which also uses abstraction of the T_P operator. Our work is based on their solution. Heinze and Jaffar [HJ92] and Frühwirth *et al.* [FSVY91] study relationships between the approaches based on set equations and bottom-up abstract interpretation.

Barbuti and Giacobazzi [BG92] also use bottom-up abstract interpretation to infer types, but they require types to be a priori assigned to the function symbols in the program. An interesting technique based on abstract compilation and multiple using of the *Prop* abstract domain has been presented by Codish and Demoen [CD94].

Bottom-up abstract interpretaion methods are often combined with so-called "magic transformation". This allows one to approximate call-success semantics using algorithms which approximate the declarative semantics. In particular one can infer call and success types of the program, as pointed in [GdW94]. For general remarks about the role of "magic transformation" in static analysis of logic programs see [DR94, Nil95].

A framework for approximating operational semantics of logic programs has also been employed to find all possible runtime instantiations of program variables [JB92, HCC95]. Type graphs used therein as an abstract domain are essentially regular types.

Chapter 5

Incorrectness diagnosis

5.1 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 5.1.1 A program P with a class of initial atomic goals $\mathcal{G}$ is *partially correct* wrt S iff $CS(P,\mathcal{G}) \subseteq S$, it is *complete* wrt to S if $S \subseteq CS(P,\mathcal{G})$. $\square$

Thus, partial correctness wrt S means that all calls and successes of the program in any computation starting with a goal from $\mathcal{G}$ 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 5.1.2 Assume that the program of 4.5 contains an error in the clause:

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

and that the specification provided by the user is the same as the call-success types inferred by the analyzer for the bug-free program, namely:

```
CALL-type    : nqueens(nat,any)
SUCCESS-type : nqueens(nat,list(nat))
-------------------------
CALL-type    : labeling(list(anyfd))
SUCCESS-type : labeling(list(nat))
-------------------------
CALL-type    : constrain_queens(list(anyfd))
SUCCESS-type : constrain_queens(list(anyfd))
-------------------------
CALL-type    : safe(anyfd,list(anyfd),int)
SUCCESS-type : safe(anyfd,list(anyfd),int)
-------------------------
CALL-type    : noattack(anyfd,anyfd,int)
SUCCESS-type : noattack(anyfd,anyfd,int)
```

Now, the program is incorrect wrt the specification. 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-success semantics we are going to formulate a sufficient condition for partial correctness. For this we need the following definition.

Definition 5.1.3 Let M be an instance-closed set of constrained atoms, Consider an implication C of the form $c_0, B_1, \ldots, B_n \to H$, where c_0 is a constraint, $n \geq 0$, and $B_1, \ldots, B_n, H$ are (non constraint) atoms. We say that C **is satisfied** wrt M (this fact will be denoted by $M \approx C$) when $T^{\mathcal{C}}_{\{C\}}(M) \subseteq M$.

□

In other words, C is satisfied iff 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 $c[]H\theta \in M$.

The following proposition is a simple consequence of prop. 2.4.2.

Proposition 5.1.4 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

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

PROOF The conditions VC mean that $T^{\mathcal{C}}_{PCS}(S) \subseteq S$. Hence $T^{\mathcal{C}}_{PCS}(\mathcal{G}) \subseteq T^{\mathcal{C}}_{PCS}(S) \subseteq S$. By simple induction we have $(T^{\mathcal{C}}_{PCS})^n(\mathcal{G}) \subseteq S$ and finally $(T^{\mathcal{C}}_{PCS})^\omega(\mathcal{G}) \subseteq S$. □

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

Example 5.1.5 Consider the erroneous clause for `safe` of 5.1.2. 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 a specification. The specification gives the call and success types of the predicates involved, including the types of the built-in `is`, which are not specified by the user but provided by the system library.

Consider the types specified by the user as in example 5.1.2 and the first implication above. To check it one has to show that for every substitution 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,sint)`. 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.

□

5.2 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 5.1.4 it follows that if a program is incorrect then it contains an incorrect clause[1]. 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 5.1.3 are satisfied. Consider such an implication $C = c_0, b_1, \ldots, b_n \to H$ and a specification S. One may imagine a following procedure of checking whether $S \approx\!\!\!\!| \; C$. Consider the function $T_P^{\mathcal{C}}$ defined in def. 2.3.3. Now $T_{\{C\}}^{\mathcal{C}}(S) \subseteq S$ iff $S \approx\!\!\!\!| \; C$. The problem is that $T_{\{C\}}^{\mathcal{C}}(S) \subseteq S$ may be undecidable.

Assume that we have a computable approximation of the function $T_P^{\mathcal{C}}$, i.e. a computable function $T_P^{\mathcal{A}}$ over specifications such that $T_{\{C\}}^{\mathcal{C}}(S) \subseteq T_{\{C\}}^{\mathcal{A}}(S)$ (for all specifications S in our specification language). As the check for $\subseteq$ for our specifications is decidable, this gives a decision procedure for $T_{\{C\}}^{\mathcal{A}}(S) \subseteq S$.

Definition 5.2.1 An implication C is **abstractly satisfied** wrt S if $T_{\{C\}}^{\mathcal{A}}(S) \subseteq S$. □

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

Proposition 5.2.2 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 $T_P^{\mathcal{A}}$ conditions (VC) may be satisfied but not abstractly satisfied.

[1] 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.

The proposition gives an algorithm for error diagnosis. Computing whether $T^{\mathcal{A}}_{\{C\}}(S) \subseteq S$ 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 5.2.3 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..10 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 $T^{\mathcal{A}}_{P}$. $T^{\mathcal{A}}_{\{C\}}(S)$ may state that the argument of p at success is `anyfd`. It is still a correct, though very imprecise approximation. For this choice of $T^{\mathcal{A}}_{P}$ does not permit to establish correctness of the clause, since the conditions (VC) are not abstractly satisfied.

□

5.3 Discussion

Verification of runtime properties Proving partial correctness of logic program (wrt runtine properties) has been studied by many authors. We apply a modification for CLP of the *inductive assertion method* of Drabent and Małuszyński [DM88, Dra88]. (See also [AM94, BM97] for its application in the context of directional types.) The specifications are expressed using pre- and postconditions for every predicate in the program. A special case of this method was presented by Bossi and Cocco [BC89]. Their approach is restricted to proving properties which are closed under substitution.

Colussi and Marchiori [CM91] consider a related method, where assertions are associated with the program points rather then the predicates. We refer to Apt and Marchiori [AM94] for a comprehensive discussion and comparison of the methods.

Abstract diagnosis *Abstract diagnosis* [CLV94, CLMV98] is an approach in which 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. Our abstract domain does not satisfy the conditions of the abstract diagnosis method, so the method, as it is, is inapplicable.

For the general discussion about assertion-based diagnosis for CLP and its relations to other techniques see [BDD+97].

Chapter 6

Implementation issues

This chapter surveys some details and main design decisions of the prototype implementation of the analyzer and the diagnoser. The current version of the prototype has been implemented in CHIP [Cos96]. The analyzer has been also ported to SICStus Prolog and CIAO [CLI97, PBH98]. Section 6.4 shows the graphical user interface of the entire system which integrates both the analyzer and the diagnoser.

6.1 The analyzer

First step performed by the analyzer is transforming an input program P to a program P^{CS} of proposition 2.4.2 or to verification conditions of proposition 5.1.4. This is essentially equivalent to the so-called "magic transformation" of the given program. So each predicate, say `p` of the program is associated with two new predicates `call_p` and `succ_p`, which correspond to $^\bullet$`p` and `p`$^\bullet$ from the previous sections. For example a clause:

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

is transformed into the following clauses:

```
call_safe(X,Y,1) :-
    call_constrain_queens([X|Y]).

call_constrain_queens(Y) :-
    call_constrain_queens([X|Y]),
    succ_safe(X,Y,1).
```

```
succ_constrain_queens([X|Y]):-
    call_constrain_queens([X|Y]),
    succ_safe(X,Y,1),
    succ_constrain_queens(Y).
```

The declaration of (a type of) a top goal, like for instance:

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

must be given by the user. This declaration is transformed to the clause:

```
call_nqueens(A,B) :- int(A), any(B).
```

Predicates `int/1` and `any/1` describe built-in types (see sect. 3.5) and are considered by the analyzer accordingly. The core of the "magic transformation" algorithm had been already implemented by Gallagher and de Waal [GdW92, GdW94].

One of the parts of the system is a database of call and success types for built-in predicates. During the next step the analyzer replaces occurrences of built-in predicates with types stored in the database. For instance `succ_indomain(X)` is replaced with `nat(X)`. Calls for built-in predicates remain unchanged, they are verified against the database information during the analysis.

Providing type definitions rather than actual predicate definitions is also a method to deal with programs which are divided into several modules. The same can be done in case of standard or user-defined libraries.

During the next step, RULC clauses describing the initial goals are constructed from the entry declarations provided by the user. The type definitions given by the user in the form of regular term grammars are transformed into a RULC program. Only those instances of parametric rules are employed that are needed to define the types used in the entry declarations.

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

are translated into:

```
call_maxt(A,B) :-
    'tree(int)'(A),
    any(B).

'tree(int)'(void).
'tree(int)'(t(A,B,C)) :-
```

```
        int(A),
        'tree(int)'(B),
        'tree(int)'(C).
```

At this point the analyzer starts the fix-point bottom-up computation that approximates the c-semantics of a program (transformed to P^{CS}). Declarations of external predicates and of the initial goal are also taken into account. The program is split up in according to the *strongly connected components* in its dependency graph. A fix-point is computed for each of the components separately. The implementation of this part is based on a code provided by Gallagher and de Waal [GdW92, GdW94]. See therein for further details.

In this phase of the analysis, operations on types are employed. We have extended them to handle constraints (see sect. 3.3). The implementation of that part has been done in a modular way, so it should be easy to plug-in type operations relevant for other constraint domains.

After the analysis has been completed the inferred types are presented to the user. Inside the analyzer, types are just RULC programs, e.g. the success type for `maxt1` is represented as:

```
succ_max1(X1,...) :- t10(X1),...
t10(void).
t10(t(X1,X2,X3)) :-
    int(X1),
    t10(X2),
    t10(X3).
```

This representation is rather cumbersome to the user, hence we provide an equivalent regular grammar notation (see sect. 3.6) to communicate with her:

```
SUCCESS-type : max1(t10,...)
t10 --> void
t10 --> t(int,t10,t10)
```

If the user has defined a type which is equivalent to an inferred one then the system displays the user-defined type. This also includes the case when the user has given a polymorphic definition, like:

```
:- typedef tree(A) --> void; t(A,tree(A),tree(A)).
```

In presence of the above definition the inferred success type for `maxt` will be displayed as follows:

```
SUCCESS-type : max1(tree(int),...)
```

Note that the parameter `A` is instantiated to a particular inferred type (`int` in our example), as our analysis does not infer polymorphic types. The inferred types are shown to the user on her request.

6.2 The benchmarks

The prototype of the system that integrates the analyzer and the diagnoser has been implemented in CHIP. However, as CHIP is not a general purpose language, we found it to be rather unsuitable for the analysis. Therefore this part of the system has been implemented also in SICStus Prolog, and recently in CIAO [CLI97].

Table 6.1 shows the times of analyzing sample programs under SICStus Prolog. The tests have been performed on PC with Linux. Some of the examples were given COSYTEC (the CHIP provider). Here P^{CS*} denotes a program P after the "magic transformation" together with clauses defining user's types and built-in types.

Benchmark P	Clauses in P	Clauses in P^{CS*}	Time
`min_max.pl` a sample using `min_max/2` predicate	2	70	1.6 s
`nqueens.pl` solves N queens problem	8	85	3.7 s
`sumfac.pl` computes a sum of factorials of first N consecutive numbers	11	94	7.3 s
`ship.pl` a ship loading example (from COSYTEC)	8	325	10.3 s
`typical1.pl` part of a typical CHIP program (from COSYTEC)	3	70	1.8 s
`typical2.pl` a typical data gathering scheme (from COSYTEC)	6	94	3.4 s
`combination.pl` calculates a product mix for a wood cutting problem (from COSYTEC)	8	1160	69.8 s

Table 6.1: The benchmarks

The table shows that the magic transformation may give enormous increase of the number of clauses. The increase depends on the number of literals in the bodies of the clauses in the original program. The time needed for inferring types of the original program depends on the size and on the structure of the transformed program. Complexity analysis of the inference algorithm and experiments with programs used in practice are topics of the future work.

Note that, since the analyzed program is divided into strongly connected components, which typically consist of 1 - 4 predicates, the huge number of clauses in P^{CS*} does not slow down the analysis considerably.

If the type inference turns out to be too expensive for a given program the user may instead provide directly a type specification and use the diagnoser. Verification of a given type specification can be done efficiently.

6.3 The diagnoser

After the analysis the inferred types are shown to the user, who may decide to start diagnosis. 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 the chapter 5, 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. At this point warnings about suspicious clauses may be displayed. The pending list is updated taking into account the new information and unless it is empty the diagnoser waits 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 locate 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).

6.4 The user interface

Fig. 6.1 shows the GUI of the system. The top window contains four browsers. The first one allows to select a source file to be analyzed.

The browser named "Inferred" shows a list of user-defined predicates, as well as built-in ones, that occur in the program. By pointing one of them the user requires its inferred types (or those from the database of built-ins) to be displayed.

Pressing the button "Diagnose Predicate" results in starting diagnosis of a part of the program that is relevant for the actually highlighted predicate. Then the browser "Ask" is filled up with all the necessary queries which have to be answered by the oracle. Predicates for which the specification has been already provided are moved to the browser "User"[1]. At each point of the diagnosis process the user may retract formerly given specification by pushing buttons "Clear" or "Clear All". Then the relevant predicates are moved back to the list of pending oracle queries. The diagnosis of a given predicate is over when the browser "Ask" is empty.

[1]The entry point declaration is considered as a part of user's specification and is present in the browser "User" from the very beginning of the diagnosis.

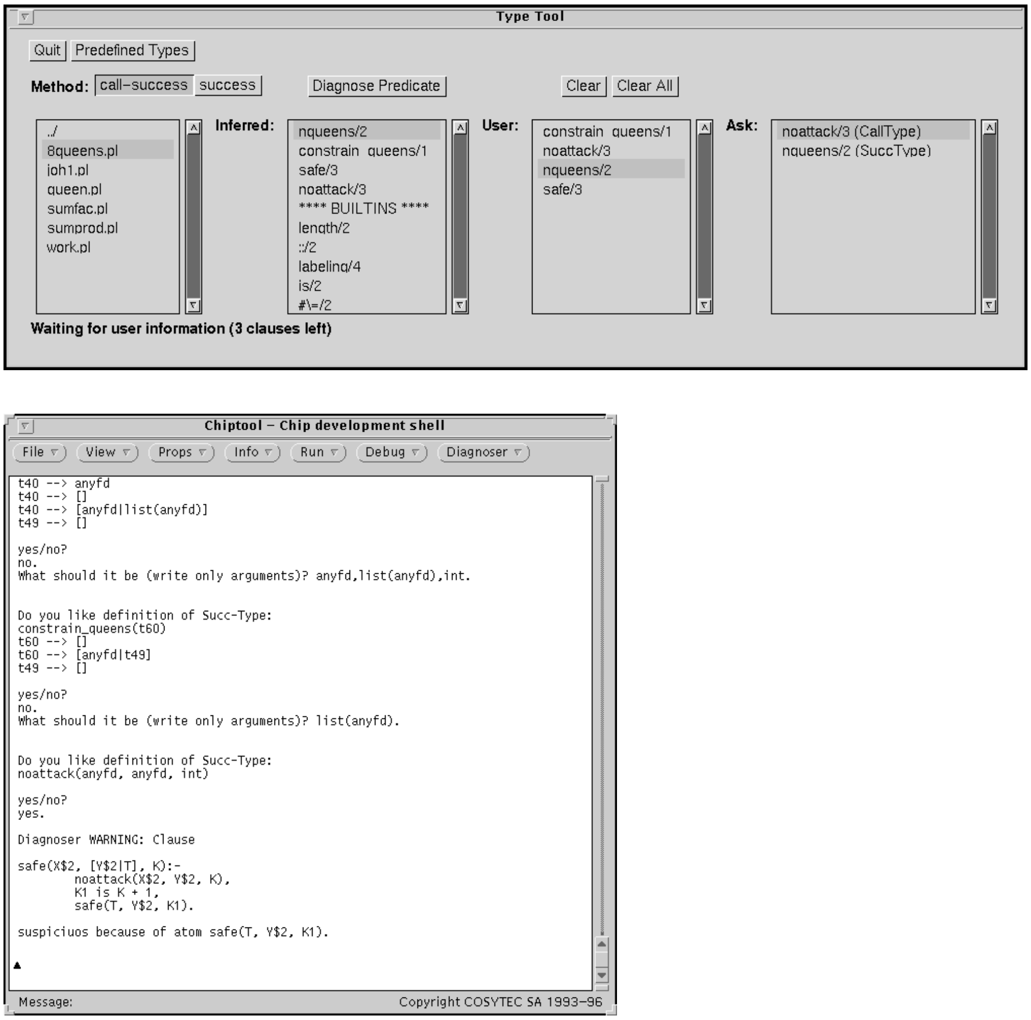

Figure 6.1: The user interface of the analyzer/diagnoser

Chapter 7

Discussion and future work

7.1 Concluding remarks

In this thesis we propose a method of computing semantic approximations for CLP programs and show how to use it in the process of finding some kind of programming errors. We are mainly interested in CLP(FD), particularly in the language CHIP. Our approach is based on the (operational) call-success semantics and the (declarative) c-semantics.

As a specification language to express the semantic approximations we propose a system of regular types for CLP, which is an extension of an approach used for logic programs. The types are defined by (a restricted class of) CLP programs, called RULC programs. We present an adaptation to CLP of an algorithm for computing regular approximations of the declarative semantics. This algorithm can also be used for approximating the call-success semantics, due to a characterization of this semantics by the c-semantics of a transformed program.

We have adopted a regular approximation system (described in [GdW92, GdW94]) to constraint logic programming over finite domains. The current version is implemented for the programming language CHIP. We expect it to be easily portable to analyze other CLP languages, as we have isolated its parts which are specific for CHIP. The prototype of the analyzer has been implemented in CHIP and has been ported to SICStus Prolog and CIAO [CLI97, PBH98]. The system presents types to the user as regular term grammars, which are more easily comprehensible than RULC programs. It provides a restricted but useful kind of polymorphism (cf. Section 3.6)

We also show how to perform incorrectness diagnosis based on the inferred type information. The prototype of the tool integrating both the type analyzer and the diagnoser has been implemented in CHIP. The very diagnosis algorithm uses an approximate specification (a superset) of the intended semantics of the program. In contrast to most other 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 program being incorrect wrt the specification. It may however happen that it points out a clause that is abstractly incorrect but (concretely) correct wrt the specification.

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.

7.2 Limitations of the approach

The presented approach assumes the Prolog selection rule, while in many CLP languages selection with delays is possible. For example in CHIP the `is/2` predicate is delayed if the arguments of the arithmetic expression are not sufficiently instantiated.

Example 7.2.1 *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 analyzer warns about the call-type of `is/2`*, as the second argument is not a ground arithmetic expression.*

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

The left-to-right 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 [BM97].

Example 7.2.2 *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.*

```
:- 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 `maxt` *have the last argument 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*[1] *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.*

[1] 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.

The restricted expressive power of our specification language makes possible 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 7.2.3 *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 5.1.4, 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.*

7.3 Future work

There are several lines along which the work presented here may be extended. Some of them have been mentioned in respective places in the text.

Other constraint domains. It would interesting to investigate our diagnosis method in the more general context, i.e. without syntactic unification. The technique presented in the thesis can be viewed as a special case of such a general approach.

Our prototype implementation can be extended towards further constraint domains, like real numbers. As already pointed in Chapter 6 we believe that this would be rather easy to do, as parts of the code which are specific for CHIP are separated.

Improvements of the analysis. Some improvements of our analysis towards increasing precision can be done. In particular the widening operator requires further redesigning and re-implementation. The current one takes into account only the most recent iteration. It is possible to design a widening function so it considers results of two most recent iterations and by comparing them is able to locate places where the abstract denotation increases.

Another direction is to consider an alternative implementation based on a transformation of a given program directly to a RUL(C) program, as discussed in [FSVY91].

Delayed goals In many CLP languages the applied selection rule admits goals to be delayed, until, for instance some variables have become instantiated. It would be desirable to refine our method to handle such cases.

More expressive type language Enriching our type language with parametric polymorphism is surely worth to investigate. Another direction for further research may include types which in more accurate way reflect non-Herbrand part of the underlying constraint domain.

Prescriptive and descriptive types for CLP Our framework is based on descriptive types. A relation between prescriptive and descriptive types, and notion of partial correctness has been clarified in [DM98]. Nevertheless, this topic requires further careful studies.

Appendix A

A sample diagnosis session

In the example below we present a sample analysis session of our system. Consider the erroneous program solving N-quuens problem:

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

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

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

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

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

The predicate `List::1..N` associates each element of the list `List` with the domain `1..N`. The call of `labeling(List,0,most_constrained,indomain)` results in assigning each domain variable from `List` with the least such a value from its domain that the current constraints are satisfiable. The variables are selected in according to the `most_constrained` heuristics, which means that the variable which occurs in most constraints is selected first.

The diagnosing procedure starts with inferring types for the programs. Here are the messages given by the analyzer:

```
Analyzer Message: Analyzing file 8queens.pl.

Analyzer WARNING: Problems with Call-Type of: #\= / 2.

Analyzer Message: end of file 8queens.pl analysis.
```

The analyzer gave the warning about call type for `#\= / 2`. This means that the call type for this predicate (in the clause defining `noattack / 3`) computed by the analyzer is not a subtype of the one which is required for `#\= / 2`[1]. Now assume that the user wants to inspect all the types inferred by the system. The types displayed by our system are presented below:

```
Call-Type: nqueens(int, any)

Succ-Type: nqueens(nat, t66)
t66 --> [nat|t49]
t49 --> []

-------------------------

Call-Type: constrain_queens(list(anyfd))

Succ-Type: constrain_queens(t60)
t60 --> []
t60 --> [anyfd|t49]
t49 --> []

-------------------------

Call-Type: safe(t40, t41, int)
```

[1] In general this kind of warning may be due to the lack of precision of the analysis.

```
t40 --> anyfd
t40 --> []
t40 --> [anyfd|list(anyfd)]
t41 --> anyfd
t41 --> [anyfd|list(anyfd)]
t41 --> []

Succ-Type: safe(t40, t49, int)
t40 --> anyfd
t40 --> []
t40 --> [anyfd|list(anyfd)]
t49 --> []

-------------------------

Call-Type: noattack(t40, anyfd, int)
t40 --> anyfd
t40 --> []
t40 --> [anyfd|list(anyfd)]

Succ-Type: noattack(anyfd, anyfd, int)
```

The inferred type `t66` is a singleton list of natural numbers. The type `t60` is an empty or a singleton list of `anyfd`. The types `t40` and `t41` both describe unions of `anyfd` and `list(anyfd)`. Obviously all of them are not the intended types.

Now assume that the user dislikes the success type for `nqueens/2`, and therefore she starts the diagnosis with this predicate. The system puts all the relevant predicates in the list, and then then user is queried for their specification, as follows:

```
Diagnoser Message: Incorrectness Diagnosis: predicate nqueens / 2

Do you like definition of Call-Type:
constrain_queens(list(anyfd))

yes/no?
yes.
```

If the inferred information fits the user's expectation, then she says "yes".

Otherwise the specification for a given call/success type must be provided, like below:

```
Do you like definition of Call-Type:
safe(t40, t41, int)
t40 --> anyfd
t40 --> []
t40 --> [anyfd|list(anyfd)]
t41 --> anyfd
t41 --> [anyfd|list(anyfd)]
t41 --> []

yes/no?
no.
What should it be (write only arguments)? anyfd, list(anyfd), int.


Do you like definition of Succ-Type:
safe(t40, t49, int)
t40 --> anyfd
t40 --> []
t40 --> [anyfd|list(anyfd)]
t49 --> []

yes/no?
no.
What should it be (write only arguments)? anyfd, list(anyfd), int.


Do you like definition of Succ-Type:
constrain_queens(t60)
t60 --> []
t60 --> [anyfd|t49]
t49 --> []

yes/no?
no.
What should it be (write only arguments)? list(anyfd).
```

```
Do you like definition of Succ-Type:
noattack(anyfd, anyfd, int)

yes/no?
yes.
```

Now the diagnosis comes to the point where the clause responsible for the error has been found:

```
Diagnoser WARNING: Clause

safe(X2, [Y2|T], K):-
    noattack(X2, Y2, K),
    K1 is K + 1,
    safe(T, Y2, K1).

suspicious because of atom safe(T, Y2, K1).
```

The clause has been reported since one of the corresponding verification conditions has been violated. The user can keep on diagnosing to be sure that there are no other suspicious clauses.

```
Do you like definition of Call-Type:
noattack(t40, anyfd, int)
t40 --> anyfd
t40 --> []
t40 --> [anyfd|list(anyfd)]

yes/no?
no.
What should it be (write only arguments)? anyfd, anyfd, int.


Do you like definition of Succ-Type:
nqueens(nat, t66)
t66 --> [nat|t49]
t49 --> []

yes/no?
```

```
no.
What should it be (write only arguments)? nat, list(nat).

Diagnoser Message:
end of predicate nqueens / 2 Incorrectness Diagnosis.
```

Now the other predicates may be diagnosed. The information that has been already provided by the user is stored by the diagnoser, so the user does not have to re-enter it.

Bibliography

[AB96] K.R. Apt and R. Ben-Eliyahu. Meta-variables in Logic Programming, or in Praise of Ambivalent Syntax. *Fundamenta Informaticae*, 28(1-2):22–36, 1996.

[AM94] K.R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–764, 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.

[BG92] R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of Computer Programming*, 19(3):281–313, 1992.

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

[Boy96] J. Boye. *Directional Types in Logic Programming.* Linköping studies in science and technology, dissertation no. 437, Linköping University, 1996.

[Car97] L. Cardelli. Type systems. In Allen B. Tucker Jr., editor, *The Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, 1997.

[CC92a] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programming. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[CC92b] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CD94] M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In *Proceedings of the First Symposium on Static Analysis*, volume 864 of *LNCS*, pages 281–297. Springer-Verlag, September 1994.

[CDMP98] M. Comini, W. Drabent, J. Małuszyński, and P. Pietrzak. A type-based diagnoser for CHIP. ESPRIT DiSCiPl deliverable, September 1998.

[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.

[Cla79] K. L. Clark. Predicate logic as computational formalism. Technical Report 79/59, Imperial College, London, December 1979.

[CLI97] The CLIP Group. *CIAO System Reference Manual.* Facultad de Informática, UPM, Madrid, August 1997. CLIP3/97.1.

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

[CLV94] M. Comini, G. Levi, and G. Vitiello. Abstract debugging of logic programs. In L. Fribourg and F. Turini, editors, *Proc. Logic Program Synthesis and Transformation and Metaprogramming in Logic 1994*, volume 883 of *Lecture Notes in Computer Science*, pages 440–450, Berlin, 1994. Springer-Verlag.

[CM91] L. Colussi and E. Marchiori. Proving correctness of logic programs using axiomatic semantics. In K. Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 629–642. MIT Press, 1991.

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

[CP98] W. Charatonik and A. Podelski. Directional type inference for logic programs. In G. Levi, editor, *Proceedings of SAS'98, the International Symposium on Static Analysis*, LNCS. Springer - Verlag, 1998. To appear.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[Der93] P. Deransart. Proof method of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.

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

[DM98] P. Deransart and J. Małuszyński. Towards soft typing for CLP. In *proceedings of Types for Constraint Logic Programming, post-conference workshop of JICSLP'98*, 1998.

[DP98] W. Drabent and P. Pietrzak. Inferring call and success types for CLP programs. ESPRIT DiSCiPl deliverable, September 1998.

[DP99] W. Drabent and P. Pietrzak. Type Analysis for CHIP. In *Proc. of the Seventh International Conference on Algebraic Methotology and Software Technology (AMAST'98)*, LNCS. Springer-Verlag, 1999. To appear.

[DR94] S. Debray and R. Ramakrishnan. Abstract interpretation of logic programs using magic transformations. *Journal of Logic Programming*, 18(2):149–176, 1994.

[Dra88] W. Drabent. On completeness of the inductive assertion method for logic programs. Unpublished note (available from `www.ipipan.waw.pl/~drabent`), Institute of Computer Science, Polish Academy of Sciences, May 1988.

[DZ92] P. Dart and J. Zobel. A regular type language for logic programs. In Pfenning [Pfe92b], pages 157–187.

[Fer87] G Ferrand. Error diagnosis in logic programming. *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.

[FP98] F. Fages and M. Paltrinieri. A generic type system for CLP($\mathcal{X}$). Technical Report R97–16, LIENS, 1998. Poster abstract in *Proc. of the 1998 Joint International Conference and Symposium on Logic Programming*, MIT Press, pages 353–354.

[FSVY91] 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, Amsterdam, July 1991. IEEE Computer Society Press. Corrected version available from `http://WWW.pst.informatik.uni-muenchen.de/~fruehwir`.

[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, *Proc. 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.

[HCC95] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, March 1995.

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

[HJ90] N. Heinze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 197–209. ACM Press, 1990.

[HJ92] N. Heinze and J. Jaffar. Semantic types for logic programs. In Pfenning [Pfe92b], pages 141–155.

[HL94] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[HT92] P.M. Hill and R.W. Topor. A semantics for typed logic programs. In Pfenning [Pfe92b], pages 1–62.

[JB92] 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.

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

[JMMS98] J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, (37)1–3:1–46, 1998.

[LC98] L. Lu and J. G. Cleary. On Dart-Zobel algorithm for testing regular type inclusion. Under review, 1998.

[Llo87a] J. Lloyd. Declarative error dianosis. *New Generation Computing*, 5(2):133–154, 1987.

[Llo87b] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second, extended edition, 1987.

[LR91] T.K. Lakshman and U.S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Proc. of the 8th International Logic Programming Symposium*, pages 202–217. MIT Press, 1991.

[Mis84] P. Mishra. Towards a theory of types in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 289–298, 1984.

[MO84] A. Mycroft and R. O'Keefe. A polymorphic type sytem for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[NF94] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional programming. *Journal of Functional Programming*, 4(3):337–370, 1994.

[Nil95] U. Nilsson. Abstract interpretation: A kind of magic. *Theoretical Computer Science*, 142:1–2, May 1995.

[PBH98] G. Puebla, F. Bueno, and M. Hermenegildo. A framework for assertion-based debugging in constraint logic programming. In *proceedings of Types for Constraint Logic Programming, post-conference workshop of JICSLP'98*, 1998.

[Pfe92a] F. Pfenning. Dependent types in logic programming. In Pfenning [Pfe92b], pages 189–216.

[Pfe92b] F. Pfenning, editor. *Types in Logic Programming.* MIT Press, 1992.

[Pie97] P. Pietrzak. Type Inference for CLP(FD). In *Proc. of International Logic Programming Symposium 97*, Port Jefferson, NY, 1997. MIT Press. Poster. Extended version in *The Workshop on Tools and Environments for (Constraint) Logic Programming, 1997.*

[PR89] C. Pyo and U. Reddy. Inference of polymorphic types for logic programs. In E. Lusk and R. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 1115–1132. MIT Press, 1989.

[RNP92] Y. Rouzaud and L. Nguyen-Phuong. Integrating modes and subtypes into a prolog type-checker. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 85–97. MIT Press, 1992.

[Sha82] E. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1982.

[SHC96] Z. Somogyi, F. Hederson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):14–64, 1996.

[SHK97] J.-G. Smaus, P. Hill, and A. King. Domain construction for mode analysis of typed logic programs. Technical report, November 1997.

[Smo89] G. Smolka. *Logic Programming over Polimophically Order-Sorted Types.* PhD thesis, University of Kaiserslautern, 1989.

[Tes96] A. Tessier. Declarative debugging in constraint logic programming. In *Asian Computing Science Conf.*, LNCS. Springer– Verlag, December 1996.

[YS91] E. Yardeni and E. Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–153, 1991.

[Zob87] J. Zobel. Derivation of polymorphic types for Prolog programs. In J-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 817–838. MIT Press, 1987.