Linköping Studies in Science and Technology

Dissertation No. 772

A Type-based Framework for Locating Errors in Constraint Logic Programs

by

Paweł Pietrzak

Department of Computer and Information Science
Linköpings universitet
SE–581 83 Linköping, Sweden

Linköping 2002

To Gosia, Marta and Daga

Abstract

This thesis presents a method for automatic location of type errors in constraint logic programs (CLP) and a prototype debugging tool. The approach is based on techniques of verification and static analysis originating from logic programming, which are substantially extended in the thesis. The main idea is to verify partial correctness of a program with respect to a given specification which is intended to describe (an approximation of) the call-success semantics of the program. This kind of specification, describing calls and successess for every predicate of a program is known as *descriptive directional type*. For specifying types for CLP programs the thesis extends the formalism of *regular discriminative types* with constraint-domain-specific *base types* and with parametric polymorphism.

Errors are located by identifying program points that violate verification conditions for a given type specification. The specifications may be developed interactively taking into account the results of static analysis.

The main contributions of the thesis are:

- a verification method for proving partial correctness of CLP programs with respect to polymorphic specifications of the call-success semantics,
- a specification language for defining parametric regular types,
- a verification-based method for locating errors in CLP programs,
- a static analysis method for CLP which is an adaptation and generalization of techniques previously devised for logic programming; its implementation is used in our diagnosis tool for synthesizing draft specifications,
- an implementation of the prototype diagnosis tool (called TELL).

Acknowledgments

First and foremost, I would like to express my gratitude to my advisor Jan Małuszyński. His constant encouragement and support during the development of this work have been invaluable. I am very grateful to Włodek Drabent who considerably contributed to the work, especially in solving the theoretical problems. Various ideas presented in the thesis have come up after meetings with people involved in the DiSCiPl project. I am especially thankful to Pierre Deransart for many discussions and improvements he suggested. I would like to thank Marco Comini who has implemented a very first prototype of the diagnoser, a few years ago. Furthermore, I would like to thank Ulf Nilsson and the rest of the members of the Logic Programming Group, and later the Theoretical Computer Science Group, for the creative atmosphere and much help I have received. I am indebted to Jacek Błażewicz and Maciej Drozdowski who made it possible to continue my research while staying at Poznań University of Technology. Special thanks to Jacek Martinek for having introduced me to logic programming many years ago. Finally, I am also grateful to the administrative and technical staff at IDA, especially Lillemor Wallgren and Gunilla Mellheden for their invaluable help with numerous matters during my years as a PhD student.

Linköping, August 2002
Paweł

This work has been supported by the ESPRIT 4 Project 22532 DiSCiPl, and by The Swedish Research Council grant 1999-109.

List of papers

This thesis includes the following five papers:

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

II. W. Drabent, J. Małuszyński, and P. Pietrzak. Locating Type Errors in Untyped CLP Programs. In Hermenegildo et al. editors, *Analysis and Visualization Tools for Constraint Programming*, volume 1870 of *LNCS*,, chapter 4, pages 121–150. Springer-Verlag, 2000.

III. W. Drabent, J. Małuszyński, and P. Pietrzak. Using parametric set constraints for locating errors in CLP programs, *Theory and Practice of Logic Programming*, 2(4–5), pp. 549–611, 2002.

IV. J. Carlson and P. Pietrzak. Improving goal directed bottom-up evaluation of logic programs, 2002. Submitted.

V. P. Pietrzak. The TELL diagnosis tool, 2002. Unpublished.

Contents

Introduction

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 gaining 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 are quite involved. The motivation for the presented work is to facilitate this task.

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 computed result is wrong (i.e. does not correspond to user's expectation),
- a procedure of the program is called with wrong arguments, causing a run-time error.
- the program fails to compute an expected answer,

In a traditional setting, the process of locating an error starts with a symptom observed when running the program on test input data. A systematic testing technique may be needed to find symptoms, however we are not guaranteed to find one.

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 is involved and control flow is not explicitly reflected 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]). Declarative debugging was also extended for lazy functional programming [NF94] and constraint logic programming [Tes96, TF00].

2 The approach

In contrast to the above mentioned approaches, we propose to locate errors in a CLP program without searching for symptoms, that is without executing the program. The idea can be linked to methods for proving partial correctness of a program with respect to a specification, such as [Cla79, Der93, DM88, BC89] (see Section 3.2.1).

Our framework relies on formal program verification. An error searching algorithm tries to construct a proof that the program is correct w.r.t. a given specification. If the proof is obtained then every execution will be free of symptoms violating the specification. Conversely, if a symptom violating the specification can be observed, a proof does not exist. Our specification language is restricted due to decidability reasons, thus it is able to express only certain properties. In a consequence, only some limited class of errors can be captured. Nevertheless, in an incorrect program our tool finds all such errors.

To use the tool the user need not be familiar with the underlying program verification techniques.

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

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

We focus on the question whether incorrect procedure calls and wrong answers may appear in some computations of a given program. The specification provided by the user describes a superset of the expected calls and a superset of the expected answers in computations of the program. The program may or may not satisfy these expectations. The above mentioned sets are described in terms of parametric types such as lists, trees, etc. Such descriptions of calls and successes are often called *directional types* [AL94, Boy96]. The language of types is easy to understand and allows efficient checking of the verification conditions. On the other hand, this choice restricts the considered errors to type errors.

We deal with untyped CLP languages and in contrast to a common practice in typed languages we do not require types of program constructs to be specified a priori. Moreover, in our approach we try to find a reason, that is a fragment of a program, responsible for the type specification being violated.

The type specification is usually provided in a step-wise interactive way. At each stage of this process the program is checked against the fragment of the specification at hand. So incremental building of the specification is coupled together with locating errors. Even small fragments of the specification are often sufficient to locate (some) errors in the program. On the other hand, if no program errors have been located when the specification is completed then the program is correct (w.r.t. the specification).

In the proposed methodology the process of type specification is usually preceded by static analysis which infers directional types of the program. The inferred directional types approximate the semantics of the program at hand. They may also provide indication that the program is erroneous. In this case the user may decide to start the process of specification and error location. The results of the type inference may facilitate it, namely the inferred types may be thought of as a draft for the target specification. Thus, in our methodology type inference plays only an auxiliary, though useful, role.

The methodology is supported by a prototype error locating tool. The present version of the tool works for a subset of SICStus Prolog [SIC98] including constraints. Previous version was tailored to the constraint programming language CHIP [Cos98]. A part of the tool has been integrated with CIAO Preprocessor [BLGP$^+$99, PBH00b]. The tool can be easily adapted to handle other CLP languages.

The structure of the tool is illustrated in Fig 1.

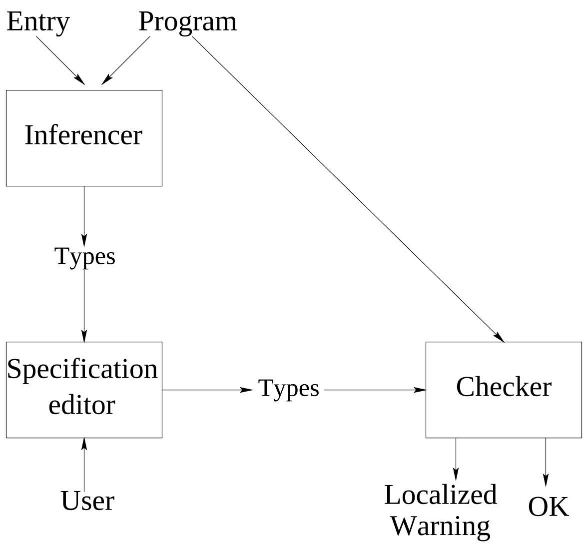

Figure 1: The structure of the error locating tool

The tool includes a type checker, a type inferencer and a specification editor. The tool has also a library of predefined types. Among others, the library provides definitions of often occurring types and specifications for built-in predicates. The specification of a program is introduced through the editor. It may refer to types defined in the library and/or to application-specific types, defined a priori by the user.

The input consists of a CLP program and of an *entry declaration*. The latter is a specification of intended (atomic) initial calls. Thus, the user may explore different ways of using the program. The type inferencer constructs directional types for all predicates of the program, thus providing a specification such that the input program is correct w.r.t. it. However, these types may not correspond to the user's intentions. This is due to program errors or to inaccuracy of type inference.

The intended types have to be provided by the user. They are introduced in a step-wise interactive manner. When providing the type of a predicate the user may first inspect the inferred type and accept it, or specify instead a different type. The tool monitors the process and immediately reports as an error any violation of the verification conditions for the so far introduced types.

3 Background and related work

In this section we provide an overview of works related to the material contained in the thesis. We point out similarities and differences (wherever applicable) in order to justify the contributions of the thesis.

We start with a brief and informal introduction to constraint logic programming. In Section 3.2 verification and debugging techniques are shortly discussed. The verification frameworks presented in Section 3.2.1, as well as abstract diagnosis mentioned in Section 3.2.3, are directly related to our work. We decided to provide also some introductory material on declarative debugging in Section 3.2.2, since there are some similarities to our work. We however are not going to mention works on debugging by tracing, even though their principal goal is the same as ours. We found tracing as entirely different from what is presented in the thesis and thus unrelated. Section 3.3 describes main approaches to types in (C)LP. Finally, Section 3.4 surveys various methods of approximating program semantics by types.

3.1 Constraint Logic Programming

This thesis is focused on finding errors in constraint logic programs (CLP). Constraint logic programming is a generalization of logic programming. Likewise a logic program, a pure CLP program is a set of logical formulæ called clauses[1]. However, a logic program is usually interpreted over the Herbrand domain, while a CLP program is interpreted over specific *constraint domain* which need not be the Herbrand domain.

A CLP program describes relations between objects from the constraint domain. The user-defined relations correspond to *predicates* on a syntax level, and are defined by a set of clauses. *Constraints* are basic relations, such as equality, inequality, etc., between elements of the domain. Predicates denoting constraints are sometimes called *constraint predicates.* In some CLP system the programmer may define his own constraints, by using meta-level facilities.

A CLP program has two main semantics. The first one is called *declarative semantics.* It is characterized by a least model of a program (the program is a set of formulæ). This model is an interpretation that agrees with semantics of constraints. It is well-known that the least model always exists. Moreover, it coincides with a least fixed point of a monotonic operator (called *immediate consequence operator*), which is a function on interpretations. For a given set of known facts, the operator computes new facts that immediately follow from the known ones and from the program (see e.g. [JMMS98], or [Llo87b] for the logic programming case).

The way in which programs are executed is described by *operational semantics.* This semantics is usually described as a transition system, which in the case of logic program becomes SLD-resolution (or SLDNF-resolution if negation is admitted). An ultimate goal of executing a CLP program is to solve a set of constraints (i.e. computing one or more valuations of variables that make the constraints true) or to show that the constraints are unsatisfiable (i.e. that there is no solution). A mechanism that provides an algorithm for solving the constraints is called a *constraint solver.* Constraints in logic programming are equations over the algebra of terms (called also the Herbrand domain). The equations are solved by unification. Almost every CLP language incorporates terms and the unification algorithm co-exists with solvers specific to the domain (e.g. finite domain numbers, rational numbers, boolean). The operational aspect of CLP is also captured by call-success semantics. This semantics consists of two sets of *constrained atoms* (see Paper III [DMP02]) : a set of call patterns and a set of success patterns. We use this framework as a semantic basis of our verification method.

The foundations of logic programming can be found in [Llo87b, Apt90]. An introductory material on CLP can be found in [JM94, MS97], and semantic aspects of CLP in [JMMS98].

[1]See for instance Paper III, Section 2.3.

3.2 Verification and debugging

The thesis is related to the central point of program development which is obtaining a program that satisfies the programmer's intentions. In particular this means that whenever the program is called in the intended way and terminates, the obtained results conform to user's expectations. In other words, we want the program to be *partially correct*[2]. To be able to verify the program, the programmer must express the intentions in a formal way. Such a formally expressed intention is called a *specification*. The specification is an approximate description of the program's semantics.

The specification is used in a *verification* process. The verification boils down to checking sufficient conditions. This means that if the verification succeeds then it is assured that the program is correct. Otherwise, there is an indication of an error in the program. Note also, that if the program is incorrect then the verification fails. The properties of interest expressed in the specification may be of different nature. Therefore different verification methods (different forms of the sufficient conditions) have been devised.

The program may be also found erroneous without formal verification. This happens if program's behavior does not fit to the user's (possibly informally expressed) intention. In either case a debugging action has to be taken.

A process of locating (a piece of code that contains) a bug is called *diagnosis*. By *debugging* we mean a process that involves diagnosis and removing bugs from the program. Sometimes verification techniques can be used to locate bugs, likewise in our thesis.

Various semantic frameworks and verification methods can be used. In this thesis we use call-success semantics and approximate specifications. See [BDD$^+$97] for a general discussion on approximations in verification and debugging.

The works surveyed below concern logic programming.

3.2.1 Verification frameworks

The general verification methods proposed for logic programming can be roughly classified with respect to the semantics considered.

Declarative properties

Declarative approaches to verification deal with declarative semantics and can thus handle declarative properties. We abstract from the execution model, which can be seen as an advantage, but on the other hand we cannot reason about run-time properties such as variable instantiation.

[2]Throughout the thesis we will often use the term "correct" meaning "partially correct", as we do not deal with termination and with program completeness.

Clark [Cla79] and Deransart [Der93, DM93] consider proving declarative properties of definite programs (i.e. with no negation in clause bodies). Proving correctness (and completeness) of normal programs (i.e. with negation) has been studied by Ferrand and Deransart [FD93] and Malfon [Mal94] and recently advocated in the work of Drabent and Miłkowska [DM01].

Operational properties

Often, we are interested in runtime properties, such as characterization of call patterns. Verifying the call patterns allows to detect possible run-time errors at compile time. Approaches dealing with these properties take into account a particular execution model, typically LD-resolution[3]. They are inspired by the classical verification methods devised for imperative programming [Hoa69, Flo67].

In the thesis we extend the *inductive assertion method* (originally developed for logic programming) of Drabent and Małuszyński [DM88, Dra88] to CLP. 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] (see also [Apt97]). 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.

Pedreschi and Ruggieri [PR99] present an attempt of a uniform verification framework that, apart from partial correctness covers other aspects such as termination. Their approach also applies to normal programs.

Deriving proof methods by abstract interpretation

Abstract interpretation [CC77] is one of static analysis methods that enables compile-time predicting certain properties of a program execution. The properties captured by the analysis are described using an *abstract domain.* In order to achieve safeness, i.e. the relevant results of the analysis, it is required that the abstract and concrete domain are related my means of a formal construction called *Galois connection.* Abstract interpretation is used e.g. in compilers - to produce efficient code, in software validation or software validation.

The papers [LV98, CGLV00] apply abstract interpretation to verification. They demonstrate that sufficient conditions for verification of logic programs can be systematically derived if the considered class of specifications is defined as an abstract interpretation domain with Galois connection relating them to the semantics (procedural or declarative) of logic programs.

[3] I.e. the SLD-resolution with the leftmost selection rule

Unfortunately, as shown in [DP98] (Paper I of this thesis), such a Galois connection does not exist in our case of specifications being regular sets of terms, so the method is not directly applicable.

3.2.2 Declarative debugging

Locating program errors by simple tracing is particularly tedious in declarative languages, mainly due to complex execution which is not clearly reflected in a text of the program. A systematic technique for locating errors, which is particularly suitable for logic programming, is *declarative* or *algorithmic* debugging proposed in [Sha82] for logic programs. A debugging session starts with a *symptom* which can be either an incorrect answer, or a missing answer of the program, i.e. this concerns a particular test computation that does not conform to the user's requirements. Then, the declarative debugger seeks for an *error* by comparing elements of the actual semantics involved in the computation of the symptom, with user's expectations. The user's expectations are of the declarative nature, i.e. the user is supposed to know what has to be computed, but not how it has to be done. The user's requirements are acquired by questioning the user in an interactive process. Some of the questions may involve big terms and therefore may be complex. The interactive debugging session is a kind of a searching process which eventually leads to finding an error, i.e. the part of the program causing the symptom.

This approach is related to ours in the sense that the answers provided by the user to the diagnoser, may be seen as a (partial) program specification. However, in contrast to our method, the declarative debugging session starts with a symptom and concerns a single (test) execution only.

The formal properties of declarative debugging, in case of definite programs, were investigated by Ferrand [Fer87], and results for general programs are given in [Llo87b, Llo87a]. Naish [Nai91] shows further improvements in order to cope with missing answers in Prolog programs. Declarative debugging was also adapted for constraint logic programming [Tes96, TF00]. Pereira [Per86] developed a framework that adds an operational aspect to declarative debugging.

The declarative (or algorithmic) debugging has been also adapted to lazy functional languages [NF94] and imperative languages [KSF90, Sha91].

Observe, that in the declarative debugging scheme the user, sometimes called an oracle, is supposed to answer any detailed question about the specification of the program. This is often an unrealistic assumption. One way to reduce the number of queries and to simplify them is to apply partial formal specifications in form of assertions [DNTM88, DNTM89, BDM97]. Then, whenever a query can be answered by looking up the assertions, it not posted to the user.

Another approach to simplify the user's effort in the debugging session was taken by [LS88]. The idea is to use abstract semantics and abstract oracle, that approximate, respectively, the program's semantics and the programmer's intention. This makes is possible to ask questions that are easier to answer by the programmer.

3.2.3 Abstract diagnosis

A technique closely related to our approach is *abstract diagnosis* [CLV94, CLMV96, Com98, CLMV99]. It has been thought of as an extension of declarative debugging. The specification approximates declarative (intended) semantics of the program, which is given by the semantic function (the monotonic operator mentioned in Section 3.1). Then an *abstract semantic function* is designed that safely approximates the semantic function. Unlike the declarative debugging method, abstract diagnosis is not based on the driven-by-symptoms interaction with the programmer. Instead, it is assumed that a complete (approximate) specification is given. Then, the abstract semantic function is used in an inductive proof of the program's correctness.

Abstract diagnosis is a generic approach that heavily uses an *abstract interpretation* framework based on Galois connection [CC77]. This may be problematic, as sometimes for interesting properties, such as regular types in our thesis, Galois connection does not exist (see Paper I, [DP98]). Despite of that we can still perform a similar kind of verification.

Our technique is restricted to a particular specification language, as we believe that regular descriptive types (see Section 3.3.2) form a natural, easy-to-use, and expressive enough formalism for writing specifications. Moreover our method also concerns finding an error rather than just stating if the program is or is not correct.

3.3 Types in constraint logic programming

We are interested in static (compile-time) detection of errors. In principle, introducing *types* to programming languages, (see e.g. [Car97, CW85]) has a similar motivation. 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.

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.

If a language is untyped then we can use types as a specification and apply a program verification technique. Such a technique can be viewed as a typing mechanism. A programmer can still provide type declarations with essentially the same motivation as in typed languages, i.e. to make sure, that variables will take appropriate values. It is also possible to infer such types from a declaration-free program, to find out what values the variables will take.

3.3.1 Descriptive vs. prescriptive types

There has been a considerable effort to introduce types into logic programming. Some of those works concern CLP. The proposed approaches can be divided into two main streams: descriptive and prescriptive.

The descriptive approach concerns untyped languages. Types are meant here as sets of values, or more specifically, sets of terms. We may say that descriptive types have semantic nature. A purpose of introducing types is to approximate the program's semantics or/and user's intentions. Whether the semantics is operational-based (call-success) or declarative is another issue. An important aspect is that the program has the same semantics with or without types.

Unlike in the descriptive approach, in the prescriptive one types are part of the underlying language. Type constitute a part of the syntax, and as such are verified syntactically, without referring to the semantics. If typing rules are violated then the checked program does not conform to the syntax, thus does not belong to the language and consequently, is not a program. If types are checked successfully then the program has semantics that takes the types into account, namely semantics of typed logic with parametric polymorphism, and possibly with a subtyping relation.

3.3.2 Descriptive types

Describing descriptive types

Descriptive types, that are present in logic programming, are sets of terms. Types ought to be decidable and often are required to be closed under substitution (i.e. if a term belongs to the type then all its instances do). All the known descriptive approaches use *regular sets of terms* as types. Regular sets of terms are recognized by tree automata [GS97, CDG$^+$97]. However in the context of typing logic programs other equivalent formalisms have been applied:

- *regular term grammars* - introduced by Mishra [Mis84]. He used the grammars in a *discriminative* form (called therein *tuple-distributive*). This kind of a grammar corresponds to deterministic root-to-frontier tree automata.

  Dart and Zobel [DZ92] proposed *non-discriminative* regular type languages[4].

[4] This formalism is equivalent to non-deterministic root-to-frontier automata or to either form of frontier-to-root automata.

- *RUL programs and uniform programs* - is a syntactically restricted form of a logic program; RUL programs were devised by Yardeni and Shapiro [YS91] (describe discriminative types) and followed by [GdW94] and later adapted to constraint logic programming in [DP98] (Paper I) and [DP99]. [FSVY91] introduced uniform programs that express non-discriminative types.

- *set constraints* - initiated by Heintze and Jaffar [HJ90b, HJ90a, Hei92b, Hei92a, HJ94]. Set constraints are inclusions between set expressions built up by: variables (ranging over sets of trees), tree constructors (interpreted as functions over sets of trees) and a set of operators specific to a particular class of set constrains. The reader is referred to [PP97] for a brief survey.

- *type graphs* - introduced by Janssens and Bruynooghe [JB92] and then taken by [HCC95, Mil99]. As stated in [HCC95] type graphs can be seen as data-structure to represent regular tree grammars.

Types based on declarative semantics

A notion of descriptive (regular) types has been introduced to logic programming by Mishra [Mis84]. His work was based on the least Herbrand model semantics, i.e. the types approximate either the program's semantics or the user's intention. The problems considered were how to check that the least model semantics is included in a regular set of terms (the type checking problem) and how to approximate it by regular sets (the type inference problem) (see Section 3.4).

Yardeni and Shapiro [YS91] show a type checking method which is essentially an inductive proof using an abstract semantic function and types as an abstract domain (cf. abstract diagnosis, Section 3.2.3). Their work is similar to ours, however their type language is not as expressive as ours as it does not handle constraints. Moreover, this framework uses different semantics, namely declarative.

Directional types

The approaches based on the least model semantics do not take into account the intended use of the predicates and give therefore few possibilities for finding typing errors. The focus is mostly on detecting that for some predicates the inferred types are empty sets in which case the predicates never succeed or that success types do not violate (in a given sense) user's type declaration. Moreover, approximating declarative semantics is not suitable for some purposes, such as reasoning about call-patterns.

This observation led to the notion of *directional types*.

Checking of directional types based on set constraints was discussed in [AL94]. The types used are sets of non-ground terms[5]. They are specified by set constraints together with a lifting function Sat that maps a set of ground terms to a set of not necessarily ground terms. Type checking is based on the same verification condition we use, which in general form originates from [DM88, BC89] and was specifically formulated for directional type checking in [Apt93] (see also [AM94, Apt97]).

Charatonik and Podelski [CP98, Cha00] re-use procedures of [FSVY91] for directional type checking and inference. Following [FSVY91] they characterize types by the least model of a uniform program. The directional types are regular, but in general, not discriminative.

Boye [Boy96, BM97] identified two aspects of directional types. First one reflects passing values along the clause bodies in the left-to-right execution model. This resembles our method. Besides this he uses types for proving program correctness with the (declarative) *annotation method* of [Der93]. This technique does not depend on operational semantics, and therefore can handle some cases that methods based on simplified execution model cannot. It is however not useful if we are interested in reasoning about call-patterns.

Another perspective on declarative aspect of directionality in logic programming was given by Naish [Nai96]. Since his approach is restricted to ground terms it was possible to apply model theoretic semantics, rather than proof theoretic notions of [Boy96, BM97]. However, no specific algorithms are given.

Polymorphism in directional types is discussed by Boye [Boy96]. Despite of different semantics and verification framework that he uses, the verification conditions have a similar nature to ours and give rise to similar parametric set constraints. Our work [DMP02] (Paper III), [DMP01] goes further in that we use such parametric set constraints in a sufficient correctness test, and also for type inference, while the simplification techniques of [Boy96] are rather limited in handling parameters.

The problem of polymorphic directional type checking is also addressed in [RT00, RT01]. This work presents a formal system, where directional well-typing of a logic program for given type specification is defined in terms of proofs constructed from given axioms and typing rules. This is different from our approach where the well typing algorithms are derived from the semantic concept of program correctness and parametric types are understood as families of sets. Nevertheless, the semantics of types as sets is also provided in [RT00]. It is done by a fixpoint construction, which for a given alphabet of typed function symbols associates each used type with a subset of the Herbrand universe.

[5]I.e. terms with variables. Terms without variables are called *ground*.

3.3.3 Prescriptive types

Prescriptive approach to typing logic programs dates back to [MO84] where the Milner's type system with *parametric polymorphism* [Mil78], originally developed for functional programs, has been adapted to Prolog. In this approach the function symbols and the predicates of a logic program are supposed to have a priori declared types. The types are used to restrict the syntax of the language to *well-typed* formulæ. Semantically, prescriptive typing corresponds to taking many sorted typed logic as a foundation of logic programming, instead of untyped logic. Semantic foundations for typed logic programs are given in [HT92].

The framework of Mycroft and O'Keefe [MO84] does not support subtyping. It has been demonstrated that well-typedness of a program in this kind of type system can be fully checked at compile-time. The restriction for type checking that has to be fulfilled is so-called the *head condition* [HT92] (also called *definitional genericity* [LR91]). It says that the types of the arguments of a clause head must be equivalent up to renaming to the types declared for the predicate. Then, the successfully verified program can be executed using the standard (untyped) SLD-resolution engine, assuring that every resolvent in every computation is well-typed (i.e. no run-time type checking is needed). This property is summarized by a well-known slogan: "Well-typed programs do not go wrong". Deransart and Smaus [DS01] strengthened this result by referring to tree-grammar semantics (which is declarative) and by weakening the form of the head condition.

The Mycroft and O'Keefe type system provided semantic basis for logic programming languages as Typed Prolog [LR91], Gödel [HL94] and Mercury [SHC96]. Polymorphic types without subtype ordering have been also studied by Hanus [Han91]. His system does not impose the head condition, i.e. the types of the arguments of a clause head may be instances of those declared by the user[6]. This makes it possible to support higher-order programming. A cost for this flexibility is that types have to be checked at run-time.

Type checking and type inference become more difficult if subtyping is admitted. In general typed unification is required. Many approaches have been proposed over last several years. Smolka [Smo89] and later Hill and Topor [HT92] defined frameworks for order-sorted types with parametric polymorphism in which head condition is present. In both papers the authors consider the problem of type inference (or type reconstruction), i.e. finding the most general typing for variables in the program for given types of predicates and function symbols. Their algorithms are incomplete and fail to find a most general type in some cases. This was demonstrated by Beierle [Bei95].

In all the above approaches subtyping was allowed only between type

[6] Hanus has made a similar improvement for type systems with subtyping [Han92].

constructors with the same arity. Fages and Coquery [FC01] define a type system where this restriction is not imposed. They however keep the head condition, claiming that this is a natural choice.

As already stated type systems with subtyping require an execution model augmented with a typed version of the unification algorithm. However, under some assumptions it is possible to execute the program using an untyped model. This result is due to [SFD00] and it is limited to logic programs with a dataflow given by mode declarations.

3.4 Approximating semantics with descriptive types

As already mentioned, descriptive types, are commonly used for approximating the semantics of a program at hand. Most of the proposed approaches use semantic program analysis techniques [NNH99]. Others apply program transformation methods or type inference rules. The relation among some of them is studied is [HJ92].

3.4.1 The techniques based on abstract interpretation and data flow analysis

Without type definitions for function symbols

Kluźniak [Klu88] initiated application of data flow analysis for type inference. The method was devised for Ground Prolog and was further improved and put in more formal framework by Kågedal [Kåg93, Kåg95]. This method has been further extended to Logic Programming by Tan and Lin [TL96].

A different approach to type inference, based on the generic (top-down) abstract interpretation framework of [Bru91], has been introduced in [JB92]. As already mentioned, this method uses *type graphs* to describe types that correspond to possible instantiations of program variables. The type graphs form possibly infinite domain, thus a straightforward fixpoint iteration algorithm may not terminate. To deal with this one can apply a *widening operator* of [CC92a], which computes an upper bound of the intermediate results found in the recent iteration step (thus guarantees safe approximation) and makes sure that every sequence of computed (intermediate) results is finite (thus, the iteration converges).

An efficient implementation of type analysis, based on the above work, is described in [HCC95].

Gallagher and de Waal [GdW92, GdW92] compute descriptive types that approximate declarative semantics. Following [YS91] types (called in [GdW94] *regular approximations*) are represented as RUL programs. In fact, the construction from [GdW94] can be seen as an adaptation of a similar solution of [YS91] and augmenting it with widening (called in [GdW94] shortening). Gallagher and de Waal compute also call and success types (i.e. a certain form of directional types approximating call-success semantics), by combining the method for declarative semantics with *magic transformation* [BMSU86][7].

This approach to type analysis has been adapted and extended to constraint logic programs in our work.

In [Mil99] Mildner has studied the regular type domain from various perspectives. He surveyed many widening operators (including those of [JB92, HCC95]), improving some of them in terms of a trade-off between precision and efficiency. He also pointed out that the shortening function of [GdW94] may not terminate. Therefore in our analyzer we implemented one of the correct Mildner's proposals to widening.

Gallagher and Puebla [GP02] proposed an abstract interpretation over non-deterministic tree automata. They exploit an observation due to Cousot and Cousot [CC95] that for a particular program one may build a finite domain of tree automata (or regular tree grammars), and thus make sure that the analysis (a fixpoint iteration) terminates.

With type definitions for function symbols

These approaches demand signatures for function symbols to be given prior to the analysis. This requirement makes that some researchers call the approach prescriptive typing (i.e. [Lu00]), since types for function symbols must be know a priori. On the other hand, the types for the function symbols may be seen as a basis for an abstract domain, fixed for a particular program or for a particular set of function symbols. It is then used for static analysis, whose ultimate goal is to approximate the program's semantics. We would like to stick to the latter point of view.

This line was initiated by [KH85, HK88]. For given types for function symbols the analysis infers procedure types for calls and successes.

Barbuti and Giacobazzi [BG92] proposed a natural method of inferring types, which is based on abstracting declarative semantics. It turns out however, that this technique is able to approximate only a "well-typed" fragment of the program's least model. So this is essentially a prescriptive approach to types. Nevertheless, due to many similarities to other techniques listed in this section, we decided to mention the work here.

A technique based on manipulation on the *Prop* domain [MS93][8] was proposed in [CD94]. It was further refined in [CL00] by applying an elegant theory of ACI-unification (associative, commutative and idempotent). The approach allows type variables in the domain, and thus to express polymorphic dependencies.

[7]Magic transformation is a method that aims for simulating top-down execution of the program and a given goal, using a bottom-up evaluation engine. For discussion about applying magic transformation to program analysis see [DR94, Nil95, CD93].

[8]*Prop* has be designed to capture information about groundness of variables and sharing between variables.

Abstract interpretation technique has been also used by Lu [Lu98] to infer polymorphic types.

In all the above approaches types were discriminative. Non-discriminative types inference is studied by Lu [Lu00].

3.4.2 Other approaches

Program transformation methods

These approaches use logic programs to represent types. Frühwirth [Frü89] proposed a method for type inference which is based on two source-to-source program transformations: partial evaluation and projection. Combining these two transformations results in a program with restricted syntax (resembling RUL or uniform programs) and semantics that approximates the semantics of the original program. Thus, the result may be seen as types that approximate the program semantics.

A related technique, that uses different form of logic programs (uniform programs) to represent types, was studied in [FSVY91]. A uniform program is derived from the original program, so this approach can be viewed as a program transformation technique. The paper shows different approximations of declarative semantics and establishes formal connections to other approaches to descriptive types, like [YS91, HJ90b].

Syntactic rules methods

There have been few works on inferring descriptive types, which are based on type inference calculus, reminiscent to type inference rules in functional languages. The first paper by Mishra [Mis84] can be seen as an example of such approach. Then, Zobel [Zob87] tried to extend the method to handle polymorphic types.

This work was later continued by Pyo and Reddy [PR89] who used a notion of the *implication type constructor* due to [Red88].

Set constraints approaches

Applying set constraints to analyze logic programs has be studied for the first time by Heintze and Jaffar [HJ90b, Hei92b]. They approximate declarative semantics by first, extracting set constraints from the program, and solving them.

Inference of directional types in the framework of set constraints was illustrated by an example in [Hei92a]. (The main topic of the paper are implementation techniques for solving set constraints.) In the example the types are inferred by constructing set constraints analogous to our encoding of verification conditions, and solving them. Inferencing directional types for logic programs is also discussed in [CP98].

In [DMP02] (Paper III) we use sets constraints to formulate the problem of semantic approximation and program verification. We use however some techniques specific to abstract interpretation.

4 The structure of the thesis

The thesis is a collection of five papers. Below, a brief overview of them is given.

Paper I

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

This paper introduces the call-success semantics for CLP and provides verification conditions for it. A (sophisticated) type system for CLP is defined. We also modify the technique of Gallagher and de Wall [GdW92, GdW94] of bottom-up abstract interpretation to synthesize approximations of the call-success semantics of a given program. Our prototype program analyzer works for the programming language CHIP. The paper is a full version of [DP99].

Paper II

W. Drabent, J. Małuszyński, and P. Pietrzak. Locating type errors in untyped CLP programs. In Hermenegildo et al. [DHM00], chapter 4, pages 121–150.

This paper presents a *static diagnosis* tool that locates type errors in CLP programs without executing them. The tool automatically locates at compile time all the errors (with respect to a given type specification) in a program.

The presentation is informal. The focus is on the motivation of this work and on the functionality of the tool.

Paper III

W. Drabent, J. Małuszyński, and P. Pietrzak. Using parametric set constraints for locating errors in CLP programs, ***Theory and Practice of Logic Programming*****, 2(4–5), pp. 549–611, 2002.**

This paper introduces a framework of parametric descriptive directional types for CLP.

Set-constraint techniques are used for formulating and checking verification conditions for (parametric) polymorphic type specifications. The specifications are expressed in a parametric extension of the formalism of term grammars. The soundness of the method is proved and the prototype debugging tool supporting the proposed approach is illustrated on examples.

The problem with termination inherited from [GdW94] has been also solved.

The paper is a full version of [DMP01] and it is a substantial extension of the previous work [CDMP98, CDP99, DMP00b, DMP00a] concerning monomorphic directional types.

Paper IV

J. Carlson and P. Pietrzak. Improving goal directed bottom-up evaluation of logic programs, 2002.

This paper describes a new strategy for goal directed bottom-up evaluation of logic programs. The method is based on a combination of two known techniques: dividing a program into strongly connected components and the Induced Magic-sets technique of [Cod99]. The impact of the method is illustrated by experimental results.

Paper V

P. Pietrzak. The TELL diagnosis tool, 2002, unpublished

This report presents the recent version of the diagnosis tool, that handles parametric polymorphism. It describes the architecture and usage of the tool. Some functionalities described in the paper are not yet implemented.

5 The contributions

As already pointed out, the main idea behind our method is to combine verification techniques with types as a specification language. Various methods have been devised/adapted contributing to the approach at different levels.

- **The verification method**

  The starting point are well-known verification conditions for partial correctness of LP w.r.t. a specification which gives a set of procedure calls and a set of procedure successes. The question is what is the semantics suitable for description of calls and successes in CLP, and whether the verification conditions of LP can be adapted for this semantics. In **Paper I** [DP98] we introduce the call-success semantics for CLP and provide verification conditions for it.

  In **Paper III** [DMP02] this verification method is extended to parametric specifications, which can be viewed as a family of non-parametric ones.

- **The specification language**

  In **Paper I** [DP98] a specification language of regular types has been extended to the CLP case. While types in LP are sets of atoms, their counterpart in CLP are sets of constrained atoms. We adapted regular types for CLP so that one can describe sets of constrained terms and atoms. This results in a rather complicated type system. We define the needed type operations and prove their properties.

  Since our experience had shown that the full power of the type language of **Paper I** [DP98] is not necessary, we simplified it in **Paper III** [DMP02], giving up some of its expressiveness, yet keeping it useful. We further extended it in another dimension, namely to handle parametric polymorphism. We define a notion of a parametric regular term grammar together with appropriate operations.

  Once the parametric polymorphism has been introduced to the type language, it is possible to express and to reason about polymorphic properties of some predicates, such as `append/3`, which operates on three lists of the same type not known a priori.

- **Locating errors**

  Errors are fragments of a program that make it incorrect w.r.t. the specification. The verification method combined with the specification language of types is thus used to locate the errors. Whenever a correctness proof of a program cannot be constructed, the possible smallest parts of the program which are the reasons why the verification fails, are found and considered erroneous. This technique is described and illustrated by a number of examples in **Paper II** [DMP00a], and also in [CDMP98, CDP99, DMP00b]. Correctness of the method is demonstrated.

  Again, the parametric extension carries to locating errors, as described in **Paper III** [DMP02]. It is shown therein that parametric polymorphism brings more accuracy and allows us to detect more errors.

- **Synthesis of a specification**

  Another part of our framework is an algorithm that computes a specification for a given program. This is a specification for which the program is correct, i.e. the specification approximates the call-success semantics of the program. As explained in Section 2, this aims at minimizing the user's effort in providing the specification. An automatically generated specification may be understood as a draft for the target specification.

  Our approach to computing a call-success approximation of a program is primarily described in **Paper I** [DP98]. The basic idea is to adapt an algorithm of [GdW94] to handle an extended language of types. Furthermore, a bottom-up fixpoint iteration is applied (similarly to what was done in [GdW94]) to so-called *magic programs* - a result of the *magic transformation*. Consequently, it gives an approximation of the call-success semantics.

A starting point of such an analysis is (a type of) a top goal. For some programs a type of a top goal may contain type parameters. Our analysis is able to handle parameters originating from the top goal, as described in **Paper III** [DMP02].

Unfortunately, as pointed out in [Mil99], the original algorithm of [GdW94] may not terminate in some circumstances. In **Paper III** [DMP02] we have corrected this error by adaptation of the Mildner's solution.

- **Implementation**

  The implementation of the diagnosis tool consists of three main parts:

  - the verification module,
  - the type analyzer,
  - the user interface.

  Two first modules share basic procedures operating on term grammars, such as inclusion test, computing intersection, union, and so on. The modules work together as described in Section 2. More detailed description of the operating mode, sample diagnosis sessions, the structure of the tool etc. can be found in **Paper II** [DMP00a] and in **Paper V**.

  The prototype of the diagnosis tool (called TELL) handles a subset of SICStus Prolog [SIC98], including constraints. The previous version supported the programming language CHIP [Cos98]. The diagnoser has been implemented in SICStus Prolog. The type analyzer module has been also incorporated to the CIAO system [BCC$^+$97] as a part of the generic preprocessor [BLGP$^+$99, PBH00b]. The GUI part employs Tcl/Tk.

- **Improvements of the analysis**

  In **Paper IV** [CP02] we introduce an improvement of the formerly known techniques of (approximate) evaluation of logic programs, based on magic transformation. The proposed strategy is a combination of two known techniques: dividing a program into strongly connected components (SCC) and Induced Magic-sets [Cod99], which is an efficient evaluation strategy derived from magic transformation.

We have implemented the strategy as a generic engine, parameterized by the (abstract) computation domain. We have conducted experiments with the type domain, illustrating an impact of the method.

Paper I

Inferring call and success types for CLP programs

Włodzimierz Drabent and Paweł Pietrzak

Abstract

This paper proposes a tool to support reasoning about (partial) correctness of constraint logic programs. The tool infers a specification that approximates the semantics of a given program. The main intended application is program debugging. We deal with a "call-success" semantics of CLP. We consider a restricted class of specifications, which are regular types of constrained atoms. The call-success semantics of a CLP program is characterized by the declarative semantics of another CLP program ("magic transformation"). Then bottom-up abstract interpretation is used to approximate the latter.

We study the theoretical background of this approach. We are mainly interested in applying it to CLP over finite domains. Our prototype program analyzer works for the programming language CHIP.

1 Introduction and motivation

The work reported addresses the problem of correctness of CLP programs. Intuitively, a program is correct if it behaves as expected by the user. But user expectations are seldom well documented. This paper describes an analyzer that for a given CLP program produces a characterization of the form of calls and successes in any execution of the program starting from a given class of goals. The user may inspect the description produced to see whether it conforms to her expectations. The language of description has thus to be relatively simple in order to allow effective automatic inference and to be easily understood by the user. This paper presents a concretization of this idea and a tool based on it. This is an extension of our previous work [DP99].

The approach proposed is an adaptation of the existing static analysis techniques for logic programs to the needs of CLP. The results adapted and the contributions of this paper are as follows.

The starting point are well-known verification conditions for partial correctness of LP wrt to a specification, which gives a set of procedure calls and a set of procedure successes. (Such verification conditions were proposed in [DM88, Dra88]; a useful special case was given in [BC89, AM94]). The question is what is the semantics suitable for description of calls and successes in CLP, and whether the verification conditions of LP can be adapted for this semantics. This paper introduces the call-success semantics for CLP and provides verification conditions for it.

Generally the conditions are undecidable. But they become decidable for a restricted class of specifications. As shown by Boye [Boy96], in the case of LP it is sufficient to consider specifications describing regular tree sets. In the literature this kind of specifications is often called regular types [YS91, DZ92]. While successes and calls in LP are atoms, their counterpart in CLP are constrained atoms. Therefore this paper adapts regular types for CLP so that one can describe sets of constrained terms and atoms. This requires adaptation of the operations on regular types. We define these operations and we prove their properties.

Finally we need static analysis techniques. We show that the verification conditions for a CLP program constitute another CLP program whose declarative semantics describes the calls and successes (Such approach is often called "magic transformation"). For this purpose we introduce a generalization for CLP of c-semantics [FLMP89]. This results in more precise descriptions than using the standard $\mathcal{D}$-model semantics. We adopt then the technique of Gallagher and de Wall [GdW92, GdW94] of bottom-up abstract interpretation to synthesize approximations of the call-success semantics of a given program. As a side effect we obtain a tool to approximate the declarative semantics of CLP programs. We deal with partial correctness, the approximations are supersets of the actual semantics of programs. (A wider class of approximations is discussed in [BDD$^+$97]).

Using of types, as in our work, to approximate the semantics of programs in an untyped language is usually called *descriptive* typing. Another approach is *prescriptive* typing. In that approach the type information influences the semantics of a program. In particular, variables are typed and may only be bound to the values from the respective types. Prescriptive typing is a basis of a few programming languages (e.g. TypedProlog [LR91], Gödel [HL94], Mercury [SHC96]). In contrast to prescriptive typing, our approach is applicable to any (C)LP language.

We implemented a type inference tool which is a major modification of the LP analyzer of Gallagher and de Waal [GdW92, GdW94]. Our first prototype has been implemented in CHIP for analysis of CHIP programs. Subsequently it has been ported in cooperation with UPM Madrid and with COSYTEC to other platforms (SICSTUS, Ciao) for off-line analysis of CHIP programs. It is possible to modify it for analysis of other CLP languages. The type-inference tool is also used as a part of our type-based diagnoser for CHIP which automatizes localization of type errors [CDP99, CDMP98].

The paper is organized as follows. The next section summarizes basic concepts of CLP and presents the declarative and the operational semantics. Then we propose a system of regular types for CLP. Section 4 describes the type inference method used in this work. Then we present an example of type analysis for CHIP.

2 Semantics of CLP

In this section we present the approach to CLP semantics used in our work. We want to express groundness information in our semantics. We have to take into account that most of implementations of CLP use a semantics with syntactic unification[9]. As the declarative semantics we will use a generalization of c-semantics [FLMP89] to CLP. The operational semantics is given by sets of call and success instances of atomic goals.

After recalling some basic notions of constraint logic programming we introduce the declarative and operational semantics used in this work.

2.1 Basic concepts

We consider a fixed constraint domain. It is given by fixing a 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[10].

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 paper by a *constraint* we will mean a primitive 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.

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.

---

[9] Many actual implementations of CLP use syntactic unification: function symbols occurring outside of constraints are treated as constructors. So, for instance, $5-1$ and $2+2$ are not unifiable.

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

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 paper we are mainly interested in CLP with syntactic unification. We believe however that our approach can be adapted to the standard semantics of CLP.

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. 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 2.1 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 $\mathcal{D} \models c' \to c$. The relation of being an instance is transitive. (Take an instance $c'[]E\theta$ of $c[]E$ and an instance $c''[]E\theta\sigma$ of $c'[]E\theta$. As $\mathcal{D} \models c'' \to c'\sigma$ and $\mathcal{D} \models c' \to c\theta$, we have $\mathcal{D} \models c'' \to c\theta\sigma$).

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 2.2 $a + 7$, $Z + 7$, $1{+}7$ are instances of $X + Y$, but 8 is not.

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

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

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

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

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. 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 [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

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

(where $Vars(E)$ is the set of free variables occurring in E, and $\exists_{-V}$ stands for quantification over the variables not in V). $\square$

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

$T_P^{\mathcal{C}}$ is continuous w.r.t. $\subseteq$. So it has the least fixpoint $T_P^{\mathcal{C}} \uparrow \omega = \bigcup_{i=0}^{\infty}(T_P^{\mathcal{C}})^i(\emptyset)$. 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 2.4 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 $x{\in}\{1,2\}[]p(x)$. (It also contains variants of the latter constrained atom, obtained by renaming variable x). $M(P_2)$ contains additionally all the instances of $x{\in}\{1,2\}[]p(x)$, like $y{=}1[]p(y)$.

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 paper.

2.4 Call-success semantics

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 and H, constraint c'' is equivalent to $(c \wedge c')\theta$ and is satisfiable and $\mathit{Vars}(G_i) \cap \mathit{Vars}(C_{i+1}) = \emptyset$.

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 $\exists_{-\mathit{Vars}(A_1)} c[]A_1$). 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 2.5 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 2.6 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.

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').

3 Types

We are interested in computing approximations of the call-success semantics of programs. A program's semantics is an instance closed set of constrained atoms, an approximation is its superset. The approximations are to be manipulated 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[11]. 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 list with elements of any type t.

Section 3.1 describes the class of types we use 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 used in Section 3.3, where we present algorithms for operations on types. The operations are needed in 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 to the case of constrained terms.

[11]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.

The next definition combines a success set of a given unary predicate and a corresponding set of constrained terms.

Definition 3.1 Let P be a CLP program. Let p be a unary predicate. Then $[\![p]\!]_P := \{\, c[]u \mid c[]p(u) \in M(P)\,\}$. □

A type will be determined by a unary predicate in a (restricted kind of a) constraint logic program. All the predicates of a program R (which defines types) are unary; a predicate symbol t is considered to be a *name* of a type and $[\![t]\!]_R$ is the corresponding type.

Definition 3.2 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 $\mathit{Vars}(F) \subseteq \{x_1, \ldots, x_n\}$. $F[u_1, \ldots, u_n]$ will denote F with each x_i replaced by the term u_i.

Definition 3.3 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 3.4 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:

$$t_0(f(x_1, \ldots, x_n)) \leftarrow t_1(x_1), \ldots, t_n(x_n). \tag{1}$$

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

$$t_0(x) \leftarrow c[x]. \tag{2}$$

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

□

Example 3.5 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 3.6 A type of lists with (possibly nonground) 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
$M(R) = cl\left(\{\, c[x_1],\dots,c[x_n]\,[]\, list([x_1,\dots,x_n]) \mid n \geq 0 \,\} \cup \{\, c[x][]elem(x) \,\}\right)$.

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.[12] Thus our approach makes it possible to express prescriptive types like those of programming language Goedel [HL94].

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

Example 3.7 The type of all ground terms (over the given signature) is defined by predicate *ground* and a (RUL) program containing the clause $ground(f(x_1,\dots,x_n)) \leftarrow ground(x_1),\dots,ground(x_n)$ 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 the RUL program defining *ground* is the corresponding program for constraint *true*.

3.2 Regular constraints

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

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

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

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 $\mathcal{D} \models c \rightarrow c'[x]$. 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 $\mathcal{D} \models c \rightarrow c'[f(u_1,\ldots,u_n)]$. The resulting algorithm is also linear time provided that the check for $\mathcal{D} \models c \rightarrow c'$ 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 ground types corresponding to the possibly non ground types defined by R.

Definition 3.8 Let R be a RULC program. Let $R_c = \{ t_i(x) \leftarrow c_i[x] \mid i = 1, \ldots, n \}$ be those clauses of R that contain constraints. Let (for $i = 1, \ldots, n$) R_i be a corresponding program for c_i and t_i, such that any predicate symbol distinct from $t_1, \ldots, t_n$ occurs in at most one of the programs $R, R_1, \ldots, R_n$ (and that t_i does not occur in R_j, for $i \neq j$). Then

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

□

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 3.9 [Inclusion] Let R_1, R_2 be RULC programs (not necessarily distinct). Relation $\sqsubseteq$ is defined as the greatest[13] 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 $t_1(f(x_1, \ldots, x_n)) \leftarrow r_1(x_1), \ldots, r_n(x_n) \in R_1$, $n \geq 0$,
   (a) there is a clause $t_2(f(x_1, \ldots, x_n)) \leftarrow s_1(x_1), \ldots, s_n(x_n) \in R_2$ 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 $t_2(f(x_1, \ldots, x_n)) \leftarrow s_1(x_1), \ldots, s_n(x_n) \in R_c$, 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 3.10 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 $[\![t]\!]_R \neq \emptyset$ then $(t, R) \sqsubseteq (t', R')$.

[13] $\sqsubseteq$ is 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.

Definition 3.11 [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 3.12 For R_1, R_2, R, t_1, t_2 and $t_1 \sqcap t_2$ as in the last definition,
$$[\![t_1 \sqcap t_2]\!]_R = [\![t_1]\!]_{R_1} \cap [\![t_2]\!]_{R_2}$$

Definition 3.13 [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 3.4.

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
$$\begin{array}{l} t_1(f(x_1, \ldots, x_n)) \leftarrow q_1(x_1), \ldots, q_n(x_n) \in R_1 \\ t_2(f(x_1, \ldots, x_n)) \leftarrow r_1(x_1), \ldots, r_n(x_n) \in R_2 \end{array}$$
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 $(t_1 \sqcup t_2, R'') = (t_1, \mathit{ground}(R_1'')) \sqcup (t_2, \mathit{ground}(R_2''))$ 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 3.14 If $(t_1 \sqcup t_2, R) = (t_1, R_1) \sqcup (t_2, R_2)$ then
$$[\![t_1]\!]_{R_1} \cup [\![t_2]\!]_{R_2} \subseteq [\![t_1 \sqcup t_2]\!]_R$$

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 **C** is used by the c-semantics of programs. So **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 **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 3.15 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 $I \subseteq [\![approx]\!]_R$. R is a *regular approximation* of P if it is a regular approximation of $M(P)$. □

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

Example 3.16 Let P be the following CLP(R) program

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

Then the following program is a regular approximation of P.

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

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 3.17 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 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 $R' \ncong R$. Now, as $M(P) \subseteq \gamma(R')$, we have $R = \alpha(M(P)) \preceq \alpha\gamma(R') = R'$, contradiction.

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. $T_P^{\mathcal{A}}$.[14] 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)

The concept of *finite domains* was introduced to logic programming by [Hen89]. We will basically follow this framework, including the terminology. So within this section "domain" stands for a finite domain in the sense of [Hen89]. We assume that a domain is a finite set of natural numbers (including 0). This is the case in most of CLP(FD) languages. To any domain S there corresponds a *domain constraint* $x \in S$, with the expected meaning. Usually a variable involved in such a constraint is called a domain variable.

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 arithmetical expressions and its subset of so called linear terms.

Defining the first three of them by a RULC program would require an infinite set of clauses. So we extend RULC programs by three "built-in" types[15]. We introduce unary predicate symbols *nat*, *neg* and *anyfd*, which cannot occur in the left hand side of a RULC clause. We assume that (independently from a RULC program) $[\![nat]\!]$ is the set of all non-negative integer constants, $[\![neg]\!]$ is the set of all negative integer constants and $[\![anyfd]\!]$ is $cl(\{\, x{\in}S[]x \mid S \subseteq \mathbb{N},\ S \text{ is finite} \,\})$.[16] We allow clauses of the form $t(x) \leftarrow builtin(x)$ to occur in RULC programs (where *builtin* is one of the three symbols). By an instance of the head of such clause we mean any element of $[\![builtin]\!]$.

The type *int* of integers and the type of arithmetical expressions are defined by means of these special 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.

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

[15] Alternatively we can assume that the type of integers is finite. A similar solution is taken in constructing a semantics for CLP with interval constraints [BO97].

[16] If all the finite domains are the subset of some maximal domain $0..max$, then this type may be defined by a RULC clause $anyfd(x) \leftarrow x \in 0..max$.

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. We believe that a formalism of regular term grammars with constraints is more suitable for this purpose. It additionally provides some form of parametric types. If the parameters are not used, the formalism is equivalent to RULC programs. 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 3.19 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 subsection.

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 3.18 We consider *rules* of the form

$$\begin{array}{l} t(\alpha_1, \ldots, \alpha_n) \to f(t_1, \ldots, t_k) \\ t(\alpha_1, \ldots, \alpha_n) \to c[x] \end{array}$$

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 paper.

Example 3.19 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.

4 Type inference

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

A constraint occurring in a program clause is a conjunction of atomic constraints. We use a constraint solver to find a kind of Cartesian projection of such constraint. (Another constraint solvers may also be used).

Example 4.1 Consider CLP(FD) and a constraint $c = 0{<}x,\ x{<}y,\ y{<}7$. A CLP(FD) solver finds that $c \to x{\in}\{1..5\}, y{\in}\{2..6\}$. Types t, s defined by a RULC program $\{\, t(x) \leftarrow x{\in}\{1..5\};\ s(y) \leftarrow y{\in}\{2..6\} \,\}$ will be used to approximate c, by the conjunction $t(x), s(y)$.

Let $c[x_1, \ldots, x_n]$ be a constraint (with the free variables $\{x_1, \ldots, x_n\}$). If $c_1[x_1], \ldots, c_n[x_n]$ are regular constraints such that $\mathcal{D} \models c[x_1, \ldots, x_n] \to c_1[x_1], \ldots, c_n[x_n]$ then $t_1(x_1), \ldots, t_n(x_n)$ is called a **regular approximation** of c in a RULC program $\{\, t_i(x_i) \leftarrow c_i[x_i] \mid i = 1, \ldots, n \,\}$.

4.2 Approximating a clause

Now we give an algorithm that computes a function *solve*. Its arguments are a CLP clause and a RULC program and its value is a RULC program. It approximates the semantics $T_{\{C\}}^{\mathcal{C}}$ of a clause C:

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

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

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

is maintained.

1. Compute a regular approximation $t_1(x_1), \ldots, t_m(x_m)$ of c_0, in a RULC program R' (such that R and R' are predicate-disjoint). Set D as

$$h \leftarrow t_1(x_1), \ldots, t_m(x_m), approx(b_1), \ldots, approx(b_n)$$

and Q as $R \cup R'$.

Since $t_1(x_1), \ldots, t_m(x_m)$ approximates c_0 and since $c[]b \in \gamma(R)$ iff $c[]approx(b) \in M(R)$, this step established the invariant.

2. 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.

3. 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 3.12), hence the invariant is obviously preserved.

4. For each variable z in D, that occurs in the head but not in the body, add the atom $any(z)$ to the body (and clause $any(x) \leftarrow.$ to Q, if it is not there).

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

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

The function *norm* is used to obtain a fixpoint in a finite number of steps and will be defined below. 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 $R \preceq T^{\mathcal{A}}_P(R)$.

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

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

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

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.

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][17].

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

Definition 4.3 [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 [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 4.5 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:

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

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

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 4.6 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 $1 \leq i \leq n$. 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. The type t at a position π in Q_n depends on the 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 4.7 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

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

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.

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,
    constraint_queens(List),
    labeling(List).

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

constraint_queens([]).
constraint_queens([X|Y]):-
    safe(X,Y,1),
    constraint_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 the top goal and its call patterns 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    : nqueens(nat,any)
success : nqueens(nat,natlist)
-------------------------
call    : labeling(fdlist)
success : labeling(natlist)
-------------------------
call    : constraint_queens(fdlist)
success : constraint_queens(fdlist)
-------------------------
call    : safe(anyfd,fdlist,int)
success : safe(anyfd,fdlist,int(Z)
-------------------------
call    : noattack(anyfd,anyfd,int)
success : noattack(anyfd,anyfd,int)
```

Types `natlist` and `fdlist` denote lists of `nat` and `anyfd` respectively. Assume now that the second clause defining `safe/3` contains a bug:

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

Types inferred by the analyzer look like follows (we show only those which differ from ones generated previously):

```
success : nqueens(nat,t102)
t102 --> [nat|t78]
t102 --> []
t78 --> []
-------------------------
call    : labeling(t90)
t90 --> []
t90 --> [anyfd|t78]
```

```
success : labeling(t102)
-------------------------
success : constraint_queens(t90)
-------------------------
call    : safe(anyfd,t71,int)
t71 --> []
t71 --> [anyfd|fdlist]
t71 --> t

success : safe(anyfd,t78,int).
```

The types inferred are obviously suspicious and should be helpful in localizing the bug in the program. For instance, the second argument of success of `nqueens/2` (type `t102`) is an empty list or a one-element list of naturals. A similar problem is with `constraint_queens`. The problem may be traced down to `safe/3` which succeeds with the empty list as the second argument.

6 Conclusions and future work

In this paper we propose a method of computing semantic approximations for CLP programs. Our aim is a practical tool that would be helpful in debugging. We are mainly interested in CLP(FD), particularly in the language CHIP. Our approach considers 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 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 responsible for the built-ins of CHIP. The prototype has been implemented in CHIP and has been ported to SICStus Prolog and CIAO [BCC+97]. 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)

A subject for future work is obtaining more precise analysis by using a more sophisticated treatment of constraints and implementing a richer class of types (in the present implementation a restricted class of constraints is allowed in RULC programs).

Another direction of further work is relating our technique to abstract debugging [CLV94]. A clear relationship between these two techniques should be established. The first step is presented in [CDP99, CDMP98]. We also plan to develop a tool combining the two approaches.

ACKNOWLEDGMENT

The authors want to thank Jan Małuszyński for discussions and suggestions.

Paper II

Locating type errors in untyped CLP programs

Włodzimierz Drabent, Jan Małuszyński and Paweł Pietrzak[18]

1 Overview

This chapter presents a *static diagnosis* tool that locates type errors in CLP programs without executing them. The existing prototype is specialized for the programming language CHIP [Cos98], but the idea applies to any CLP language.

The tool works with approximated specifications which describe types of procedure calls and successes. The specifications are expressed as a certain kind of term grammars. The tool automatically locates at compile time all the errors (with respect to a given specification) in a program. The located erroneous program fragments are (prefixes of) clauses. The tool aids the user in constructing specifications incrementally; often a fragment of the specification is already sufficient to locate an error.

The presentation is informal. The focus is on the motivation of this work and on the functionality of the tool. Some related formal aspects are discussed in [DP98, Pie98].

2 Introduction

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

A rather ad hoc approach to locating an error is tracing the execution which shows a symptom. In the case of declarative languages 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]). Declarative diagnosis of CLP programs is studied in [TF00].

[18] The authors acknowledge the contribution of Marco Comini who was largely involved in the development of an early version of the diagnosis tool [CDMP98, CDP99].

In contrast to the above mentioned approaches, we propose to locate errors in a CLP program without searching for symptoms, that is without executing the program. The idea can be linked to methods for proving partial correctness of a program with respect to a specification, such as [Cla79, Der93, DM88, BC89]. Our tool tries to construct a proof that the program is correct w.r.t. the specification. If the proof is obtained then every execution will be free of symptoms violating the specification. Conversely, if a symptom violating the specification can be observed, a proof does not exist. In an incorrect program our tool finds all the fragments that are responsible for non existence of the proof. To use the tool the user need not be familiar with the underlying program verification techniques.

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

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

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

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

Figure 2 presents a general overview of the tool. The two main components are the *type analyser* and the *diagnoser*.

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

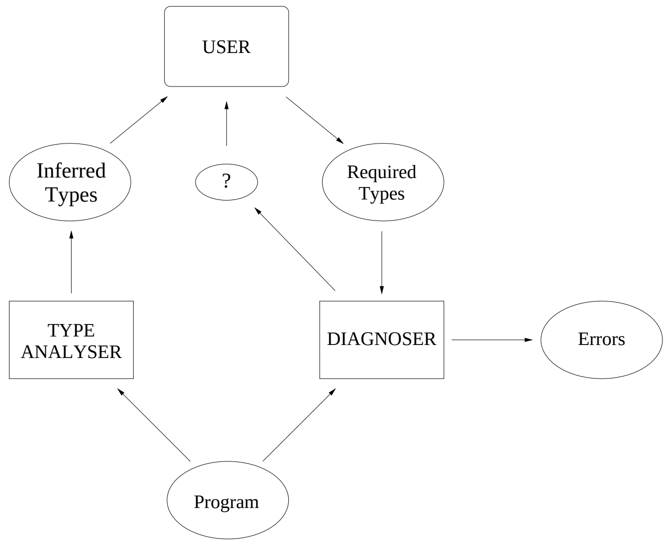

Figure 2: Overview of the tool

The specification requests are represented by the question mark in Fig. 2. An error locating message is generated by the diagnoser as soon as a sufficient subset of the required types is specified. Providing further specification may result in locating more errors.

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

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

The rest of this paper is organized as follows. Section 3 discusses the concept of partial correctness and the language of types as a basis for formulation of the diagnosis problem. The use of the tool and its functionality is illustrated on an example in Section 4. Section 5 explains informally the underlying principles of static diagnosis, Section 6 describes the treatment of delays in our approach and Section 7 discusses some implementation issues. Section 8 enumerates some limitations of our approach. Related work is surveyed in Section 9. Section 10 presents conclusions and future work.

3 The Specification Language

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

Notice that the form of procedure calls and successes is among the properties that can be described by assertions discussed in [PBH00a]. The properties we deal with can be considered as a special case of `calls` and `success` assertions, tagged with `true` or with `check` depending on whether the properties are obtained from program analysis or are a part of a specification. However [PBH00a] does not consider actual ways of describing the sets of constrained atoms, to which assertions refer. What we do here is introducing a particular formalism to specify a suitable for our purposes class of sets of constrained atoms, namely a formalism of term grammars to specify regular types of constrained terms/atoms.

Before introducing our specification language in Section 3.2, we explain how the behaviour of a CLP program is characterized in terms of predicate calls and successes. This includes discussing some basics of the chosen CLP semantics and presenting a formal definition of a procedure call and success. Some readers may prefer to skip the latter.

We assume a fixed CLP language (object language) over a fixed constraint domain $\mathcal{D}$. In the next subsection $\mathcal{D}$ is arbitrary, the rest of the paper deals with CLP over finite domains. To simplify the presentation, we first present our approach applied to programs without delays (i.e. executed under the Prolog selection rule)[19]. Treatment of delays is discussed in Section 6.

3.1 Calls and Successes of a CLP Program

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

The errors we want to locate demonstrate themselves as wrong computed answers or as wrong arguments of predicate calls, where "wrong" refers to

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

user expectations or to a priori given requirements concerning the use of built-ins. These aspects of computations can be captured by associating with each predicate two sets: one of them describing all calls and the other all successes in the considered class of computations. This section discusses this idea in more detail. For a more formal presentation see [Pie98].

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

Computations of a CLP program involve constraints. Therefore, predicate calls and successes take the form of *constrained atoms*. A constrained expression (atom, term, ...) is a pair of the form $c \,[]\, E$ where c is a constraint and E is an expression such that each free variable of c occurs in E. For example, $X :: 1..4 \,[]\, p(X,Y)$ is a constraint atom in CHIP notation[20]. For a c not satisfying the latter condition, $c \,[]\, E$ will be an abbreviation for $(\exists_{\ldots} c) \,[]\, E$ where the quantification is over all variables not occurring in E. A constrained expression $true \,[]\, t$ may be represented as t.

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

An *LD-derivation* is a sequence $G_0, C_1, \theta_1, G_1, \ldots$ of goals, input clauses and mgu's (similarly to [Llo87b]). A goal is of the form $c \,[]\, A_1, \ldots, A_n$, where c is a constraint and $A_1, \ldots, A_n$ are atomic formulae (including atomic constraints). For a goal $G_{i-1} = c \,[]\, A_1, \ldots, A_n$, where A_1 is not a constraint, and a clause $C_i = H \leftarrow B_1, \ldots, B_m$ the next goal in the derivation is $G_i = (c \,[]\, B_1, \ldots, B_m, A_2, \ldots, A_n)\theta_i$ provided that θ_i is an mgu of A_1 and H (and that G_{i-1} and C_i do not have common variables). If A_1 is a constraint then $G_i = c, A_1 \,[]\, A_2, \ldots, A_n$ ($\theta_i = \epsilon$ and C_i is empty) provided that c, A_1 is satisfiable.

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

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

$X :: [1,2,4] \,[]\, p(X,7)$ is a possible pair of a call and a success for p defined by $p(X,7) \leftarrow X \neq 3$.

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

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

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

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

3.2 Describing Sets of Constrained Atoms

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

For the purposes of program analysis and diagnosis, we need to compute certain operations on sets: set intersection and union (possibly approximated), tests for membership, inclusion and emptiness, and operations of construction and deconstruction which are explained in Section 5.2. The expressive power of the formalism is limited to facilitate effective and efficient computation of these operations. Due to this limitation, the call-success semantics of a CLP program is usually not expressible in the specification language and the specifications provided for programs describe approximations of the semantics. The approximations will be called *types* following the terminology used in the descriptive approach to types in logic programming.

Our formalism is a generalization and adaptation to CLP of the ideas of [DZ92]. In particular, we add a possibility to distinguish ground types from those containing also non-ground terms.

Types will be denoted by *type terms* constructed from *type constants* and *type constructors*.

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

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

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

$$\begin{array}{l} p \to 0 \\ p \to s(p) \end{array}$$

mean that the type constant p denotes the set of terms $\{0, s(0), s(s(0)), \dots\}$. The meaning of p is formally defined as the set of all terms not containing type symbols that can be derived from p by applying the grammatical rules. For a precise definition the reader is referred to [DMP00b].

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

$$\begin{array}{l} int \to neg \\ int \to nat \end{array}$$

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

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

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

More precisely, a grammatical rule can be defined as an expression of the form:

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

where t is an n-ary type constructor ($n \geq 0$), $\alpha_1, \ldots, \alpha_n$ are distinct type variables, f is a standard type constant ($k = 0$) or f is a function symbol of the object language ($k \geq 0$), and each τ_i ($1 \leq i \leq k$) is:

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

In addition, it is required that no function symbol is a principal symbol of two distinct grammar rules. Here by a principal symbol of a rule we mean such f that $\ldots \,[]\, f(\ldots)$ can be generated from the right hand side of the rule.

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

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

4 An Example Diagnosis Session

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In CHIP, the built-in predicate `is/2` is subject to delay until its second argument is ground. In our example the analyser finds that `K` is an integer at the call of `K1 is K+1`, hence this call is not delayed. The treatment of delays in our system is discussed in Section 6.

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

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

The call type comes from the entry declaration and the computed success type is described by a term grammar. The intended result should be a placement of n queens on the $n \times n$ chess board, represented by a list of natural numbers. So the expected success type of the second argument is $list(nat)$ (as defined by the grammar rules in Section 3.2 with α set to nat) and not the type constructed, which denotes a singleton list of natural numbers. We request diagnosis of the inspected predicate. In response, the diagnoser finds all predicates which may influence the types of this predicate, and asks the user about their intended call- and success-types.

In our example, the diagnoser will request specification of the following types, where `C` stands for "call-type" and `S` for "success-type": `(C)constrain_queens/1`, `(C)safe/3`, `(S)safe/3`, `(S)constrain_queens/1`, `(S)noattack/3`, `(C)noattack/3`, `(S)nqueens/2`.

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

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

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

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

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

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

violates the specification (see also Fig. 4).

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

Figures 3 and 4 show the graphical use interface of the diagnoser. It has three information windows and a display. After completing the analysis phase the leftmost window shows all predicates of the program. The computed call- and success- types of a predicate can be displayed by clicking a

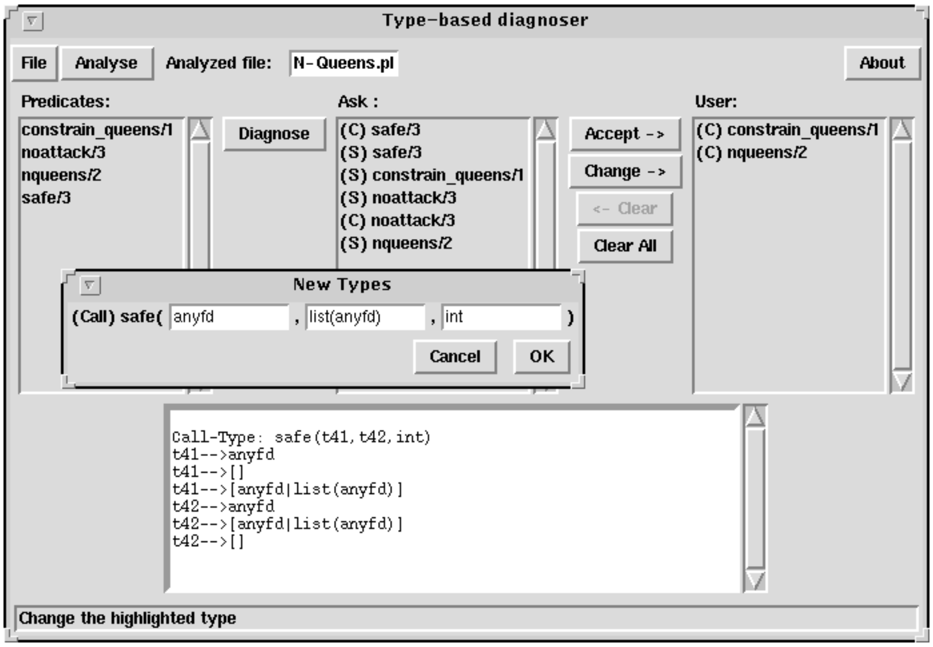

Figure 3: Specifying a new call type for `safe/3`.

predicate in this window. If they do not conform to the user's expectation, diagnosis may be started by pressing the button "Diagnose". In response, the diagnoser generates the list of all types which may be needed to be specified for completing the diagnosis and displays them in the "Ask" window. At each step of the diagnosis session this window shows which types remain to be specified. The "User" window shows which types have been already specified during the session. A type selected in the "Ask" window is one that has to be specified. The corresponding type inferred by the analyser is shown in the display at the bottom. The user may accept it as a part of the specification, by pressing the "Accept" button. Otherwise, clicking the "Change" button will trigger a pop-up window for writing a new specification for the active entry of the "Ask" window, as illustrated in Figure 3. The already specified types shown in the "User" window may be inspected and withdrawn (by pressing the button "Clear" or "Clear All"), if the user decides to specify them differently. Figure 4 shows the warning generated by the diagnoser.

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

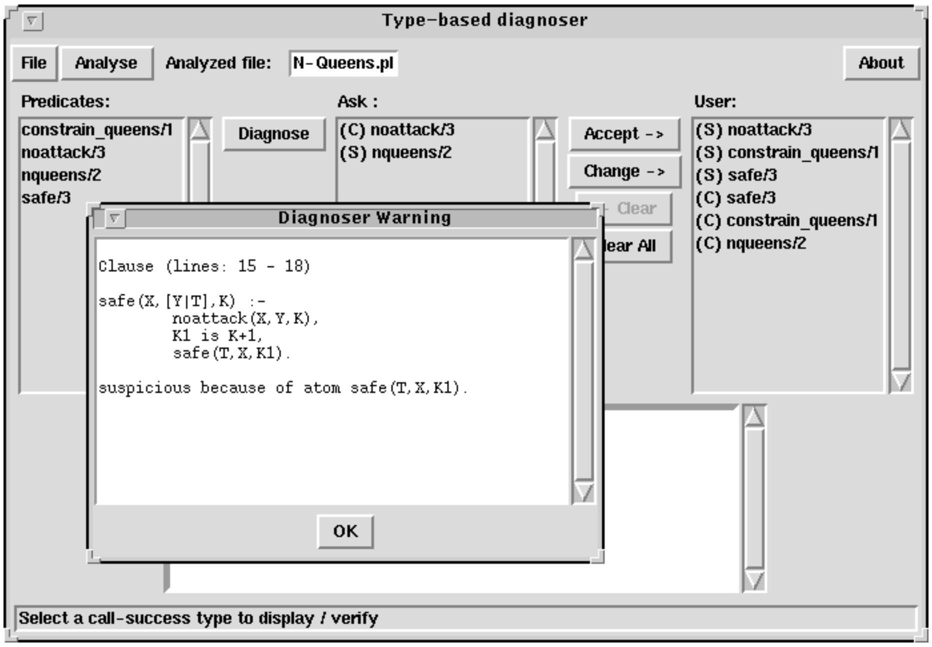

Figure 4: Warning given by the diagnoser.

been retrieved from a library, and the call-type of the main predicate).

Observe that the diagnosis engine is a compile time tool. It does not require performing any test computations of the diagnosed program. In contrast to the usual debugging techniques, like tracing or even declarative debugging, it is not driven by error symptoms appearing in test computations. The diagnoser has access only to the diagnosed program and to its specification constructed interactively by the user with the help of the results of static analysis. For finding the error it uses partial correctness proof techniques. As the example shows, an error may be sometimes located by providing only part of the specification. The role of the program analyser is auxiliary and in principle diagnosis can be done without it. However, in our tool this option has not been implemented and the analyser is invoked at the beginning of each session. Knowing a difference between a type found by the analyser and that expected by the user, makes it possible to focus the diagnosis to a fragment of the program. The part of the analyser's output that is found by the user to be correct, is re-used as a (part of) the specification.

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

5 The Diagnosis Method

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

5.1 Correct and Incorrect Clauses

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

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

Example 5.1 We show an example of a clause that leads to an incorrect call even if the rest of the program behaves correctly. Consider the clause located by our diagnoser in the example session of Section 4.

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

Assume that the rest of the program behaves correctly: each success of a predicate is in its specified success-type. With this assumption we examine the clause. The specification is that given to the system in the example diagnosis session of Section 4. Additionally, the call type of `noattack/3` is the same as its success type.

1. Assume that the clause is resolved with a call that belongs to the specified call-type `safe(anyfd,list(anyfd),int)`. Hence, the subsequent call to `noattack/3` must belong to `noattack(anyfd,anyfd,int)`; this follows from the structure of arguments in the head of the clause and from the definition of the type `list(anyfd)`. This conforms to the specified call-type of `noattack/3`, the obtained type is a subset of the specified call-type of `noattack/3`.

2. Assume that the call to `noattack/3` succeeds and the success of this call is in the specified success-type `noattack(anyfd,anyfd,int)`. Hence the built-in `is/2` is called according to its specification (which requires that the second argument is a ground arithmetic expression).

3. Consequently the success of `is/2` is in its specified success type (`K1` is bound to an integer). Hence the subsequent call to `safe/3` is in the type `safe(list(anyfd),anyfd,int)`. This is in conflict with the specified call-type of `safe`, which is `safe(anyfd,list(anyfd),int)`. The former is not a subset of the later.

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

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

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

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

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

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

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

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

Example 5.2 Here we show a correct program containing an incorrect clause. The reason is that the specification is too weak. Moreover, a sufficiently strong specification does not exist (in the considered class of specifications).

Consider a clause

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

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

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

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

It can be proved that for any specification formalism such examples exists. (The set of procedure calls or successes of a given procedure in a program may be undecidable. On the other hand, a specification formalism usable in a context like ours defines only decidable sets).

Example 5.3 Here we show that a problem of an incorrect clause occurring in a correct program can be caused by different usages of a procedure in the program.

Consider the example from Section 4 and a (corrected) clause

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

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

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

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

5.2 Incorrectness Diagnosis

Incorrectness diagnosis is performed by automatic identification of clauses that are incorrect w.r.t. to a given specification. The core of the diagnoser is thus an algorithm that automatically checks correctness of prefixes of a clause. The basic idea is to compute types that are (subsets of) the sets of all possible calls of the body atoms (and of all successes of the head). The assumptions about the calls of the head and the successes of the body atoms are the same as in the previous section (for instance in Ex. 5.1). The correctness check may be described in terms of some primitive operations. We describe them referring to the example:

- *Deconstruction.* This operation determines the type of a subterm of a given typed term. Consider the clause of Example 5.1. The call type of its head predicate is `safe(anyfd,list(anyfd),int)`. The head is `safe(X,[Y|T],K)`. Using the decomposition operation we obtain types of the terms bound to the variables in the head in any correct call. For `Y` and `T` it yields types `anyfd` and `list(anyfd)`, as the type of `[Y|T]` is `list(anyfd)`. (The types of `X` and `K` are trivially `anyfd` and `int`). The reader may check that decomposition is easy to compute using the rules defining types.

- *Type intersection.* Suppose that, by means of deconstruction, types are found for some occurrences of a variable. Then its possible values are determined by the intersection of the types. Assume that the example clause has been called as specified and that the execution reached the call of `is/2`. The variable `K` at this point is bound to a constrained term that is in the intersection of the call-type of the third argument `safe/3` and the success type of the third argument of `noattack/3`. Both types are `int`, so the intersection is trivial in this case. The algorithm used in our tool for computing intersection of types is described in [DP98, Pie98].

- *Construction.* This operation determines the type of a term from types of its subterms. New grammar rules have to be constructed to describe it. In our example, the type, say `t10`, of `K+1` can be computed knowing the type of `K` (`nat`) and `1`:

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

- *Type inclusion.* To establish (in)correctness of a clause we have to check whether the clause leads to (in)correct calls or successes. For example, whenever the rest of the program behaves as specified, the first argument of the last body atom will be bound at call to a term in `list(anyfd)`. This has been established by decomposition of the initial call. The specified call-type for this argument is `anyfd`. Incorrectness is established by checking that the former type is not a subset of the latter. The algorithm used in our tool for checking type inclusion is described in [DP98, Pie98]

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

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

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

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

If the specification at hand describes all predicates of the program the above algorithms can be used to find all incorrect clauses. If some type specifications are missing, we can use the following technique. We describe it for the case where $B = p(...)$ is a body atom (and under assumption that the specification of the call type of B is not missing). First replace all the missing types by the most general type *any*. If the algorithm computes the type of the call of B, which is a subset of the specified call type of p, then the clause is correct (for any possible specification of the missing types). Conversely, suppose that after replacing the missing types by the empty type the computed call-type of B is not a subset of the specified one. Then the clause is incorrect.

6 Delays

Modern Prolog and CLP implementations use Prolog selection rule with delays. This means that, for some predicates, an atom can be selected only if its arguments are sufficiently instantiated. The requirements on the form of selectable goals are given by delay declarations. In this section we explain handling of delays by our system. The presentation is informal. The predicates subject to delays will be called *delayed* predicates. For simplicity, we deal only with built in delayed predicates.

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

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

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

As discussed in Section 3.1, we do not treat constraint predicates as delayed. Whenever an atomic constraint is initially selected, it is added to the constraint store. Thus it immediately succeeds or fails depending on the result of the constraint consistency check.

For each delayed predicate, instead of its call-type, we maintain its selection-type describing the set of selectable calls. If an initial call is in this set then it is immediately selected (i.e. not delayed)[21]. For instance, the selection-type for the built-in predicate `is/2` of CHIP is the set of constrained atoms $c \, [] (u \, \mathtt{is} \, w)$ where w is a ground arithmetic expression (and c, u are arbitrary).

Now we present an algorithm for checking correctness of a clause in the context of derivations with delays. The algorithm is a modification of that shown in Section 5.2. Given a clause $H \leftarrow B_1, \ldots, B_n$, the algorithm approximates the set of initial calls of each B_i. If the result is a subset of the selection-type of B_i then we are sure that B_i is not blocked (even if its predicate is a delayed one).

[21]So if we want to approximate the set of the selectable calls, the selection-type has to be a subset of this set, not a superset.

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

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

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

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

The same property holds if the only warnings issued by the algorithm are related to the last check in the algorithm (B_i never selected). Such a warning may correspond to a run-time error. As an example take B_i to be ...`is`...

and suppose that no atom in *type*(B_i) has an arithmetical expression as the second argument, so that the intersection considered in the check is empty. Then execution of the program by CHIP will stop with a run-time error (unless B_i is never initially selected).

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

Example 6.1 In the scalar product program below, each call of is/2 is blocked, as the second argument is not ground.

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

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

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

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

An advantage of the approach is its simplicity and ease of augmenting the system to handle delays. Its generalizations are subject of future work. We expect that a major improvement can be achieved by discovering the cases when a blocked body atom is selected before "the clause succeeds". It may be interesting to consider specifying both call- and selection-types for delayed predicates.

7 The Diagnosis Tool

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

Both the analysis and diagnosis algorithms work with types represented as parameterless regular term grammars. Equivalently, such grammars can be seen as a restricted class of CLP programs [DP98]. Parametric grammars are employed only in the user interface. There is a library of type definitions which may be augmented by the user. It contains for instance a parametric grammar defining type $list(\alpha)$ (Cf. Section 3.2). Whenever possible, the types computed by the system are presented to the user in terms of those defined in the library or declared by the user. In this way the user faces familiar and meaningful type names instead of artificial ones. For instance, assume that the system has to display a type $t77$ together with grammar rules $t77 \to [\,]$, $t77 \to [t78|t77]$. It finds that they are an instance of the rules defining $list(\alpha)$ and displays $list(t78)$ instead. Similarly, the user's input can refer to the types defined in the library.

The principles of the analysis are described in a separate paper [DP99]. The analyser and the diagnoser share some basic components. The algorithm is implemented in SICStus Prolog, the implementation is based on that by [GdW94]. We made several lower level improvements to the original implementation, like introducing more efficient data structures (AVL trees instead of lists). They resulted in substantial improvement in analysis efficiency. Running the analyser on a number of examples, we have observed 20-25% speedup.

The type analysis algorithm constructs call and success types of the predicates defined by program clauses, thus computing an approximation of their call-success semantics. To be able to deal with real programs, it uses a library of type specifications of built-in predicates. Similarly it is able to deal with fragments of programs (for instance with programs under development). In the latter case the user is required to provide type descriptions for the undefined predicates. Such descriptions correspond to `trust` assertions of [PBH00a].

The types constructed by the analyser are on request shown to the user, who may decide to start diagnosis, as illustrated in Section 4. The diagnosis relies on the type specification provided incrementally by the user. As discussed in Section 4, the specification process is supported by the possibility to accept some types constructed in the analysis phase as specified ones. It is also restricted to the predicates relevant for the diagnosed predicate. Moreover, the list of types is sorted, so that types that are needed to verify many clauses are on the top, and will be answered first. This strategy often results in finding an error after having specified just a few types. The user may stop the diagnosis with the first error message, which is often obtained without specifying all requested types. The diagnosis process may be continued by specifying all requested types. In this case, the tool will locate all incorrect clause prefixes in the fragment of the program relevant for the diagnosed predicate.

An error message contains an incorrect clause. Its incorrect prefix is indicated by referring to the atom whose type computed by the method of Section 5 is not a subset of the respective specified type. This atom is the head of C or the last atom of an incorrect prefix.

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

8 Limitations of the Approach

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

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

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

Example 8.1 Let us consider the following erroneous *append* program, where we have `[X|Ys]` instead of `[X|Zs]`.

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

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

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

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

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

Example 8.2 Consider an erroneous version of a program for "closing" open (or partial) lists.

```
:-entry close_list(any).

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

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

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

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

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

Example 8.3 The program below deletes an integer number from a given list of integers, by means of the `app/3` predicate.

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

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

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

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

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

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

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

Analysing various calls of the same predicate separately would obviously bring more precise results and would require employing a *polyvariant* analysis method, as that of [JB92].

Essentially the same limitation affects the treatment of built-ins. Their success types are described taking into account all their possible usages. As pointed out in Section 5 this may make the diagnoser generate undesired warnings (as a clause of a correct program may turn out to be incorrect, due to a too general success type of a built-in).

The call-success semantics may not be suitable, as concerning the use of logical variables. The user may not be interested in the actual calls but rather in the successes related to initial calls. We illustrate this by the example originating from [BM97].

Example 8.4 The following program analyses a binary tree `T` with nodes labeled with natural numbers and constructs a binary tree `NT` of the same shape with all nodes labeled with the maximal label of `T`. The program includes a type declaration defining a parametric type `tree(A)` by two grammar rules (conf. Section 3.2).

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

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

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

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

The call-success analyser infers the following types:

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

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

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

Hence, it correctly shows that during the execution some successes of `maxt` have an argument of the type `tree(any)`, since the constructed trees have nodes labeled by variables. To show that in the final result both arguments of `maxtree` are of the type `tree(int)`, one has to use a richer class of specifications[22] 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.

9 Related Work

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

The question of what the reason of an incorrect behaviour of a program is has been investigated in the framework of declarative diagnosis [Sha82, Llo87a, Fer87]. The concept of incorrectness error originating from that work can be related to the program not being partially correct w.r.t. a specification. Eventually this can be linked to violation of a verification condition in a proof method for partial correctness of run-time properties. Several such methods has been discussed in the literature, e.g. [DM88, BC89, DP99]. Our concept of incorrect clause can be linked to such a verification condition.

The general idea of using semantic approximations for program verification and for locating errors was discussed in our previous work [BDD$^+$97]. That paper was the main inspiration of this work.

The static diagnosis technique presented here is similar to the abstract diagnosis of [CLMV99]. The latter is essentially a method for verifying a program against a specification that describes a program property. It is required that the space of possible properties forms a *Galois insertion* with the semantics of the programming language. Our type domain does not fit into the Galois insertion framework due to lack of an abstraction function [DP98, Pie98]. Unlike our approach, the work of [CLMV99] is not focused on implementing a tool, but rather on theoretical aspects of the diagnosis problem. It aims at finding both incorrectness and incompleteness errors.

The decision to use term grammars as a specification language made it possible to extend for our purposes the well-known results and techniques on regular sets and regular term grammars and the techniques of constructing regular descriptive types for logic programs. There is a vast literature about it (see e.g. survey in [Mil99]). More specifically, our term grammars can be

[22] 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.

directly linked to regular term grammars of [DZ92]. They provide "ad hoc" extension of the latter for dealing with CLP over finite domains. In [Pie98] we present a more systematic way for extending regular term grammars with constraints, which is however not directly used in our tool. There have been many proposals for constructing descriptive regular types for logic programs, e.g. [Mis84, JB92, CP98]. We adopted for extension to CLP the technique of [GdW94]. One of the reasons was that in that case we were able to re-use part of the analyser code for diagnosis.

The paper [BDD$^+$97] was also a starting point for developing two tools that are strongly related to ours, both are described in this volume.

The first one is a framework of assertion-based debugging ([PBH00a, HPB99]). The assertion language used in that framework is a superset of our specification language. It not only allows to express call- and success-types but also reacher properties of calls and successes and several other properties such as determinacy, non-failure, cost, etc. Abstract interpretation is used to infer some properties of the program, including types. (The type inference program has been ported for that purpose from our tool). The user may provide a priori a partial specification by stating some assertions describing her expectations. These are automatically compared with the inferred assertions. The comparison may confirm that the latter imply the former. A failure in verifying this may be due to program error or to incompleteness of the checking method. In the case of such failure the system generates a warning and allows to incorporate run-time checks for non-verified assertions. A special warning is issued when the comparison shows that the inferred assertion and the specified one do not intersect. This implies that the latter will be violated by each computation of the program (or the control will not reach the corresponding program points).

In that approach no attempt is made to locate erroneous clauses, since usually the specified assertions are rather few and may not be sufficient for that purpose. Thus the warnings inform about "abstract symptoms" but do not locate errors. In contrast to our interactive tool the framework is fully automatic.

The assertion tool of Prolog IV, described in [Laï00] uses verification techniques similar to ours for locating erroneous clauses at compile time. However, the assertion language is different from ours. The assertions are built from a fixed number of primitives, mostly some predefined constraints of Prolog IV. This gives less flexibility than our types, but allows direct use of constraint solver for verification of assertions. If an assertion cannot be proved statically then a run-time test is generated, like in the method of [PBH00a, HPB99]. In contrast to our approach, the specification has to be given a priori and the specification process is not supported by static analysis.

Our work was initially described in [CDMP98, CDP99]. Since then, the treatment of delays was added and the presentation of the method has been substantially changed and improved. The latter includes changing the type description formalism. Some changes have been done to the implementation. They include: designing and implementing a graphical user interface and modifications of the analysis algorithm that improved efficiency.

10 Conclusions and Future Work

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

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

As stated in Section 9 relevant techniques adopted concern:

- representing and manipulating regular sets [DZ92],
- construction of regular approximations of logic programs [GdW94],
- methods for proving properties of logic programs [DM88, BC89, DP99]

We had to extend them to handle constrained terms. The efficiency of type construction by our tool was acceptable on the benchmark consisting of the CHIP demonstration programs but we hope to be able to improve it. For instance optimizations similar to those suggested in [Cod99] should be possible, as our type analysis algorithm is equivalent to bottom-up abstract interpretation of the magic transformation of the source program. Developing a different algorithm for construction of types may be another way of attacking this problem. Another planned improvement is optimization of query ordering to minimize the specification effort necessary for locating the first incorrect clause. The topics of future research include: modifying standard types (conf. Section 3.2) in order to extend the tool for other constraint domains, better handling of delays, and diagnosis of incompleteness.

Paper III

Using parametric set constraints for locating errors in CLP programs

Włodzimierz Drabent, Jan Małuszyński
and Paweł Pietrzak

Abstract

This paper introduces a framework of parametric descriptive directional types for constraint logic programming (CLP). It proposes a method for locating type errors in CLP programs and presents a prototype debugging tool. The main technique used is checking correctness of programs w.r.t. type specifications. The approach is based on a generalization of known methods for proving correctness of logic programs to the case of parametric specifications. Set-constraint techniques are used for formulating and checking verification conditions for (parametric) polymorphic type specifications. The specifications are expressed in a parametric extension of the formalism of term grammars. The soundness of the method is proved and the prototype debugging tool supporting the proposed approach is illustrated on examples.

The paper is a substantial extension of the previous work by the same authors concerning monomorphic directional types.

1 Introduction

The objective of this work is to support development of CLP programs by a tool that checks correctness of a (partially developed) program w.r.t. an approximate specification. Failures of such checks are used to locate fragments of the program which are potential program errors.

The specifications we work with extend the traditional concept of *directional type* for logic programs (see e.g. [BLR92]). Such a specification associates with every predicate a pair of sets that characterize, respectively, expected calls and successes of the predicate. Checking correctness of a logic program w.r.t. directional types has been discussed by several authors (see e.g. [AL94, Boy96, BM97, CP98] and references therein). Their proposals can be seen as special cases of general verification methods of [DM88, BC89, Der93]. Technically, directional type checking consists in proving that the sets specified by given directional types of a program satisfy certain *verification conditions* constructed for this program. For directional types expressed as set constraints the verification conditions can also be expressed as set constraints and the check can be performed by set constraint techniques (see e.g. [AL94]).

In this paper we propose an extension of directional types which addresses two issues:

- CLP programs operate on *constraint domains* while (pure) logic programs are restricted to one specific constraint domain which is the Herbrand universe. Directional types of a logic program characterize calls and successes of each predicate as sets of terms. This is not sufficient for CLP where manipulated data include *constraints* over non-Herbrand domains. To account for that we use a notion of *constrained term* where a constraint from a specific domain is attached to a non-ground term. We define the concept of directional type for CLP programs using sets of constrained terms.

- In logic programming, as well as in CLP, some procedures may be associated with families of directional types, rather than with single types. For example, typical list manipulation procedures may be used for lists with elements of any type and return lists with the elements of the same type. This is known as *parametric polymorphism* and can be described by a *parametric specification*, in our case by a *parametric directional type*. We extend the concept of partial correctness of CLP program to the case of parametric specifications and we give a sufficient condition for a program to be correct w.r.t. a parametric specification. We apply this condition to correctness checking of CLP programs w.r.t. to parametric directional types, and for locating program errors. As shown by examples in Section 6, use of parametric specifications improves the possibility of locating errors.

The problem of checking of polymorphic directional types has been recently formulated in a framework of a formal calculus [RT00, RT01]. As explained in Section 7.1 that approach is substantially different from ours.

A parametric specification can be seen as a family of (parameter-free) specifications. As mentioned above, our specifications refer to sets of constrained terms. The sufficient conditions for correctness can be formulated as set constraints, involving operations on the specified sets, such as projection, intersection and inclusion.

For constructing an automatic tool for checking correctness of specifications two questions have to be addressed:

- How to represent sets so that the necessary operations can be effectively performed,

- How to deal with parametric specifications.

The first problem was already discussed in [DMP00a, DMP00b], which extends our earlier work [CDMP98, CDP99]. We have chosen to represent sets of constrained terms by a simple extension of the formalism of discriminative term grammars, where sets of constrained terms are constructed from a finite collection of base sets. Term grammars (or equivalent formalisms) and set constraints have been used by many authors for specifying and inferring types for logic programs (see among others [Mis84, FSVY91, DZ92, GdW94, AL94, Boy96, DJMT97, CP98]). We show how the operations on discriminative term grammars can be extended to handle sets of constrained terms introduced by the extended discriminative term grammars.

A solution to the second problem is a main contribution of this paper. We derive it by showing how the approach of [DMP00a] can be extended to the case of parametric specifications. (In our former work parametric grammars were used only in the user interface, to represent families of grammars.) First we have to give a new, more precise, presentation of that approach. We present a natural extension of the notion of partial correctness to the case of parametric specifications, so that the special case of parameterless specifications reduces to the notion used in our previous work. We introduce a concept of PED grammar (parametric discriminative extended term grammar) as a formalism for specifying families of sets of constrained terms. We define operations on PED grammars that make it possible to approximate results of the respective operations on members of the so defined families. We use them for checking correctness of programs w.r.t. parametric directional types, and for locating potential errors.

If the verification conditions of a logic program are expressed as set constraints, it is possible to infer directional types that satisfy them. For example, the techniques of [HJ90a, HJ91] make it possible to construct a term grammar[23] describing the least model of the set constraints. The use of these techniques for program analysis in general was discussed in [Hei92a].

On the other hand, it is possible to use abstract interpretation techniques to infer directional types of a program. Soundness of an abstract interpretation method can be justified by deriving it systematically from the verification conditions. An example of an abstract interpretation approach is [JB92, HCC95]. A technique of [GdW94], similar to abstract interpretation, derives types in a form equivalent to discriminative term grammars. In [DMP00a] we modified the latter technique to infer directional types for CLP programs. In this paper we present its further extension for inferring parametric directional types. We prove that this extension is sound in the sense that the program is correct w.r.t. the inferred parametric types.

We use our technique of parametric type checking for locating errors in CLP programs. More precisely, we check correctness of a program w.r.t.

[23] In general this grammar is non-discriminative.

a parametric specification of directional types and we indicate fragments of clauses where the check of the verification conditions fails. However, CLP languages are often not typed so that programs do not include type specifications. Therefore our methodology does not require that the type specification is given a priori. The user decides a posteriori whether or not to type check a program, or its fragment.

The type specification is usually provided in a step-wise interactive way. At each stage of this process the program is checked against the fragment of the specification at hand. So incremental building of the specification is coupled together with locating errors. Even small fragments of the specification are often sufficient to locate (some) errors in the program. On the other hand, if no program errors have been located when the specification is completed then the program is correct (w.r.t. the specification). Notice however that not every error message corresponds to the actual error in the program. That is why we call the error messages “warnings”. This is due to using approximated specifications and to approximations made in the process of checking.

In the proposed methodology the process of type specification is preceded by static analysis which infers directional types of the program. The inferred types may provide indication that the program is erroneous. In this case the user may decide to start the process of specification and error location. The results of the type inference may facilitate it, as discussed below and in Section 6. Thus, in our methodology type inference plays only an auxiliary, though useful, role.

The methodology is supported by a prototype error locating tool. The present version of the tool works for a subset of the constraint programming language CHIP [Cos98]. However, it can be easily adapted for other CLP languages.

The structure of the tool is illustrated in Fig. 5. The tool includes a type checker, a type inferencer and a specification editor. The tool has also a library of PED grammars. Among others, the library provides descriptions of often occurring types and specifications for built-in predicates. The specification of a program is introduced through the editor. It may refer to library grammars and/or to grammars provided by the user together with the checked program.

The input consists of a (possibly incomplete) CLP program and of an *entry declaration*. The latter is a parametric specification of intended (atomic) initial calls in terms of some PED grammar. In this way a family of sets is specified. Each member of the family is a different set of intended calls, corresponding to a different use of the program. The type inferencer constructs parametric directional types for all predicates of the program, thus providing a specification such that the input program is correct w.r.t. it. However, these types may not correspond to user intentions. This is due to program errors or to inaccuracy of type inference.

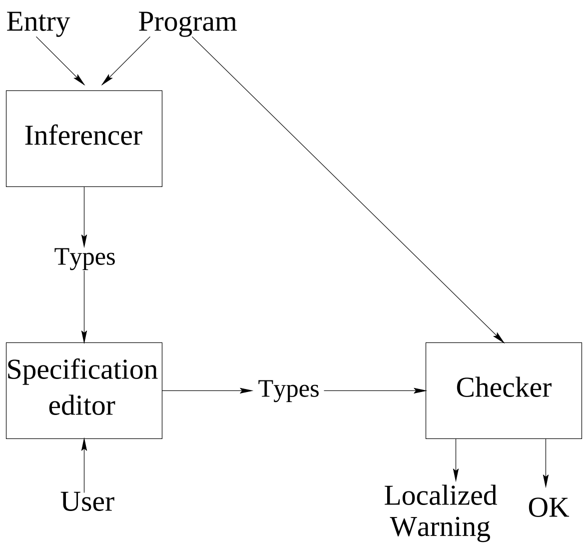

Figure 5: The structure of the error locating tool

The intended types have to be provided by the user. They are introduced in a step-wise interactive manner. When providing the type of a predicate the user may first inspect the inferred type and accept it, or specify instead a different type. The tool monitors the process and immediately reports as an error any violation of the verification conditions for the so far introduced types.

While our approach makes it possible to locate some errors in CLP programs it should be clear that it is limited:

- It locates only type errors.

- Our types are based on discriminative regular grammars; the expressive power of this formalism is limited.

- To deal with constraints we extend this formalism from terms to constrained terms. However our treatment of constraints is rather crude. Roughly speaking, our formalism is able to define only a finite collection of sets of constraints (for any given variable). This limited approach lets us however find typical type bugs related to constraints. In our former work [DP98] we studied a more sophisticated (non parametric) type system for constrained terms. It seems however too complicated. Charatonik [Cha98] showed that a certain approach to approximating the semantics of CLP programs is bound to fail, as the resulting set constraints are undecidable.

- Correctness w.r.t. parametric type specifications requires type correctness for all values of the type parameters. Thus only quite general sufficient conditions for correctness are possible. They however seem to work well on typical examples.

A usual question discussed in the literature is the theoretical worst case complexity of the proposed type checking and type inference algorithms. We show that our type checking algorithm for a clause is exponential w.r.t. the number of variable repetitions. In our approach to locating errors type inference plays an auxiliary role and is implemented by an adaptation of the algorithm of [GdW94] with some ideas of [Mil99]. While we prove soundness of this adaptation, we do not elaborate on the theoretical complexity issues, which by the way were not discussed by the authors of the algorithm. As concerns practical efficiency of our implementation, it turns out to be satisfactory on all examples we tried so far.

The main original contributions of the paper are:

- formulation of the concept of partial correctness of CLP programs w.r.t. parametric specifications,
- a method for proving such correctness,
- a technique for checking of parametric directional types for CLP programs, based on this method,
- a prototype tool for locating program errors based on this technique.

The paper is organized as follows. Section 2 surveys some basic concepts on set constraints and constraint logic programs. Section 3 discusses the notion of correctness of a CLP program with respect to a specification, a sufficient condition for partial correctness and a technique for constructing approximations of program semantics. The main contributions of the paper are presented in the next sections. Section 4 introduces PED grammars to be used as a parametric specification formalism for CLP programs. Section 5 introduces the notion of correctness w.r.t. a parametric specification and presents a method for proving such correctness. It shows how correctness can be effectively checked in case of parametric specifications provided as PED grammars. It also discusses how to construct a parametric specification of a given program. Finally it explains how program errors can be located by failures of the parametric correctness check. Section 6 discusses the prototype tool and illustrates its use on simple examples. Section 7 discusses relation to other work and presents conclusions.

This paper is an extended version of a less formal presentation of this work in [DMP01].

2 Preliminaries

In this section we present some underlying concepts and techniques used in our approach. We introduce set constraints and term grammars. They are a tool to define sets of terms. Then we generalize them to define sets of constrained terms. The section is concluded with an overview of basic notions of constraint logic programming (CLP).

2.1 Set Constraints

This section surveys some basic notions and results on set constraints. We will extend them later to describe approximations of the semantics of CLP programs and to specify user expectations about behaviour of the developed programs.

We build set expressions from the alphabet consisting of: variables, function symbols (including constants), the intersection symbol $\cap$ and, for every variable X, the generalized projection symbol $^{-X}$.

A set expression is a variable, a constant, or it has a form $f(e_1, \ldots, e_n)$, $e_1 \cap e_2$, or $t^{-X}(e)$, where f is an n-ary function symbol, $e, e_1, \ldots, e_n$ are set expressions, t is a term and X a variable. Set expressions built out of variables and function symbols (so including neither an intersection symbol nor a generalized projection symbol) are called *atomic*.

Set expressions are interpreted over the powerset of the Herbrand universe defined by a given alphabet. A valuation that associates sets of terms to variables extends to set expressions in a natural way: $\cap$ is interpreted as the intersection operation, each n-ary function symbol ($n \geq 0$) denotes the set construction operation

$$f(S_1, \ldots, S_n) = \{\, f(t_1, \ldots, t_n) \mid t_i \in S_i,\ i = 1, \ldots, n \,\}$$

(for any sets $S_1, \ldots, S_n$ of ground terms) and symbol t^{-X} denotes the generalized projection operation

$$t^{-X}(S) = \{\, X\theta \mid t\theta \in S,\ \theta \text{ is a substitution, } X\theta \text{ is ground} \,\}.$$

(for any term t, variable X and set S of ground terms)

Notice that we do not need special symbols for the projection operation and for the set of all terms. The latter is the value of $t^{-X}(S)$, where X does not occur in t and some instance of t is in S. Projection, defined as $f_{(i)}^{-1}(S) = \{\, t_i \mid f(t_1, \ldots, t_n) \in S \,\}$, can be expressed as $f_{(i)}^{-1}(S) = f(X_1, \ldots, X_n)^{-X_i}(S)$.

Set expressions defined above are a proper subset of some classes of set expressions discussed in literature. In particular $t^{-X}(S)$ (where X occurs in t) is a special case of the generalized membership expression of [TDT00], in the notation of that paper it is $\{\, X \mid \exists_{-X}\, t \in S \,\}$. An (unnamed) operation more general than t^{-X} has also been used in [HJ90b].

Our choice of the class of set expressions is guided by our application, which is parametric descriptive types for CLP programs. Later on we generalize set expressions to deal with sets of constrained terms (instead of terms) and to include parametric set expressions.

The set constraints we consider are of the form

$$Variable \; > \; Set\ expression$$

An interpretation of set constraints is defined by a valuation of variables as sets of ground terms. A model of a constraint is an interpretation that satisfies it when $>$ is interpreted as set inclusion $\supseteq$. Ordering on interpretations is defined by set inclusion: $I \leq I'$ iff $I(X) \subseteq I'(X)$ for every variable X. In such a case we will say that I' approximates I. It can be proved (see for instance [TDT00] and Proposition 2.9) that a collection G of such constraints is satisfiable and has the least model to be denoted $\mathcal{M}_G$. The value of a set expression e in the least model of G will be denoted by $[\![e]\!]_G$; the subscript may be omitted when it is clear from the context.

2.1.1 Term Grammars

A finite set of constraints of the form

$$Variable \; > \; Atomic\ set\ expression$$

will be called *term grammar*. The least model of such a set of constraints can be obtained by assigning to each variable X the set of all ground terms *derivable* from X in this grammar. The derivability relation $\Rightarrow_G^*$ of a grammar G is defined in a natural way: some occurrence of a variable X in a given atomic set expression is replaced by a set expression e such that $X > e$ is a constraint in G. Then $[\![X]\!]_G$ is the set of all ground terms derivable from X in G.

A set S is said to be defined by a grammar G if there is a variable X of G such that $S = [\![X]\!]_G$. A grammar rule $X > t$ will be sometimes called a *rule for* X.

Example 2.1 For the following grammar the elements of $[\![List]\!]$ can be viewed as lists of bits.

$$\begin{array}{ll} List > nil & \qquad B > 0 \\ List > cons(B, List) & \qquad B > 1 \end{array}$$

A pair $\langle X, G\rangle$ of a variable X and a grammar G uniquely determines the set $[\![X]\!]_G$ defined by the grammar; such a pair will be called a *set descriptor* (or a type descriptor). Sometimes we will say that $\langle X, G\rangle$ defines the set $[\![X]\!]_G$. By $\langle X\rangle_G$ we denote the collection of all rules of G applicable in derivations starting from X.

We will mostly use a special kind of term grammars.

Definition 2.2 A term grammar is called *discriminative* iff

- each right hand side of a constraint is of the form $f(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are variables, and
- for a given variable X and given n-ary function symbol f there is at most one constraint of the form $X > f(\ldots)$

□

It should be mentioned that discriminative term grammars are just another view of deterministic top-down tree automata [CDG$^+$97]. Variables of a grammar are states of an automaton, grammar derivations can be seen as computations of automata. Abandoning the second condition from Definition 2.2 leads to a strictly stronger formalism of non discriminative grammars equivalent to nondeterministic top-down tree automata.

We should explain our choice of the less powerful formalism of discriminative grammars. They seem to be sufficient to describe those sets which are usually considered to be types [AL94] and also easier to understand for the user, which is important in our application. One of the goals of this work is enhancing term grammars with parameters. It seems reasonable to begin with a simpler formalism. We also want to find out to which extent a simpler formalism is sufficient in practice.

2.1.2 Operations on Term Grammars

The role of discriminative grammars is to define sets of terms. One needs to construct grammars describing the results of set operations on such sets. In this section we survey some operations on discriminative grammars, corresponding to set operations. A more formal presentation is given in Section 4 where we introduce a generalization of term grammars.

Emptiness check. A variable X in a grammar G will be called *nullable* if no ground term can be derived from X in G. In other words, $[\![X]\!]_G = \emptyset$ iff X is nullable in G. To check whether $[\![X]\!]_G = \emptyset$, one can apply algorithms for finding nullable symbols in context-free grammars. This can be done in linear time [HMU01].

Let G' be the grammar G without the rules containing nullable symbols. Both grammars define the same sets, $[\![X]\!]_G = [\![X]\!]_{G'}$ for any variable X.

Construction. If $S_1, \ldots, S_n$ are defined by $\langle X_1, G_1\rangle, \ldots, \langle X_n, G_n\rangle$, where $G_1, \ldots, G_n$ are discriminative grammars with disjoint sets of variables then the set $f(S_1, \ldots, S_n)$ is defined by $\langle X, G\rangle$ where G is the discriminative grammar $\{X > f(X_1, \ldots, X_n)\} \cup G_1 \cup \ldots \cup G_n$ and X is a new variable, not occurring in $G_1, \ldots, G_n$.

Intersection. Given sets S and T defined by discriminative grammars G_1 and G_2 we construct a discriminative grammar G such that $S \cap T$ is defined by G. Without loss of generality we assume that G_1 and G_2 have no common variables. The variables of G correspond to pairs (X, Y) where X is a variable of G_1 and Y is a variable of G_2. They will be denoted $X\dot{\cap}Y$. The notation reflects the intention that $[\![(X,Y)]\!]_G = [\![X]\!]_{G_1} \cap [\![Y]\!]_{G_2}$.

Now G is defined as the set of all rules

$$X\dot{\cap}Y > f(X_1\dot{\cap}Y_1, \ldots, X_n\dot{\cap}Y_n)$$

such that there exist a rule $X > f(X_1, \ldots, X_n)$ in G_1 and a rule $Y > f(Y_1, \ldots, Y_n)$ in G_2. Notice that for given f at most one rule of this form may exist in each of the grammars. Thus G is discriminative. It is not difficult to prove that $[\![(X,Y)]\!]_G$ is indeed the intersection of $[\![X]\!]_{G_1}$ and $[\![Y]\!]_{G_2}$.

We have $S = [\![X]\!]_{G_1}$ for some X of G_1 and $T = [\![Y]\!]_{G_2}$ for some Y of G_2, hence $S \cap T$ is defined by G. Notice that G may contain nullable symbols even if G_1, G_2 do not.

Example 2.3 Consider two grammars

$$\begin{array}{llcll}
G_1: & X > a & \qquad & G_2: & Y > a \\
& X > f(Z,Z) & & & Y > f(E,Y) \\
& Z > f(X,X) & & & E > a \\
& Z > b & & & E > b \\
& Z > g(Z) & & & E > h(E)
\end{array}$$

The grammar defining the intersections of the sets defined by G_1, G_2 is

$$\begin{array}{ll}
G: & X\dot{\cap}Y > a \\
& X\dot{\cap}Y > f(Z\dot{\cap}E, Z\dot{\cap}Y) \\
& Z\dot{\cap}Y > f(X\dot{\cap}E, X\dot{\cap}Y) \\
& X\dot{\cap}E > a \\
& Z\dot{\cap}E > b
\end{array}$$

Union. It is well known that the union of sets defined by discriminative grammars may not be definable by a discriminative grammar; take for example the sets $\{f(a,b)\}$ and $\{f(c,d)\}$. Given sets S and T defined by discriminative grammars G_1 and G_2 we construct now a discriminative grammar G defining a superset of $S \cup T$.

Without loss of generality we assume that G_1 and G_2 have no common variables. The variables of G correspond to pairs (X, Y) where X is a variable of G_1 and Y is a variable of G_2. They will be denoted $X\dot{\cup}Y$. The notation reflects the intention that $[\![X]\!]_{G_1} \cup [\![Y]\!]_{G_2} \subseteq [\![(X,Y)]\!]_G$.

Now G consists of the rules of G_1, the rules of G_2 and of the least set of rules which can be constructed as follows:

- If $X > f(X_1, \ldots, X_n)$ is in G_1 and $Y > f(Y_1, \ldots, Y_n)$ is in G_2 then $X\dot{\cup}Y > f(X_1\dot{\cup}Y_1, \ldots, X_n\dot{\cup}Y_n)$ is in G,

- If $X > f(X_1, \ldots, X_n)$ is in G_1 and no rule $Y > f(Y_1, \ldots, Y_n)$ is in G_2 then $X\dot{\cup}Y > f(X_1, \ldots, X_n)$ is in G,
- If no rule $X > f(X_1, \ldots, X_n)$ is in G_1 and $Y > f(Y_1, \ldots, Y_n)$ is in G_2 then $X\dot{\cup}Y > f(Y_1, \ldots, Y_n)$ is in G

It is not difficult to see that the obtained grammar G is discriminative, and that $[\![X\dot{\cup}Y]\!]_G$ is indeed a superset of the union of $[\![X]\!]_{G_1}$ and $[\![Y]\!]_{G_2}$. If the first case is not involved in the construction the result is the union of these sets. If G_1, G_2 do not contain nullable symbols then $[\![X\dot{\cup}Y]\!]_G$ is the tuple-distributive closure of $[\![X]\!]_{G_1} \cup [\![Y]\!]_{G_2}$, i.e. the least set definable by a discriminative grammar and including $[\![X]\!]_{G_1} \cup [\![Y]\!]_{G_2}$. (We skip a proof of this fact, we do not use it later). So we are able to obtain the best possible approximation of the union by a discriminative grammar.

Example 2.4 The singleton sets $\{f(a,b)\}$ and $\{f(c,d)\}$ can be defined by the grammars:

$$G_1\colon\ X > f(A,B),\ A > a,\ B > b \qquad G_2\colon\ Y > f(C,D),\ C > c,\ D > d.$$

Applying the construction we obtain additional rules:

$$\begin{array}{lll} X\dot{\cup}Y > f(A\dot{\cup}C, B\dot{\cup}D) & A\dot{\cup}C > a & B\dot{\cup}D > b \\ & A\dot{\cup}C > c & B\dot{\cup}D > d \end{array}$$

Set inclusion. Given sets S and T defined by discriminative grammars it is possible to check $S \subseteq T$ by examination of the defining grammars.

By the assumption $S = [\![X]\!]_{G_1}, T = [\![Y]\!]_{G_2}$ for some discriminative grammars G_1, G_2 and some variables X, Y. We assume without loss of generality that G_1, G_2 do not contain nullable symbols. (Otherwise the nullable symbols may be removed as justified previously).

It follows from the definition of the set defined by term grammar that $[\![X]\!]_{G_1} \subseteq [\![Y]\!]_{G_2}$ iff for every rule of the form $X > f(X_1, \ldots, X_n)$ in G_1 there exists a rule $Y > f(Y_1, \ldots, Y_n)$ in G_2 and $[\![X_i]\!]_{G_1} \subseteq [\![Y_i]\!]_{G_2}$ for $i = 1, \ldots, n$. This corresponds to a recursive procedure where a check for X, Y corresponds to comparison of function symbols in the defining rules for X and Y, which may cause a failure, and a recursive call of a finite number of such checks. The check performed once for a given pair of variables need not be repeated. As the grammar is finite there is a finite number of pairs of variables so that the check will terminate.

For a formal description of the algorithm and a correctness proof see Section 4.4.5 where a more general inclusion check algorithm is presented.

Example 2.5 The following example illustrates inclusion checking. It shows that the set of non-empty bit lists with even length is a subset of the set of unrestricted lists which allow a more general kind of elements. Both sets are described by discriminative grammars.

$S > cons(B, Odd)$	$List > nil$
$Odd > cons(B, Even)$	$List > cons(E, List)$
$Even > nil$	$E > 0$
$Even > cons(B, Odd)$	$E > 1$
$B > 0$	$E > s(E)$
$B > 1$	

We check inclusion $[\![S]\!] \subseteq [\![List]\!]$. We show steps of this process. Each step will be characterized by three items: the checked pair of variables, the function symbols in their defining rules, the set of pairs to be checked after this step.

$(S, List)$	$(\{cons\}, \{nil, cons\})$	$\{\,(B, E), (Odd, List)\,\}$
(B, E)	$(\{0, 1\}, \{0, 1, s\})$	$\{\,(Odd, List)\,\}$
$(Odd, List)$	$(\{cons\}, \{nil, cons\})$	$\{\,(Even, List)\,\}$
$(Even, List)$	$(\{nil, cons\}, \{nil, cons\})$	$\emptyset$

Generalized projection. Assume that $S = [\![Y]\!]_G$ is defined by a discriminative grammar G. We show that $t^{-X}(S)$ is defined by a discriminative grammar.

Consider a term t and a mapping $\xi(t, G, Y)$ assigning a variable V_u of G to each subterm occurrence u of t, such that V_t is Y and if $u = f(u_1, \ldots, u_n)$ $(n \geq 0)$ then there exists a rule $V_u > f(V_{u_1}, \ldots, V_{u_n})$ in G. So for instance in Example 2.5, taking $t = cons(s(X), Z)$ and $Y = List$ results in $V_t = List$, $V_{s(X)} = E$, $V_Z = List$, $V_X = E$. If such a mapping exists then it is unique, as the grammar contains at most one rule $V > f(\ldots)$ for given V, f.

The mapping can be found by an obvious algorithm. It traverses t top-down and for each occurrence u of a non-variable subterm it finds the unique rule $V_u > f(V_{u_1}, \ldots, V_{u_n})$. The rule determines the variables $V_{u_1}, \ldots, V_{u_n}$ corresponding to the greatest proper subterms of u. If such a rule does not exist, mapping $\xi(t, G, Y)$ does not exist. The starting point is $u = t$ and $V_u = Y$.

Notice that if $t\theta \in S$ then $\xi(t, G, Y)$ exists and $u\theta \in [\![V_u]\!]_G$ for each subterm occurrence u in t. Hence $X\theta \in [\![V_{X^i}]\!]_G$ for each occurrence X^i of X in t. Thus $t^{-X}(S) \subseteq \bigcap_i [\![V_{X^i}]\!]_G$. (If X does not occur in t then $\bigcap_i [\![V_{X^i}]\!]_G$ denotes the Herbrand universe.) On the other hand, assume that $\xi(t, G, Y)$ exists and for each variable Z of t there exists a term u_Z such that $u_Z \in [\![V_{Z^i}]\!]_G$ for each occurrence Z^i of Z in t. Then $t\theta \in S$, where $\theta = \{\, Z/u_Z \mid Z \text{ occurs in } t\,\}$. Thus if $\xi(t, G, Y)$ exists and $\bigcap_i [\![V_{Z^i}]\!]_G$ is nonempty for each Z then

$$t^{-X}(S) = \bigcap_i [\![V_{X^i}]\!]_G\,.$$

Otherwise $t^{-X}(S) = \emptyset$.

Applying algorithms described previously, we can construct for each Z a distributive grammar G_Z defining $[\![Z']\!]_{G_Z} = \bigcap_i [\![V_{Z^i}]\!]_G$ and check this set for emptiness. This provides an algorithm which, given G, Y, t, produces for each X occurring in t a discriminative grammar G_X and a variable X' such that $t^{-X}(S) = [\![X']\!]_{G_X}$.

An algorithm similar to the presented above is used in the implementation of [GdW94], it is however only superficially described in that paper.

2.2 Specifying sets of constrained terms

Set constraints and term grammars are formalisms for defining subsets of the Herbrand universe. This is not sufficient for the purposes of CLP. We use a CLP semantics based on the notion of a constrained expression. The goal of this section is generalizing discriminative term grammars to a mechanism of defining sets of constrained terms.

2.2.1 Constrained expressions

CLP programs operate on constraint domains. A *constraint domain* is defined by providing a finite signature (of predicate and function symbols) and a structure $\mathcal{D}$ over this signature.[24] 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. All the function symbols have a fixed interpretation, they are interpreted as constructors. So the elements of $\mathcal{D}$ can be seen as (finite) terms built from some elementary values and the constant symbols by means of constructors. That is why we will often call them $\mathcal{D}$-terms. In CLP some function symbols have also other meaning (like $+$ denoting addition in CLP over integers). This meaning is employed only in the semantics of constraint predicates.

We treat function symbols as constructors, because this happens in the semantics of most CLP languages, like CHIP or SICStus Prolog [Cos98, SIC98]. They use syntactic unification. For instance, in CLP over integers, terms like $1{+}3$, $2{+}2$, $1{*}4$, 4 are (pairwise) not unifiable. Only the constraint predicates recognize their numerical values. So $2 + 2 \;\#{=}\; 1 * 4$ succeeds and $2{+}2 \;\#{>}\; 3{*}4$ fails (where $\#{=}$, $\#{>}$ are constraint predicates of, respectively, arithmetical equality and comparison).

By a *constraint* we mean an atomic formula with a constraint predicate, $c_1 \wedge c_2$, $c_1 \vee c_2$, or $\exists X c_1$, where c_1 and c_2 are constraints and X is a variable. We will often write c_1, c_2 for $c_1 \wedge c_2$. The fact that a constraint c is true for every variable valuation will be denoted by $\mathcal{D} \models c$.

[24] Sometimes we slightly abuse the notation and use $\mathcal{D}$ to denote the carrier of $\mathcal{D}$.

The Herbrand domain of logic programming is generalized to the constraint domain $\mathcal{D}$ of CLP. Analogical generalization of non ground atoms and terms are constrained expressions.

Definition 2.6 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. $\square$

A $c \,[]\, E$ with some free variable of c not occurring in E will be treated as an abbreviation for $(\exists \ldots c) \,[]\, E$, where all variables of c not occurring in E are existentially quantified,

Definition 2.7 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 $\mathcal{D} \models c' \to c\theta$ ($c\theta$ 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 $\bigcup_{E \in S} cl(E)$. $\square$

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 $\mathcal{D} \models c' \to c$, 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 $\mathcal{D} \models c'' \to c'\sigma$ and $\mathcal{D} \models c' \to c\theta$, we have $\mathcal{D} \models c'' \to c\theta\sigma$). 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 2.8 $a + 7$, $Z + 7$, $1{+}7$ 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) (CLP over finite domains, [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 $X{\in}S$).

2.2.2 Extended Set Constraints

We use a semantics for CLP which is based on constrained atoms/terms. To approximate such semantics we generalize term grammars to describe instance-closed sets of constrained terms. In discussing grammars and the generated sets, we will not distinguish between predicate and function symbols, and between atoms and terms.

For a given constraint domain $\mathcal{D}$, we introduce some *base sets* of constrained terms. We require that base sets are instance-closed. Following [DZ92] we extend the alphabet of set constraints by *base symbols* interpreted as base sets. Each base symbol b has a fixed corresponding set $[\![b]\!]$ of constrained terms, $[\![b]\!] \neq \emptyset$. We require that the alphabet of base symbols is finite. We assume that there is a base symbol $\top$ for which $[\![\top]\!]$ is the set of all constrained terms over given $\mathcal{D}$. Usually no other base sets contain (constrained) terms with (non constant) function symbols.

For instance in CLP over finite domains [Hen89], $\mathcal{D}$ contains terms built of symbols and integer numbers. The base sets we use for this domain are, apart from $[\![\top]\!]$, denoted by base symbols $nat, neg, anyfd$. They correspond to, respectively, the natural numbers, the negative integers and finite domain variables. The latter are represented as constrained variables of the form $X \in S \,[]\, X$, where S is a finite set of natural numbers. Due to the closedness requirement, $[\![anyfd]\!]$ contains also the natural numbers.

An *extended set expression* is an expression built out of variables, base symbols, function symbols (including constants), $\cap$ and the generalized projection symbols. Extended set expressions are interpreted as instance-closed sets of constrained terms. In the context of extended set expressions, a *valuation* is a mapping assigning instance-closed sets of constrained terms to variables.[25]

The construction and generalized projection operation for (instance closed) sets of constrained terms are defined as

$$f(S_1,\ldots,S_n) \;=\; cl(\{\, c_1,\ldots,c_n \,[]\, f(t_1,\ldots,t_n) \;\mid\; c_i \,[]\, t_i \in S_i,\; i=1,\ldots,n \,\})\,,$$

$$t^{-X}(S) = \{\, c \,[]\, X\theta \mid c \,[]\, t\theta \in S,\ \text{for some substitution } \theta \,\},$$

for instance-closed sets $S, S_1, \ldots, S_n$, a function (or predicate) symbol f, a term (or an atom) t and a variable X. Notice that $f(S_1,\ldots,S_n), t^{-X}(S)$ are instance-closed. A valuation, together with a fixed valuation of base symbols, extends in a natural way to extended set expressions. So if sets $S_1,\ldots,S_n$ are values of expressions $e_1,\ldots,e_n$ then the value of $f(e_1,\ldots,e_n)$

[25] Notice that we have two different languages using variables: the language of set expressions (and of set constraints and grammars), with variables ranging over sets of constrained terms, and the language of constrained terms with variables ranging over a specific constraint domain. In this paper we use the same notation for both kinds of variables. This should cause no confusion, the kind of a variable is determined by the context.

is $f(S_1, \dots, S_n)$. For a ground extended set expression t its value will be denoted by $[\![t]\!]$.

Extended set expressions can be used to construct set constraints and grammars. We consider *extended set constraints* of the form $X > t$, where X is a variable and t an extended set expression. An *extended term grammar* is a set of constraints (often called rules) of the form $X > t$, where t is an atomic set expression (i.e. one built out of variables, the base symbols and the function symbols, including constants).

A *model* of a set $\mathcal{C}$ of extended set constraints is a valuation I, under which $I(X) \supseteq I(t)$ for each constraint $X > t$ of $\mathcal{C}$.

Proposition 2.9 Any set $\mathcal{C}$ of extended set constraints has the least model.

PROOF We show that the set of models of $\mathcal{C}$ is nonempty and that their greatest lower bound is a model of $\mathcal{C}$.

I assigning to each variable the set $[\![\top]\!]$ of all constrained terms is a model of any extended set constraint.

The greatest lower bound of a set $\mathcal{I}$ of valuations is a valuation $\bigcap \mathcal{I}$ such that $(\bigcap \mathcal{I})(X) = \bigcap \{ I(X) \mid I \in \mathcal{I} \}$, for any variable X.

Let $\circ$ be a construction operation, a generalized projection operation or $\cap$. Let k be its arity. For $i = 1, \dots, k$, let $\mathcal{S}_i$ be a set of instance closed sets of constrained terms. We have

$$\circ(\bigcap \mathcal{S}_1, \dots, \bigcap \mathcal{S}_k) \;\subseteq\; \bigcap \{ \circ(S_1, \dots, S_k) \mid S_1 \in \mathcal{S}_1, \dots, S_k \in \mathcal{S}_k \}.$$

(We do not need here to show equality). Hence for any extended set expression t and any set $\mathcal{I}$ of valuations

$$(\bigcap \mathcal{I})(t) \;\subseteq\; \bigcap \{ I(t) \mid I \in \mathcal{I} \},$$

by induction on the structure of t. Hence if each element of $\mathcal{I}$ is a model of an extended set constraint $X > t$ then $\bigcap \mathcal{I}$ is a model of $X > t$, as $(\bigcap \mathcal{I})(X) = \bigcap \{I(X) \mid I \in \mathcal{I}\} \supseteq \bigcap \{I(t) \mid I \in \mathcal{I}\} \supseteq (\bigcap \mathcal{I})(t)$. Thus if $\mathcal{I}$ is the set of models of $\mathcal{C}$ then $\bigcap \mathcal{I}$ is a model of $\mathcal{C}$, hence the least model. □

Definition 2.10 The set defined by a variable X in an extended term grammar G is

$$[\![X]\!]_G = \{ c \,[\,]\, u \mid c \,[\,]\, u \in [\![t]\!],\; X \Rightarrow_G^* t \text{ and no variable occurs in } t \}$$

where the derivability relation $\Rightarrow_G^*$ is defined as for term grammars. □

Notice that we avoid confusion between the variables of grammars and the variables of constrained terms. The former occur in derivations, which end with ground terms built of function symbols (including constants) and of base symbols. The latter appear later on as a result of evaluation of base symbols in these ground terms.

The notation $[\![X]\!]_G$ is justified here by the following property.

Proposition 2.11 Let G be an extended term grammar and I the interpretation such that $I(X) = [\![X]\!]_G$ for each variable X. Then I is the least model of G.

PROOF Consider a variable X and a constrained term $c \, [\!] \, s \in [\![X]\!]_G$. So there exists a derivation $X \Rightarrow_G^* t$ such that $c \, [\!] \, s \in [\![t]\!]$. By induction on the length of the derivation, for any model J of G, $[\![t]\!] \subseteq J(X)$. Thus $I(X) \subseteq J(X)$. Hence $I \leq J$. $\Box$

Definition 2.12 An *extended discriminative term grammar G* is a finite set of rules of the form

$$X > f(X_1, \ldots, X_n) \qquad \text{or} \qquad X > b$$

where f is an n-ary function symbol ($n \geq 0$), $X, X_1, \ldots, X_n$ are variables and b is a base symbol. Additionally, for each pair of rules $X > t_1$ and $X > t_2$ in G the sets $[\![t_1^\top]\!]$ and $[\![t_2^\top]\!]$ are disjoint (where $u^\top$ stands for u with each occurrence of a variable replaced by $\top$). $\Box$

So no two rules $X > f(\vec{X})$, $X > f(\vec{Y})$ may occur in such a grammar. The same for $X > b$, $X > b'$ where b, b' are base symbols and $[\![b]\!] \cap [\![b']\!] \neq \emptyset$. If a discriminative grammar contains $X > f(\vec{X})$ and $X > b$ then no (constrained term) with the main symbol f occurs in $[\![b]\!]$. If the grammar contains $X > \top$ then it is the only rule for X.

The question is how to represent/approximate by such grammars the results of set operations for sets represented by such grammars, and how to check inclusion for such sets. We address these questions under some additional restrictions on base sets, which seem to be observed in base domains of CLP languages. We require that:

Requirement 2.13

- For any base symbol b different from $\top$, $f_{(i)}^{-1}([\![b]\!]) = \emptyset$ for every f, i. (So $[\![b]\!]$ does not contain elements of the form $c \, [\!] \, f(\vec{t})$, for any non constant f.)

- For each pair b_1, b_2 of distinct base symbols the base sets $[\![b_1]\!]$, $[\![b_2]\!]$ are either disjoint or one is a subset of the other. Moreover $[\![b_1]\!] \neq [\![b_2]\!]$.

The number of base symbols is finite. Their interpretation is fixed. We can construct a table showing, for each pair b_1, b_2 of base symbols, whether $[\![b_1]\!] \cap [\![b_2]\!] = \emptyset$, $[\![b_1]\!] \subseteq [\![b_2]\!]$ or $[\![b_2]\!] \subseteq [\![b_1]\!]$.

Now, the operations on grammars of Section 2.1.1 can be easily extended. Each of them traverses the rules in the argument grammars. Eventually we may reach a point when a base symbol is encountered instead of a constant. These cases are handled in a rather obvious way, using the table described above. Similarly as for discriminative term grammars, one obtains approximation of the union and exact intersection, generalized projection and construction.

We postpone a formal presentation to Section 4.4, where we deal with a generalization of grammars discussed here.

Example 2.14 Consider CLP(FD) [Hen89]. The following discriminative extended grammars describe, respectively, integer lists and lists of finite domain variables (possibly instantiated to natural numbers):

$$\begin{array}{ll}
Li > nil & Lfd > nil \\
Li > cons(Int, Li) & Lfd > cons(A, Lfd) \\
Int > nat & A > anyfd \\
Int > neg &
\end{array}$$

Knowing that $[\![nat]\!] \subseteq [\![anyfd]\!]$ we can apply the intersection operation to obtain a grammar defining $[\![Li\,]\!] \cap [\![Lfd]\!]$:

$$\begin{array}{l}
Li \,\dot{\cap}\, Lfd > nil \\
Li \,\dot{\cap}\, Lfd > cons(Int \,\dot{\cap}\, A, Li \,\dot{\cap}\, Lfd) \\
Int \,\dot{\cap}\, A > nat
\end{array}$$

The treatment of constraints by the formalism of extended term grammars is rather rough. It stems from a small number of fixed base sets of constrained terms. They are subject to a rather restrictive Requirement 2.13, which is necessary to simplify operations on grammars. In our former work [DP98] we discussed a richer system of regular sets of constrained terms. It can be seen as also allowing base sets of the form $cl(\{c \,[\!]\, x\})$, where the set of ground terms satisfying constraint c is regular. This results in substantially more complicated algorithms for grammar operations. According to our experience the simple type system presented in this paper seems sufficient.

2.3 Constraint Logic Programming

We consider CLP programs executed with the Prolog selection rule (LD-resolution) and using syntactic unification in the resolution steps. In CLP with syntactic unification, function symbols occurring outside of constraints are treated as constructors. So, for instance in CLP over integers, the goal $p(4)$ fails with the program $\{p(2{+}2)\leftarrow\}$ (but the goal $p(X{+}Y)$ succeeds). Terms 4 and 2+2 are treated as not unifiable despite having the same numerical value. Also, a constraint may distinguish such terms. For example in many constraints of CHIP, an argument may be a natural number (or a "domain variable") but not an arithmetical expression. Resolution based on syntactic unification is used in many CLP implementations, for instance in CHIP and in SICStus [SIC98].

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

An *LD-derivation* is a sequence $G_0, C_1, \theta_1, G_1, \ldots$ of goals, input clauses and mgu's (similarly to [Llo87b]). A goal is of the form $c \,[]\, A_1, \ldots, A_n$, where c is a constraint and $A_1, \ldots, A_n$ are atomic formulae (including atomic constraints). For a goal $G_{i-1} = c \,[]\, A_1, \ldots, A_n$, where A_1 is not a constraint, and a clause $C_i = H \leftarrow B_1, \ldots, B_m$, the next goal in the derivation is $G_i = (c \,[]\, B_1, \ldots, B_m, A_2, \ldots, A_n)\theta_i$ provided that θ_i is an mgu of A_1 and H, $c\theta_i$ is satisfiable and G_{i-1} and C_i do not have common variables. If A_1 is a constraint then $G_i = c, A_1 \,[]\, A_2, \ldots, A_n$ ($\theta_i = \epsilon$ and C_i is empty) provided that c, A_1 is satisfiable.

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

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

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

So the call-success semantics describes precisely the calls and the successes in the considered class of computations of a given program. The question is whether this set includes "wrong" elements, unexpected by the user. To require a precise description of user expectations is usually not realistic. On the other hand, it may not be difficult to provide an approximate description $Spec = (C', S')$ where C' and S' are sets of constrained atoms such that every expected call is in C' and every expected success is in S'.

Definition 2.15 A program P with the set of initial goals $\mathcal{G}$ is *partially correct* w.r.t. $Spec = (C', S')$ iff $C \subseteq C'$ and $S \subseteq S'$, where $(C, S) = CS(P, \mathcal{G})$ is the call-success semantics of P and $\mathcal{G}$.

P is partially correct w.r.t. $Spec = (C', S')$ iff P with C' as the set of initial goals is partially correct w.r.t. $Spec$. □

We will usually omit the word "partially".

To avoid substantial technical difficulties, we will consider only specifications that are closed under instantiation. This means that whenever set C' (or S') contains a constrained atom $c \,[]\, A$ then it contains all its instances.

In Section 5 we introduce parametric specifications, discuss a more precise semantics and generalize accordingly the notion of program correctness.

Our discussion of CLP semantics has been carried on under an assumption that the constraint solver is complete. Thus it is able to recognize all unsatisfiable constraints. However actual solvers are usually incomplete. As a result, goals with unsatisfiable constraints may appear in derivations. But the set of solutions represented by all answers of an incomplete solver is the same as the set of solutions represented by all answers of a complete solver. Thus, if our type checking technique indicates (possibility of) the existence of a wrong answer, beyond those characterized by a specification, then this answer will also be obtained with an incomplete solver. Thus the assumption on completeness of the solver is only a technicality needed for formal development of the method, which is also applicable in the case of incomplete solvers.

A specification describes calls and successes of all the predicates of a program, including the constraint predicates. As the semantics of constraints is fixed for a given programming language, their specification is fixed too. In our system it is kept in a system library and is not intended to be modified by the user. (The same happens for other built-in predicates of the language.) This fixed part of the specification may not permit some constrained atoms as procedure calls; such calls are not allowed in the language and result in run-time errors.[26]

Example 2.16 To illustrate the treatment of constraint predicates by specifications, assume that a CLP(FD) language has a constraint $\in$, which describes membership in a finite domain. Assume that invoking $\in(X, S)$ with S not being a list of natural numbers is an error. This should be reflected by the specifications of all programs using $\in$. In any such specification $Spec = (Pre, Post)$, a call of the form $c \,[]\, \in(X, S)$ is in Pre iff S is such a list. If such a call succeeds, X must be a finite domain variable or a natural number. We may thus require that $c \,[]\, \in(X, S)$ is in $Post$ iff S is a list of natural numbers and $c \,[]\, X$ is in $[\![anyfd]\!]$.

The following definition provides a condition assuring that a specification correctly approximates successes of constraint predicates.

Definition 2.17 We say that a specification $(Pre, Post)$ *respects constraints* if $c, A \,[]\, A \in Post$ whenever $c \,[]\, A \in Pre$ and c, A is satisfiable (for

[26] An exact description of the set of allowed calls of constraints is sometimes impossible in our framework, as the set may be not instance closed. For example, many constraints of CHIP have to be called with certain arguments being variables.

any constraint c and atomic constraint A). This is equivalent to

$$\{\, c, A \,[]\, A \mid c, A \text{ is satisfiable} \,\} \cap \mathit{Pre} \subseteq \mathit{Post}$$

as Pre is closed under instantiation.

□

3 Partial correctness of programs

In this section we present a verification condition for partial correctness of CLP programs. Then we express it by means of set constraints and show how to perform correctness checking and how to compute a specification approximating the call-success semantics of a program.

3.1 Verification condition

A sufficient condition for such correctness of logic programs was given in [DM88]. For specifications which are closed under substitution the condition is simpler [BC89], [Apt97]. Generalizing the latter for constraint logic programs we obtain:

Proposition 3.1 Let P be a CLP program, $\mathcal{G}$ a set of initial goals and $\mathit{Spec} = (\mathit{Pre}, \mathit{Post})$ be a specification respecting constraints and such that $\mathit{Pre}, \mathit{Post}$ are closed under instantiation.

A sufficient condition for P with $\mathcal{G}$ being correct w.r.t. Spec is:

1. For each clause $H \leftarrow B_1, \ldots, B_n$ of P, $j = 0, \ldots, n$, any substitution θ and any constraint c

$$\begin{array}{lll} \text{if} & c \,[]\, H\theta \in \mathit{Pre},\ c \,[]\, B_1\theta \in \mathit{Post},\ \ldots,\ c \,[]\, B_j\theta \in \mathit{Post} & \\ \text{then} & c \,[]\, B_{j+1}\theta \in \mathit{Pre} & \text{for } j < n \\ & c \,[]\, H\theta \in \mathit{Post} & \text{for } j = n \end{array}$$

2. $\mathcal{G} \subseteq \mathit{Pre}$

PROOF Follows from more general Theorem 5.1 applied to a specification set $\{(\mathit{Pre}, \mathit{Pre} \cap \mathit{Post})\}$. □

For simplicity we consider here only atomic initial goals. Generalization for non atomic ones is not difficult. For instance one may replace a goal $c \,[]\, \vec{A}$ by goal p and an additional clause $p \leftarrow c, \vec{A}$ in the program, where p is a new predicate symbol. Alternatively, one can provide a condition for goals similar to that for clauses [DM88], [Apt97].

Notice that the constraints in the clause are treated in the same way as other atomic formulae. As constraint predicates are not defined by program clauses, the requirement that the specification respects constraints is needed in the proposition.

The part of the specification concerning constraint predicates is fixed for a given CLP language. As already mentioned, in our system it is kept in a system library. It is the responsibility of the librarian to assure that the library specification respects constraints. This property depends on the constraint domain in question, and therefore no universal tool can be provided. The number of constraint predicates in any CLP language is finite, so is the library specification, which has only once to be proved to respect constraints.

We want to represent Proposition 3.1 as a system of set constraints. Each implication for a clause $C = H \leftarrow B_1, \ldots, B_n$ from condition 1 of the proposition can now be expressed by a system $F_j(C) = F_{j,1}(C) \cup F_{j,2}(C)$ of constraints, where $F_{j,1}(C)$ consists of

$$X \;>\; H^{-X}(\mathit{Call}) \,\cap\, \bigcap_{i=1}^{j} {B_i}^{-X}(\mathit{Success}) \tag{3}$$

for each variable X occurring in the program clause and $F_{j,2}(C)$ contains one constraint

$$\begin{array}{ll} \mathit{Call} > B_{j+1} & \text{if } j < n \\ \mathit{Success} > H & \text{if } j = n \end{array} \tag{4}$$

(The program variables occurring in the clause become variables of set constraints. As explained in Section 2.2.2, the predicate symbols are treated as function symbols.)

This constraint system has the following property.

Lemma 3.2 Let $C = H \leftarrow B_1, \ldots, B_n$ be a clause and $\mathit{Spec} = (\mathit{Pre}, \mathit{Post})$ a specification. If constraint set $F_j(C)$ has a model assigning to Call the set Pre and to $\mathit{Success}$ the set Post then implication of Proposition 3.1 holds, for any θ and c.

PROOF Assume that I is such a model. From (3) it follows that $c \,[\!]\, X\theta \in I(X)$ for each c, θ satisfying the premise of the implication and for each variable X in the clause. Now from (4) it follows that $c \,[\!]\, B_{j+1}\theta \in I(B_{j+1}) \subseteq \mathit{Pre}$, respectively $c \,[\!]\, H\theta \in I(H) \subseteq \mathit{Post}$ when $j = n$. □

Set constraints $F_j(C)$ express a sufficient condition for program correctness. If a specification is given, to check the correctness it suffices to check whether the specification extends to a model of $F_j(C)$ (for all $C \in P$ and j). In the sequel we show how to do this effectively for the case when Pre and Post are defined by discriminative extended term grammars.

If a specification is not given, Lemma 3.2 tells us that the program is correct with respect to the specification obtained from any model of $F_j(C)$

(for all C and j). An algorithm for constructing a discriminative term grammar describing a model of the constraints could thus be seen as a type inference algorithm for this program.

3.2 Correctness checking

In this section we present an algorithm for checking program correctness. We will consider specifications given by means of extended term grammars. Such a grammar G has distinguished variables $Call, Success$ and the specification is $Spec = ([\![Call]\!]_G, [\![Success]\!]_G)$ (so $Pre = [\![Call]\!]_G,\ Post = [\![Success]\!]_G$). We require that the variables of G are distinct from those occurring in the program. We also require that $Spec$ respects constraints. So such grammar can be seen as consisting of two parts: a fixed part describing the constraints and built-in predicates, and a part provided by the user.

Example 3.3 The specification of constraint predicate $\in$ from Example 2.16 can be given by the following grammar rules.

$$
\begin{array}{ll}
Call > \in(Any, Nlist) & Success > \in(Anyfd, Nlist) \\
Nlist > [\,] & Anyfd > anyfd \\
Nlist > cons(Nat, Nlist) & Nat > nat
\end{array}
$$

Consider an atom $B = \in(X, [I, J])$. Applying the generalized projection operation one can compute that $B^{-X}([\![Success]\!]) = [\![anyfd]\!]$ and $B^{-J}([\![Success]\!]) = [\![nat]\!]$.

Notice that within the formalism of extended term grammars we cannot provide a more precise specification. For instance we cannot express the fact that if $c \,[\!]\, \in(t_1, t_2)$ is a success then c constraints the value of t_1 to the numbers that occur in the list t_2 (formally: any ground element of $cl(\{c \,[\!]\, t_1\})$ is a member of t_2).

Our algorithm employs the inclusion check, intersection and generalized projection operations for extended term grammars. As already mentioned, they are rather natural generalizations of the operations for term grammars described in Section 2.1.1. The details can be found in Section 4.4, describing operations for parametric extended term grammars.

The algorithm resembles a single iteration of the iterative algorithm of [GdW94] for approximating logic program semantics, in its version with "magic transformation". However it works on extended term grammars. We provide its detailed description combined with a proof of its correctness, in order to facilitate a further generalization to parametric case.

As explained in the previous section, a sufficient condition for a program P to be correct w.r.t. $Spec$ is that for each n-ary clause C of P and for each $j = 0, \ldots, n$, constraints $F_j(C)$ have a model that coincides on $Call$ and $Success$ with the least model of G.

To find such a model we construct (a grammar describing) the least model of $F_{j,1}(C) \cup G$. Then we check if it is a model of $F_{j,2}(C)$. If yes then it is the required model of $F_j(C)$. Otherwise we show that the required model does not exist.

The first step is to compute the projections and intersections of (3). To each expression of the form $A^{-X}(Y)$ occurring in (3) we apply the generalized projection operation to construct a grammar G_A defining $A^{-X}([\![Y]\!]_G)$. Then we apply the intersection algorithm to grammars $G_H, G_{B_1}, \dots, G_{B_j}$. As a result (after appropriate renaming of the variables of the resulted grammar) we obtain a grammar G_X such that

$$[\![X]\!]_{G_X} = H^{-X}([\![Call]\!]_G) \cap \bigcap_{i=1}^{j} {B_i}^{-X}([\![Success]\!]_G).$$

and all the variables of G_X, except of X, are distinct from those of $F_j(C) \cup G$. Obviously, $[\![X]\!]_{G_X}$ is the same as $[\![X]\!]$ in the least model of $\{(3)\} \cup G$.

The first step is to be applied to each constraint (1) of $F_j(C)$ (with a requirement that the variables of the constructed grammars G_X are distinct). Let $G' = \bigcup_X G_X$ be the union of the grammars constructed in the first step. We combine G' and G, where the roles of G', G are to define values for, respectively, the variables of C and variables *Call*, *Success*. The least model of $G \cup G'$ is a model of $F_{j,1}(C) \cup G$ (and it coincides with the least model of $F_{j,1}(C) \cup G$ on $Vars(C) \cup \{Call, Success\}$, where $Vars(C)$ is the set of the variables occurring in C).

The second step is transforming (4) to a discriminative grammar G'', by applying repetitively the construction operation. Let us represent constraint (4) as $Y > A$ (so Y is *Call* or *Success* and A is B_{j+1} or H). For each subterm s of A, G'' employs a variable X_s. X_A is Y and if the given subterm s is a variable V then X_V is V. Otherwise X_s is a new variable, not occurring in C, G, G'. Grammar G'' contains the rule $X_s > f(X_{s_1}, \dots, X_{s_n})$ for each non variable subterm $s = f(s_1, \dots, s_n)$ of A. We have $[\![X_s]\!]_{G' \cup G''} = [\![s]\!]_{G'}$, for each subterm s. In particular $[\![Y]\!]_{G' \cup G''} = [\![A]\!]_{G'} = [\![A]\!]_{G \cup G'}$.

This completes the construction. We may say that $F_j(C)$ was transformed into a discriminative grammar $F_{C,j} = G' \cup G''$.

It remains to check whether $[\![Y]\!]_{G' \cup G''} \subseteq [\![Y]\!]_G$. If yes then $[\![A]\!]_{G \cup G'} \subseteq [\![Y]\!]_{G \cup G'}$, i.e. the least model of $G \cup G'$ is a model of $A < Y$. Thus it is the model of $F_j(C) \cup G$ required in Lemma 3.4.

Otherwise, notice first that if $F_1 \subseteq F_2$ then $[\![X]\!]_{F_1} \subseteq [\![X]\!]_{F_2}$, for constraint sets F_1, F_2. So we have $[\![Y]\!]_{G' \cup G''} = [\![A]\!]_{G \cup G'} = [\![A]\!]_{F_{j,1}(C) \cup G} \subseteq [\![A]\!]_{F_j(C) \cup G} \subseteq [\![Y]\!]_{F_j(C) \cup G}$. Thus $[\![Y]\!]_{G' \cup G''} \not\subseteq [\![Y]\!]_G$ implies $[\![Y]\!]_{F_j(C) \cup G} \not\subseteq [\![Y]\!]_G$. Hence $I(Y) \not\subseteq [\![Y]\!]_G$ for any model I of $F_j(C) \cup G$ and the required model of $F_j(C) \cup G$ does not exist.

Thus we obtained:

Lemma 3.4 The implication from Proposition 3.1 holds for a clause C and a number j if $[\![Y]\!]_{G' \cup G''} \subseteq [\![Y]\!]_G$, for grammars G', G'' constructed as above.

The inclusion can be checked by applying the inclusion algorithm (preceded by removing nullable symbols).

We now estimate the complexity of the algorithm. The cost of the intersection operation applied to two grammars with respectively v_1, v_2 variables is $O(v_1 v_2)$. The cost of removing nullable symbols is linear [HMU01].

Let us now consider the inclusion check. We may assume that grammars are stored so that the productions for each variable are kept together and ordered. Let v_1, v_2 be the numbers of variables in the grammars. For each encountered pair X, Y of variables, it has to be checked whether the pair has not occurred previously $(O(\log(v_1 v_2)))$ and the productions for X and for Y are to be found $(O(\log(v_1) + \log(v_2)))$. The pairs of productions with the same function symbol can be found in time proportional to the number of function symbols occurring in the productions found. For each pair of productions $X > f(\ldots)$, $Y > f(\ldots)$ new variable pairs are generated, their number is the arity of f. Taking as constants the maximal arity and the maximal number of function symbols in the productions for a given variable, we obtain $O(\log(v_1 v_2))$ per pair. So the total cost of inclusion check is $O(v_1 v_2 \log(v_1 v_2))$. This cost is not changed when the costs of initial sorting of the grammars are taken into account.

Notice that in our algorithm the results of all the generalized projections and intersections computed in the step for j can be reused in the next steps. Taking into account the intersections needed to compute the projections, there are $k-1$ intersections to be computed for each variable occurring k times in the clause C. The cost of computing such a k-fold intersection and the size of resulting grammar is $O(v^{k-1})$, where v is the number of variables in the specification grammar G.

Computing mappings ξ in the projections and constructing all the G'' is linear in the size of the clause. Inclusion checking for a pair of grammars with respectively $O(v^{k-1})$ and v variables can be done in time $O(v^k \log(v^k)) = O(c^k)$, where constant c depends on the number of variables in the grammar.

Thus the correctness checking algorithm described in this section works in time $O(c^k)$, where k is the maximal number of occurrences of a variable in a clause.

Example 3.5 Consider the program

```
app([],V,V).
app([A|X],Y,[A|Z]) :- app(X,Y,Z).
```

The verification conditions can be expressed as three constraint systems (we abbreviate $H = app([A|X], Y, [A|Z])$, $B = app(X, Y, Z)$):

$$V > app([\,], V, V)^{-V}(Call)$$
$$Success > app([\,], V, V)$$

$$\begin{array}{l} A > H^{-A}(Call) \\ X > H^{-X}(Call) \\ Y > H^{-Y}(Call) \\ Z > H^{-Z}(Call) \\ Call > app(X, Y, Z) \end{array}$$

$$\begin{array}{l} A > H^{-A}(Call) \ \cap \ B^{-A}(Success) \\ X > H^{-X}(Call) \ \cap \ B^{-X}(Success) \\ Y > H^{-Y}(Call) \ \cap \ B^{-Y}(Success) \\ Z > H^{-Z}(Call) \ \cap \ B^{-Z}(Success) \\ Success > H \end{array} \tag{5}$$

Let the following extended term grammar G provide a specification.

$$\begin{array}{l} Call > app(L, L, Any) \\ Success > app(L, L, L) \\ L > [\,] \\ L > [M|L] \\ Any > \top \end{array}$$

where M is further specified by grammar rules not presented here. We assume that M is not nullable in G.

Using the described techniques one can check that the specification defines a model for all above stated set constraint systems. For example we check the constraints (5). To compute the projections related to atom $H = app([A|X], Y, [A|Z])$ and $Call$ we first obtain the following mapping between the subterm occurrences in H and the variables of G.

$$\begin{array}{ll} V_{[A|X]} = L & V_{A^1} = M \\ V_{[A|Z]} = Any & V_{A^2} = Any \\ & V_X = V_Y = L \\ & V_Z = Any \end{array}$$

Similarly, for the projections related to atom $B = app(X, Y, Z)$ and $Success$, we have

$$V_X = V_Y = V_Z = L$$

The grammar describing $H^{-A}(Call)$ is $G \dot{\cap} G$ with a distinguished variable $M \dot{\cap} Any$. The clauses of $G \dot{\cap} G$ for $M \dot{\cap} Any$ are $\{ M \dot{\cap} Any > t \mid M > t \in G \}$. (Also $G \subseteq G \dot{\cap} G$.) $M \dot{\cap} Any$ is not nullable in $G \dot{\cap} G$, as M is not nullable in G.

Notice that $B^{-A}(Call) = [\![\top]\!]$ (as A does not occur in B). All the other projections from (5) are given by variable L or Any and grammar G.

Now we construct grammar G' for which

$$\begin{array}{l} [\![A]\!]_{G'} = [\![M \dot{\cap} Any]\!]_{G \dot{\cap} G} \cap [\![\top]\!] \\ [\![X]\!]_{G'} = [\![L]\!]_G \cap [\![L]\!]_G \\ [\![Y]\!]_{G'} = [\![L]\!]_G \cap [\![L]\!]_G \\ [\![Z]\!]_{G'} = [\![Any]\!]_G \cap [\![L]\!]_G \end{array}$$

Computing intersections (and renaming variables where necessary) results in a grammar G' consisting of the rules

$$\begin{array}{lll} \{\, A > t \mid M > t \in G \,\} \;\cup\; \{\, X > [\,], & Y > [\,], & Z > [\,], \\ \qquad\qquad\qquad\qquad\quad X > [M|X], & Y > [M|Y], & Z > [M|L] \,\} \end{array}$$

and the rules of G except for those for *Call*, *Success*. (Before constructing the grammar we simplified $[\![M \dot{\cap} Any]\!]_{G \dot{\cap} G} \cap [\![\top]\!]$ to $[\![M \dot{\cap} Any]\!]_{G \dot{\cap} G}$ and $[\![L]\!]_G \cap [\![L]\!]_G$ to $[\![L]\!]_G$. Formally, G' has variables distinct from those of G.) Variables A, X, Y, Z are not nullable in G'.

The least model of G' provides a valuation for variables A, X, Y, Z. It remains to check that for this valuation, together with the valuation for *Success* given by the specification G, the constraint $Success > app([A|X], Y, [A|Z])$ holds. To do this we transform this constraint into a discriminative grammar G'':

$$\begin{array}{l} Success > app(X_1, Y, X_2) \\ X_1 > [A|X] \\ X_2 > [A|Z] \end{array}$$

and apply the set inclusion algorithm to check whether the set defined by *Success* in the specification grammar G is a superset of that defined by *Success* in the obtained grammar $G' \cup G''$. The check succeeds. Hence there exists a model for the considered five constraints which agrees on variables *Call* and *Success* with the model given by the specification. Notice that this holds independently of the missing fragment of G defining M.

The same procedure can be performed for all the constraint systems generated for the given program, hence confirming that the program is correct w.r.t. the parametric specification. Also in these cases the correctness check is independent from $\langle M \rangle_G$ (the part of G defining M).

In our example the correctness check was independent from a subset $\langle M \rangle_G$ of the specification grammar G. This is not uncommon, for some programs and specification grammars a correctness check refers only to some rules of the grammar. Thus a single check is valid for a whole family of grammars. This phenomenon will be exploited in our approach to parametric specifications.

3.3 Approximating program semantics

In this work we are mainly interested in checking program correctness. However the representation of the verification condition (Proposition 3.1) as constraints (Lemma 3.2) can be used to obtain an approximation of the semantics of a given program P. In the previous section we showed how a single implication from Proposition 3.1 can be expressed by a constraint system $F_j(C)$. We begin with constructing a constraint system representing all the implications from the proposition.

Let us consider the constraints $F_j(C)$ $(j = 1, \ldots, n_C)$ for each clause C of P with n_C body atoms. Let $F'_j(C)$ be $F_j(C)$ with the variables renamed in such a way that the only common variables of (distinct) $F'_{j_1}(C_1)$, $F'_{j_2}(C_2)$ are *Call* and *Success*. Let grammar G_0 specify the initial goals and the constraint predicates. So $[\![\mathit{Call}]\!]_{G_0}$ is the set of initial goals and of the allowed calls of constraints. $[\![\mathit{Success}]\!]_{G_0}$ is (a superset of) the set of possible successes of constraint predicates.[27] Thus $([\![\mathit{Call}]\!]_{G_0}, [\![\mathit{Success}]\!]_{G_0})$ respects constraints.

Now any model I of the constraint system

$$\mathcal{C}(P) = \bigcup_{C\in P} \bigcup_{j} F'_j(C) \ \cup \ G_0$$

gives a specification $\mathit{Spec} = (I(\mathit{Call}), I(\mathit{Success}))$ with respect to which P is correct, provided that *Spec* respects constraints. This follows immediately from Lemma 3.2.

In the special case of logic programs a model of $\mathcal{C}(P)$ can be found by using the techniques for set constraint solving. For example the technique of Heintze and Jaffar [HJ90a, HJ91] produces a (non-discriminative) term grammar specifying the least model of set constraints. This technique has been used for generating approximations of logic program semantics [HJ90b, Hei92a, HJ94, CP98]. Another constraint solving approach that uses tree automata techniques, has been presented in [DJMT97, TDT00]. We expect that these techniques can be generalized to the case of CLP programs, but we did not investigate this issue yet.

Yet another approach to finding a model of the constraint system $\mathcal{C}(P)$ stems from abstract interpretation techniques (among others [JB92, HCC95], [GdW94], we generalize the latter work in [DP99, DMP00a, DMP00b] and here). $\mathcal{C}(P)$ is seen as a valuation transformer, its fixed points are models of $\mathcal{C}(P)$. Valuations are represented as discriminative grammars. A fixed point is computed iteratively.

To augment our system with a tool for computing approximations of program semantics, we provide a solution based on the latter idea. This choice was guided mainly by possibility of reusing our correctness checking algorithm and the implementation of [GdW94].

The correctness checking algorithm of the previous section can be easily modified to compute the valuation transformer related to $\mathcal{C}(P)$. This gives

[27] This approach can also be used when P is a fragment of a program, i.e. the clauses defining some predicates are missing in P. Then the semantics of such predicates has to be specified by G_0. The algorithm treats them as the constraint predicates. Examples of such program fragments are programs using built-in predicates, unfinished programs or modules of some bigger programs.

an implementation of a single step of the iteration. It remains to combine it with some technique of assuring termination.

Iteration step. Take G_i (initially G_0). To each $F'_j(C) \cup G_i$ apply the construction as in the correctness checking, obtaining a discriminative grammar $F_{C,j}$. (It is required that all the obtained grammars have distinct variables, except *Call* and *Success*). For each $F_{C,j}$, the variables occurring in $F_{C,j}$ are distinct from those in G_i except for *Call* or *Success*.

The constraints of $F'_j(C)$ are satisfied if the occurrences of *Call*, *Success* in the right hand side of each constraint of the form (3) (Section 3.1) are valuated as in the least model of G_i, and the remaining variable occurrences as in the least model of $F_{C,j}$. This follows from the discussion in the previous section.

The obtained grammar $G'_i = G_i \cup \bigcup_{C \in P} \bigcup_j F_{C,j}$ is not discriminative, due to the rules for *Call* and for *Success*. Construct a discriminative approximation of G'_i, more precisely a discriminative grammar G_{i+1} such that $[\![Call]\!]_{G'_i} \subseteq [\![Call]\!]_{G_{i+1}}$ and the same for *Success*. This is done by applying the union operation of Section 2.1.1 to G_i and all grammars $F_{C,j}$. (So G_{i+1} is $G_i \dot\cup \dot\bigcup_{C \in P} \dot\bigcup_j F_{C,j}$ with the variable $Call \dot\cup \ldots \dot\cup Call$ renamed into *Call* and $Success \dot\cup \ldots \dot\cup Success$ renamed into *Success*.)

The obtained grammar G_{i+1} has the following property. $\mathcal{C}(P) - G_0$ is true when *Call* and *Success* in all the constraints of the form (3) (Section 3.1) are valuated as in the least model of G_i, *Call* and *Success* in the constraints of the form (4) (Section 3.1) as in the least model of G_{i+1}, and the (renamed) variables of P as in the least model of G'_i.

It remains to check whether the specification given by G_{i+1} does not contain incorrect calls of constraint predicates. This boils down to checking whether all the calls of constraint predicates from the set $[\![Call]\!]_{G_{i+1}}$ are also members of $[\![Call]\!]_{G_0}$. The latter is equivalent to $[\![Call]\!]_F \subseteq [\![Call]\!]_{G_0}$, where $F = G_{i+1} - \{\, Call{>}A \mid A \text{ is not a constraint} \,\}$. Failure of the check means that we are unable to construct a specification which respects constraints. This suggests a program error and an appropriate warning is issued.

This completes an iteration step. Notice that the calls and successes of constraint predicates specified by G_{i+1} are the same as those specified by G_i and thus by G_0 (induction on i). For calls it follows from succeeding of the checks above. For successes we have that any clause $Success > p(\vec{X})$ from G_{i+1}, where p is a constraint predicate, occurs also in G_i.

The iteration is terminated if a fixpoint is reached, this means when $[\![Call]\!]_{G_{i+1}} \subseteq [\![Call]\!]_{G_i}$ and $[\![Success]\!]_{G_{i+1}} \subseteq [\![Success]\!]_{G_i}$. (The inclusion in the other direction holds for each i). The required model of $\mathcal{C}(P)$ is a valuation in which the values of the variables from G_0, except for *Call* and *Success*, are as in the least model of G_0, the values of *Call*, *Success* are as in the least model of G_i, and the variables of P are valuated by the least model of G'_i.

As a result we obtain that whenever the iteration terminates, program P is correct w.r.t. the specification given by the obtained grammar G_i.

Notice that this is justified in a different way than usually done in abstract interpretation. Instead of relating a single iteration step to the concrete semantics of the program, we showed that the obtained fixpoint satisfies a sufficient condition for program correctness.

Termination. Usually the iterative process described above does not terminate. It should be augmented with means of assuring termination. The idea is to apply a *restriction* operator $\mathcal{R}$ that maps an infinite domain of grammars to its finite subset. Moreover, the operator $\mathcal{R}$ computes an approximation of a grammar G (i.e. $[\![Call]\!]_G \subseteq [\![Call]\!]_{\mathcal{R}(G)}$ and $[\![Success]\!]_G \subseteq [\![Success]\!]_{\mathcal{R}(G)}$). The operator is applied in every iteration step: the newly obtained grammar G_{i+1} is replaced by a grammar $H_{i+1} = \mathcal{R}(G_{i+1})$. In this way we obtain a sequence of grammars $G_0, H_1, H_2, \ldots$, the sequence has the properties described in the previous paragraphs. Since the co-domain of $\mathcal{R}$ is finite, the set of grammars $\{ G_0, H_1, H_2, \ldots \}$ is finite and the iteration terminates. This technique can be seen as an instance of widening [CC92a].

An attempt at such approach was made by Gallagher and de Waal [GdW94]. Unfortunately, the termination proof given by the authors is erroneous and Mildner [Mil99] showed an artificial example which results in an infinite loop.

We adapt a technique presented in [Mil99], Section 6.5, and inspired by [JB92]. We describe it briefly. Let the *principal label* of a variable X be the set of function symbols occurring in the right hand sides of the rules defining X in a given grammar G. Let a *term grammar graph* be a directed graph with grammar variables as vertices. An edge (X, Y) belongs to the graph iff there is a rule $X > f(\ldots, Y, \ldots)$ in the grammar. The operator $\mathcal{R}$ computes an approximation of a grammar G ($[\![Call]\!]_G \subseteq [\![Call]\!]_{\mathcal{R}(G)}$ and $[\![Success]\!]_G \subseteq [\![Success]\!]_{\mathcal{R}(G)}$) assuring at the same time that there is a spanning tree of the graph of $\mathcal{R}(G)$ such that each branch of the tree contains no more than k variables with the same principal label. Since the grammar is discriminative, and since there is a finite number of function symbols in a program, the set of such spanning trees (modulo variable renaming), is finite and consequently the co-domain of $\mathcal{R}$ (modulo variable renaming) is finite. We usually apply $k = 1$.

The reasoning above does not provide any useful estimation of the complexity of the algorithm. Our experience shows that it is sufficiently efficient to compute directional types of medium size programs.

There exist variants of this method, taking into account a number of occurrences of a single function symbol along a path or just simply binding a depth of the spanning tree with a constant.

Another possibility to cope with the termination problem is to restrict the class of grammars so that the class of defined sets is a partial order of finite heights[28].

4 Parametric Set Constraints

4.1 Motivation

In Example 3.5, the correctness checking of the program was done without referring to a missing fragment $\langle M \rangle_G$ of the grammar that provided the specification. This was due to the fact that the constraints did not include generalized projections of $\langle M, G \rangle$ and all intersections involving M were of the form $M \cap M$ or $M \cap Any$, where Any is defined by clause $Any > \top$. The meaning of such expressions is preserved if we simplify them to M. As a result we obtained a term grammar referring to M. The obtained solution is parametric in the sense that it will hold for any specific choice of the missing fragment of the grammar. Thus the example demonstrates parametric polymorphism of *append*, where calls and successes are approximated by sets determined by the same specific M. This kind of parametric polymorphism is useful in locating program errors (cf. the examples in Section 6). In the rest of this section we extend previously introduced basic concepts to be able to handle parameters.

4.2 Syntax and Semantics

To define a notion of a parametric set constraint we extend the alphabet. In addition to the symbols discussed in Section 2.1 we assume that the alphabet also includes *parameters* disjoint with the other categories of symbols. Parameters will be denoted by Greek letters $\alpha, \beta, \ldots$. A *parametric set expression* is a parameter, a variable, a constant, or it has a form $f(e_1, \ldots, e_n)$, $t^{-X}(e)$ or $e_1 \cap e_2$, where f is an n-ary function symbol, t is a term, X a variable and $e, e_1, \ldots, e_n$ are parametric set expressions. Notice, that this definition extends the usual definition of set expressions, so that a usual set expression without parameters becomes a special case of a parametric set expression. A parametric term expression is *atomic* if it does not include projection and intersection symbols.

For a given valuation of the variables, a parametric set expression denotes a function from valuations of parameters to subsets of the Herbrand universe. The value of the function for a specific valuation of parameters is determined by considering parameters to be additional variables of the set expression.

[28] For example Boye [Boy96] suggested that the inference is always done with a finite lattice of types. In practice this means that for a class of applications we may have a finite library of types, represented by grammars, which may be extended by need. This will also facilitate communication with the user who will easier understand standard application-specific types than the types represented by automatically generated grammars.

We will consider *parametric set constraints* of the form

$$Variable > Parametric\ set\ expression.$$

As discussed above, a collection of non-parametric set constraints has the least model which can be defined by a term grammar. A similar property holds in the parametric case. Take a collection $\mathcal{C}$ of parametric set constraints and treat the parameters as variables. For any given fixed valuation I of the parameters there exists the least model out of the models of $\mathcal{C}$ coinciding with I on the parameters. (This can be proved similarly as Proposition 2.9).

In order to deal with sets of constrained terms parametric set expressions can be generalized to parametric extended set expressions. This is done by permitting base symbols to appear in the expressions. Parametric extended set expressions give rise to parametric extended set constraints. For any fixed valuation of parameters, a collection of such constraints has the least model. (Proof as in Proposition 2.9).

4.3 Parametric Term Grammars

Our parametric specifications will be expressed by parametric grammars. We first introduce parametric term grammars and a notion of an instance of such a grammar. Such instances define sets of terms. Then we extend this approach to define sets of constrained terms.

Definition 4.1 A *parametric term grammar* G is a finite collection of parametric set constraints of the form $X > t$ where X is a variable and t is an atomic parametric set expression. $\square$

For instance we can consider the grammar G of Example 3.5 as a parametric grammar with one parameter M.

In the context of parametric grammars, a (parametric) set descriptor is a pair $\langle X, G\rangle$ where G is a parametric grammar and X a variable or a parameter. The derivability relation is defined in the same way as for non-parametric term grammars. Notice, however, that the normal forms may include parameters.

Parameterless grammars are used to define sets, the role of parametric grammars is to define mappings on sets. This is done by assigning sets to the parameters of a grammar. The sets are given by some other grammars.

Let G be a parametric grammar such that $\alpha_1, \ldots, \alpha_k$ are all parameters occurring in G. Sometimes we will denote it $G(\vec{\alpha})$ where $\vec{\alpha} = (\alpha_1, \ldots, \alpha_k)$. A function Φ that maps each parameter α_i of G into a set descriptor $\langle X_i, G_i\rangle$ is called, abusing the standard terminology, a *parameter valuation* for G. For a given $\vec{\alpha}$ we will sometimes represent a $\Phi = \{\, \alpha_1 \mapsto \langle X_1, G_1\rangle, \ldots, \alpha_k \mapsto \langle X_k, G_k\rangle \,\}$ as the vector $(\langle X_1, G_1\rangle, \ldots, \langle X_k, G_k\rangle)$.

Definition 4.2 Let G be a parametric term grammar and let $\Phi = \{\alpha_1 \mapsto \langle X_1, G_1\rangle, \ldots, \alpha_k \mapsto \langle X_k, G_k\rangle)\}$ be a parameter valuation.

An *instance* of G under Φ is the parametric grammar $G(\Phi) = G' \cup G'_1 \cup \ldots \cup G'_k$, where

- $\langle X'_i, G'_i\rangle$ are obtained by renaming apart all variables in each $\langle X_i, G_i\rangle$ so that the grammar G and descriptors $\langle X'_1, G'_1\rangle, \ldots, \langle X'_k, G'_k\rangle$ have pairwise disjoint sets of variables.
- G' is obtained by replacing each parameter α_i in G by X'_i.

□

If $G(\Phi)$ contains no parameters then the usual notion of the sets defined by a grammar applies to $G(\Phi)$.[29] For each its variable X it defines a set, which is $[\![X]\!]_{G(\Phi)}$. So a parametric grammar $G(\alpha_1, \ldots, \alpha_k)$ defines a mapping from the sets corresponding to descriptors $\langle X_1, G_1\rangle, \ldots, \langle X_k, G_k\rangle$ to the sets defined by the grammar $G(\Phi)$. Moreover, $G(\Phi)$ defines the value for each parameter α_i of G: $[\![\alpha_i]\!]_{G(\Phi)} = [\![X'_i]\!]_{G(\Phi)}$.

The definition of an instance generalizes in an obvious way from parametric grammars to sets of (extended) parametric set constraints.

Definition 4.3 A parametric term grammar is *discriminative* if

- each right hand side of a rule is of the form $f(X_1, \ldots, X_n)$ where each X_i is a variable or a parameter.
- for a given variable X and given n-ary ($n \geq 0$) function symbol f there is at most one rule of the form $X > f(\ldots)$.

□

Notice that the instance of a discriminative grammar under a parameter valuation over discriminative grammars is discriminative.

Example 4.4 Let grammar $G(\alpha)$ be

$$List > nil \qquad\qquad List > cons(\alpha, List)$$

This grammar is discriminative. Consider $\Phi = \{\,\alpha \mapsto \langle List, G\rangle\,\}$. Since Φ shares variables with G we rename it apart to obtain $\langle List1, G'\rangle$, where G' is:

$$List1 > nil \qquad\qquad List1 > cons(\alpha, List1)$$

(The parameters are not renamed, since they are not variables). $G(\Phi)$ is

$$\begin{array}{ll} List > nil & List1 > nil \\ List > cons(List1, List) & List1 > cons(\alpha, List1) \end{array}$$

[29] It applies also to any parametric grammar H and to each variable X such that $\langle X\rangle_H$ is parameterless.

We will use the following notation, when it does not lead to ambiguity. Let G be a discriminative parametric grammar, X a variable and $\vec{\alpha} = (\alpha_1, \ldots, \alpha_k)$ the parameters occurring in G. By the *(parametric) type* $X(\vec{\alpha})$ we mean the family of sets defined by X in G (more precisely the mapping from parameter valuations to sets, assigning $[\![X]\!]_{G(\Phi)}$ to Φ). In the special case of a parameterless grammar G, type X is the set $[\![X]\!]_G$. Let $\Phi = \{\, \alpha_1 \mapsto \langle X_1, G_1\rangle, \ldots, \alpha_k \mapsto \langle X_k, G_k\rangle \}$ be a parameter valuation, where the grammars are discriminative and the parameters occurring in G_i are $\vec{\alpha}_i$, for $i = 1, \ldots, k$. Then by type $X(X_1(\vec{\alpha}_1), \ldots, X_k(\vec{\alpha}_k))$ we mean the family of sets defined by X in grammar $G(\Phi)$.

For instance the mapping corresponding to variable *List* in grammar $G(\alpha)$ of the last example can be called $List(\alpha)$. The mapping corresponding to *List* in $G(\Phi)$ can be called $List(List(\alpha))$.

Instances of parametric discriminative term grammars define sets of terms. Similarly as in the non parametric case, we generalize this formalism to specify sets of constrained terms. Assume a fixed constraint domain $\mathcal{D}$.

Definition 4.5 A *discriminative parametric extended term grammar (PED grammar)* G is a finite set of rules of the form

$$X > f(X_1, \ldots, X_n) \qquad \text{or} \qquad X > b$$

where f is an n-ary function symbol ($n \geq 0$), X is a variable, $X_1, \ldots, X_n$ are variables or parameters and b is a base symbol. Additionally, for each pair of rules $X > t_1$ and $X > t_2$ in G the sets $[\![t_1^\top]\!]$ and $[\![t_2^\top]\!]$ are disjoint (where $u^\top$ stands for u with each occurrence of a variable or a parameter replaced by $\top$). □

The definition of an instance of a grammar applies to parametric extended grammars too. A parameterless instance of such grammar defines a set of constrained atoms for each variable, as described in Section 2.2.2.

Example 4.6 Take the grammar $G(\alpha)$ from the previous example. Using $\Phi = \{\, \alpha \mapsto \langle Any, \{Any{>}\top\}\rangle \}$ we obtain $G(\Phi)$ defining lists of arbitrary constrained terms. Formally, $\langle List, G(\Phi)\rangle$ defines the set $\{\, c [\!]\, [t_1, \ldots, t_n] \mid n \geq 0,\ t_i \text{ are terms} \,\}$ (as any term of the form $[\top, \ldots, \top]$ can be generated from *List* in grammar $G(\Phi)$.

4.4 Operations on extended parametric term grammars

We now extend the operations of Section 2.1.2 to extended parametric discriminative term grammars. For each of them we show how the resulting grammar approximates a relevant set operation for each parameterless instance of the arguments.

4.4.1 Emptiness Check and Construction

A variable X in a PED grammar G will be called *nullable* if no variable-free term (i.e. a term consisting entirely of function symbols, base symbols and parameters) can be derived from X in G. So for a nullable X, $[\![X]\!]_{G(\Phi)} = \emptyset$ independently from Φ. Similarly as in non parametric case, algorithms for finding nullable symbols in context-free grammars can be applied here. Notice that for a non nullable X there exists a Φ such that $[\![X]\!]_{G(\Phi)} \neq \emptyset$ (provided that the grammar does not contain a base symbol b, for which $[\![b]\!] = \emptyset$).

The construction operation extends naturally to parametric grammars. Let $\langle X_1, G_1\rangle, \ldots, \langle X_n, G_n\rangle$ be set descriptors with pairwise disjoint variables and let f be an n-ary function symbol. By $f(\langle X_1, G_1\rangle, \ldots, \langle X_n, G_n\rangle)$ we denote set descriptor $\langle Y, G\rangle$, where Y is a new variable and

$$G = \{\, Y > f(X_1, \ldots, X_n) \,\} \cup G_1 \cup \ldots \cup G_n$$

(When the set descriptors have some common variables then $f(\langle X_1, G_1\rangle, \ldots, \langle X_n, G_n\rangle)$ can be defined by renaming apart the variables in the descriptors). Clearly:

Proposition 4.7 For any parameter valuation Φ the set descriptors $f(\langle X_1, G_1\rangle, \ldots, \langle X_n, G_n\rangle)(\Phi)$ and $f(\langle X_1, G_1(\Phi)\rangle, \ldots, \langle X_n, G_n(\Phi)\rangle)$ are identical (up to renaming of the variables introduced while building the grammar instances and of the variable introduced by the construction operation).

If $G_1(\Phi), \ldots, G_n(\Phi)$ do not contain parameters then

$$f([\![X_1]\!]_{G_1(\Phi)}, \ldots, [\![X_n]\!]_{G_n(\Phi)}) = [\![Y]\!]_{f(\langle X_1, G_1\rangle, \ldots, \langle X_n, G_n\rangle)(\Phi)}$$

□

4.4.2 Intersection

Let G_1 and G_2 be PED grammars. We assume without loss of generality that they have no common variables, but they may have common parameters. We define an operation $\dot\cap$ on such grammars; the result is a PED grammar $G_1 \dot\cap G_2$. The variables of $G_1 \dot\cap G_2$ include the variables of G_1, the variables of G_2 and new variables corresponding to pairs (X, Y) where X is a variable of G_1 and Y is a variable of G_2. The latter will be denoted $X \dot\cap Y$.

We define $G_1 \dot\cap G_2$ to consist of the rules of G_1, those of G_2 and for each $X > s \in G_1$ and $Y > t \in G_2$ at most one rule as described below.

- $X \dot\cap Y > f(s_1 \circ t_1, \ldots, s_n \circ t_n)$ $(n \geq 0)$, provided that $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$ and $s_i \circ t_i$ is the following symbol:

  1. it is the variable $s_i \dot\cap t_i$, if s_i and t_i are variables,

2. it is s_i, if s_i and t_i are parameters,
3. it is the variable Y, if one of the terms s_i, t_i is Y and the other is a parameter.

- $X \dot{\cap} Y > u$, provided that at least one of s, t is a base symbol and the following holds. Let us denote $\{s_1, s_2\} = \{s, t\}$ where s_1 is a base symbol. Now
  - $s_1 = \top$ and $u = s_2$, or $s_2 = \top$ and $u = s_1$, or
  - s_2 is a constant $c \in [\![s_1]\!]$ and u is c, or
  - s_2 is a base symbol, $[\![s_1]\!] \subseteq [\![s_2]\!]$ and $u = s_1$, or $[\![s_2]\!] \subseteq [\![s_1]\!]$ and $u = s_2$.[30]

Some decisions in this construction are arbitrary. Instead of choosing $s_i \circ t_i$ to be s_i when both s_i, t_i are parameters, one may choose t_i. For the case of s_i, t_i being a parameter and a variable one may choose $s_i \circ t_i$ to be the parameter. In the latter case we expect that our choice gives more useful results when further operations are applied to $G_1 \dot{\cap} G_2$, as a variable corresponds to a known set of rules while a parameter does not.

We notice that by construction $G_1 \dot{\cap} G_2$ is a PED grammar and all its parameters (if any) appear in G_1 or in G_2. The construction guarantees also the following property.

Proposition 4.8 For every parameter valuation Φ such that $G_1(\Phi)$ and $G_2(\Phi)$ are parameterless grammars we have

$$[\![X]\!]_{G_1(\Phi)} \cap [\![Y]\!]_{G_2(\Phi)} \subseteq [\![X \dot{\cap} Y]\!]_{(G_1 \dot{\cap} G_2)(\Phi)}$$

for all variables X in G_1 and Y in G_2.

PROOF Denote $G_1 \dot{\cap} G_2$ by G. It is sufficient to show that if $X \Rightarrow^*_{G_1(\Phi)} t$, $Y \Rightarrow^*_{G_2(\Phi)} u$ and $[\![t]\!] \cap [\![u]\!] \neq \emptyset$ then there exists a term w such that $X \dot{\cap} Y \Rightarrow^*_{G(\Phi)} w$ and $[\![t]\!] \cap [\![u]\!] \subseteq [\![w]\!]$. The proof is by induction on $\max(|t|, |u|)$ (where $|s|$ is the size of a term s).

If $t = \top$ then $X > \top \in G_1$, $X \dot{\cap} Y \Rightarrow^*_{G(\Phi)} u$ and u is the required w. Similarly, w is t in the symmetric case of $u = \top$.

If none of t, u is $\top$ and one of them is a base symbol then the other is a base symbol or a constant. Two cases are possible: $[\![t]\!] \subseteq [\![u]\!]$, rule $X \dot{\cap} Y > t$ is in G and $w = t$, or $[\![t]\!] \supseteq [\![u]\!]$, $X \dot{\cap} Y > u \in G$ and $w = u$.

Otherwise $t = f(t_1, \ldots, t_n)$, $u = f(u_1, \ldots, u_n)$ (for some function symbol f of arity $n \geq 0$) and the considered derivations are $X \Rightarrow f(\ldots) \Rightarrow^* t$ and $Y \Rightarrow f(\ldots) \Rightarrow^* u$. Grammar G contains a rule $X \dot{\cap} Y > f(X_1 \circ Y_1, \ldots, X_n \circ Y_n)$ and $G(\Phi)$ contains $X \dot{\cap} Y > f(Z_1, \ldots, Z_n)$, where $X_i \circ Y_i = Z_i$ unless $X_i \circ Y_i$ is a parameter. For each $i = 1, \ldots, n$ we have three cases.

[30] According to our assumptions on base sets, $[\![s_1]\!] \cap [\![f(\top, \ldots, \top)]\!] = \emptyset$. If $s_2 = f(\ldots)$ then no rule corresponding to $X > s$, $Y > t$ should appear in $G_1 \dot{\cap} G_2$.

1. $X_i \circ Y_i$ is the variable $X_i \dot{\cap} Y_i$. $X_i \Rightarrow^*_{G_1(\Phi)} t_i$ and $Y_i \Rightarrow^*_{G_2(\Phi)} u_i$. Clearly, $\max(|t_i|, |u_i|) < \max(|t|, |u|)$. By the inductive assumption there exists a term w_i such that $Z_i = X_i \dot{\cap} Y_i \Rightarrow^*_{G(\Phi)} w_i$ and $[\![t_i]\!] \cap [\![u_i]\!] \subseteq [\![w_i]\!]$.

2. $X_i \circ Y_i$ is a parameter from G_1. Then $Z_i \Rightarrow^* t_i$ both in $G_1(\Phi)$ and $G(\Phi)$.

3. $X_i \circ Y_i$ is a variable from G_1 or G_2. Thus $Z_i \Rightarrow^* t_i$ both in $G_1(\Phi)$ and $G(\Phi)$, or $Z_i \Rightarrow^* u_i$ both in $G_2(\Phi)$ and $G(\Phi)$.

This shows that for $i = 1, \ldots, n$ there exists a w_i such that $Z_i \Rightarrow^*_{G(\Phi)} w_i$ and $[\![t_i]\!] \cap [\![u_i]\!] \subseteq [\![w_i]\!]$. Hence $X \dot{\cap} Y \Rightarrow^*_{G(\Phi)} f(w_1, \ldots, w_n)$ and $[\![t]\!] \cap [\![u]\!] \subseteq [\![f(w_1, \ldots, w_n)]\!]$. □

Example 4.9 Grammar G_1 describes parametric non-empty lists and grammar G_2 specifies lists of natural numbers:

$$\begin{array}{llll} G_1: & \mathit{NEList} > cons(\alpha, \mathit{List}) & \qquad G_2: & \mathit{ListN} > nil \\ & \mathit{List} > nil & & \mathit{ListN} > cons(Nat, \mathit{ListN}) \\ & \mathit{List} > cons(\alpha, \mathit{List}) & & \end{array}$$

Computing $\mathit{NEList} \dot{\cap} \mathit{ListN}$ gives a rule:

$$\mathit{NEList} \dot{\cap} \mathit{ListN} > cons(Nat, \mathit{List} \dot{\cap} \mathit{ListN})$$

The new variable $\mathit{List} \dot{\cap} \mathit{ListN}$ is defined by the following rules:

$$\begin{array}{l} \mathit{List} \dot{\cap} \mathit{ListN} > nil \\ \mathit{List} \dot{\cap} \mathit{ListN} > cons(Nat, \mathit{List} \dot{\cap} \mathit{ListN}) \end{array}$$

Thus we obtained a non-empty list of natural numbers as a result.

4.4.3 Union

Let G_1 and G_2 be PED grammars. We assume without loss of generality that they have no common variables, but they may have common parameters. We define an operation $\dot{\cup}$ on such grammars; the result is a PED grammar G, denoted $G_1 \dot{\cup} G_2$. The variables of G include the variables of G_1, the variables of G_2 and new variables corresponding to pairs (X, Y) where X is a variable of G_1 and Y is a variable of G_2. The latter will be denoted $X \dot{\cup} Y$.

Now G consists of the rules of $G_1 \cup G_2$ and, for each $X \dot{\cup} Y$, of the rules constructed as follows. Let $R = \{\, t \mid X > t \in G_1 \text{ or } Y > t \in G_2 \,\}$. If $\top \in R$ then G contains $X \dot{\cup} Y > \top$, otherwise:

1. If $f(s_1, \ldots, s_n) \in R$ $(n > 0)$ and no other $f(t_1, \ldots, t_n)$ is in R then G contains $X \dot{\cup} Y > f(s_1, \ldots, s_n)$.

2. For each pair $f(s_1, \ldots, s_n)$, $f(t_1, \ldots, t_n)$ of distinct elements of R ($n > 0$),[31] G contains $X \dot{\cup} Y > f(s_1 \circ t_1, \ldots, s_n \circ t_n)$, where each $s_i \circ t_i$ is
   - $s_i \dot{\cup} t_i$, if s_i, t_i are variables,
   - s_i, if $s_i = t_i$ and is a parameter,
   - a new variable V otherwise. In this case also rule $V > \top$ is in G.
3. $X \dot{\cup} Y > s$ is in G for each $s \in R$ such that s is a constant or a base symbol and $[\![s]\!] \not\subseteq [\![t]\!]$ for any base symbol $t \in R$, $t \neq s$.

The result of the construction is a PED grammar. Its parameters (if any) may only originate from G_1 and G_2. The construction is similar to that for discriminative term grammars. The union involving parameters is approximated by $\top$ unless both arguments are the same parameter. This is because we want the construction to approximate the union for all parameter valuations.

Proposition 4.10 For every parameter valuation Φ such that $G_1(\Phi)$ and $G_2(\Phi)$ are parameterless grammars we have

$$[\![X]\!]_{G_1(\Phi)} \cup [\![Y]\!]_{G_2(\Phi)} \subseteq [\![X \dot{\cup} Y]\!]_{G_1 \dot{\cup} G_2(\Phi)}$$

for all variables X in G_1 and Y in G_2.

PROOF Denote $[\![X \dot{\cup} Y]\!]_{G_1 \dot{\cup} G_2(\Phi)}$ by R. It is sufficient to show that if $X \Rightarrow^*_{G_1(\Phi)} s$ or $Y \Rightarrow^*_{G_2(\Phi)} s$, where s is ground, then $[\![s]\!] \subseteq R$. We show this by induction on the derivation length. We can assume that the same renaming of the variables of Φ has been used in constructing $G_1(\Phi), G_2(\Phi)$ and $(G_1 \dot{\cup} G_2)(\Phi)$.

Assume that $V \Rightarrow_H s_0 \Rightarrow^*_H s$, where $V = X$, $H = G_1(\Phi)$ or $V = Y$, $H = G_2(\Phi)$. We have two cases.

- s_0 is a constant or base symbol (so $s_0 = s$). There is a rule $X \dot{\cup} Y > s'$ in $G_1 \dot{\cup} G_2$ such that $[\![s_0]\!] \subseteq [\![s']\!]$. We have $[\![s]\!] \subseteq [\![s']\!] \subseteq R$.
- $s_0 = f(X_1, \ldots, X_n)$ (where $n > 0$), $s = f(u_1, \ldots, u_n)$ and $X_i \Rightarrow^*_H u_i$ for each $i = 1, \ldots, n$. Grammar $G_1 \dot{\cup} G_2$ contains a rule $X \dot{\cup} Y > \top$ or $X \dot{\cup} Y > f(Y_1, \ldots, Y_n)$. In the first case the result is immediate. In the second case the rule have been introduced by clause 1 or clause 2 of the definition of $G_1 \dot{\cup} G_2$.

  In the case of clause 1, $(G_1 \dot{\cup} G_2)(\Phi)$ contains $X \dot{\cup} Y > f(X_1, \ldots, X_n)$ and $Y_i = X_i$ whenever Y_i is a variable. $X \dot{\cup} Y \Rightarrow f(X_1, \ldots, X_n) \Rightarrow^* s$ is a derivation of $(G_1 \dot{\cup} G_2)(\Phi)$, as $H \subseteq (G_1 \dot{\cup} G_2)(\Phi)$, Hence $[\![s]\!] \subseteq R$.

[31] Notice that for a given f at most two such elements exist.

In the case of clause 2, each Y_i is $s_i \circ t_i$. If $s_i \circ t_i$ is $s_i \dot{\cup} t_i$ then s_i, t_i are variables, one of them is X_i and by the inductive assumption $[\![u_i]\!] \subseteq [\![s_i \dot{\cup} t_i]\!]_{(G_1 \dot{\cup} G_2)(\Phi)}$, as $X_i \Rightarrow_H^* u_i$. If $s_i \circ t_i$ is a parameter then $s_i \dot{\cup} t_i = s_i = t_i$. In $(G_1 \dot{\cup} G_2)(\Phi)$ this parameter is replaced by X_i. Notice that in this grammar $X_i \Rightarrow^* u_i$. The last possibility is that $s_i \circ t_i$ is a variable W and $W > \top$ is in $G_1 \dot{\cup} G_2$.

So $(G_1 \dot{\cup} G_2)(\Phi)$ contains a rule $X \dot{\cup} Y > f(r_1, \ldots, r_n)$ where $r_i = s_i \circ t_i$ and $[\![u_i]\!] \subseteq [\![r_i]\!]_{(G_1 \dot{\cup} G_2)(\Phi)}$, for $i = 1, \ldots, n$. Hence
$[\![s]\!] = [\![f(u_1, \ldots, u_n)]\!] \subseteq [\![f(r_1, \ldots, r_n)]\!]_{(G_1 \dot{\cup} G_2)(\Phi)} \subseteq [\![X \dot{\cup} Y]\!]_{(G_1 \dot{\cup} G_2)(\Phi)}$.

□

The requirement that G_1, G_2 have no common variables is inessential when $G_1 = G_2$. This holds both for $\dot{\cap}$ and $\dot{\cup}$ and follows from the proofs of the last two propositions.

Example 4.11 Consider the grammars from Example 4.9, G_1 specifying parametric non-empty lists and G_2 describing lists of natural numbers.

$$G_1: \quad \begin{array}{l} NEList > cons(\alpha, List) \\ List > nil \\ List > cons(\alpha, List) \end{array} \qquad G_2: \quad \begin{array}{l} ListN > nil \\ ListN > cons(Nat, ListN) \end{array}$$

The rules defining $NEList \dot{\cup} ListN$ are

$$\begin{array}{l} NEList \dot{\cup} ListN > nil \\ NEList \dot{\cup} ListN > cons(V, List \dot{\cup} ListN) \\ V > \top \end{array}$$

where V is a new variable. There are similar rules for $List \dot{\cup} ListN$:

$$\begin{array}{l} List \dot{\cup} ListN > nil \\ List \dot{\cup} ListN > cons(W, List \dot{\cup} ListN) \\ W > \top \end{array}$$

4.4.4 Generalized projection for parametric sets

Let $\langle Y, G \rangle$ be a set descriptor, where G is a PED grammar, and t be a term. We are going to construct a PED grammar defining (a superset of) $t^{-X}([\![Y]\!]_{G(\Phi)})$.

We first construct a mapping $\xi(t, G, Y)$ assigning to each subterm occurrence u in t a variable or a parameter V_u. V_u occurs in G or is a new variable Any. Mapping $\xi(t, G, Y)$ has the following properties:

1. V_t is Y.

2. If $u = f(u_1, \ldots, u_n)$ $(n \geq 0)$ and V_u is a parameter or Any then $V_{u_1} = \ldots = V_{u_n} = Any$.

3. If $u = f(u_1, \ldots, u_n)$ $(n \geq 0)$ and V_u is a variable of G then
   - $V_u > f(V_{u_1}, \ldots, V_{u_n}) \in G$, or
   - $V_u > b \in G$, where b is a base symbol, $u \in [\![b]\!]$ and $V_{u_1} = \ldots = V_{u_n} = \mathit{Any}$. (Notice that if $n \neq 0$ then $b = \top$.)

If $\xi(t, G, Y)$ exists then it is unique, because the grammar is discriminative. $\xi(t, G, Y)$ can be constructed by an obvious algorithm similar to that described in Section 2.1.2.

Proposition 4.12 Let G be a PED grammar and $G' = G \cup \{\mathit{Any} > \top\}$. Let t be a term and $X^1, \ldots, X^k$ $(k \geq 0)$ be the occurrences of a variable X in t. If $\xi(t, G, Y)$ exists then

$$t^{-X}([\![Y]\!]_{G(\Phi)}) \subseteq \bigcap_i [\![V_{X^i}]\!]_{G'(\Phi)}$$

for any parameter valuation Φ such that $G(\Phi)$ is parameterless.

If $\xi(t, G, Y)$ does not exist or $\bigcap_i [\![V_{Z^i}]\!]_{G'(\Phi)} = \emptyset$ for some variable Z of t then $t^{-X}([\![Y]\!]_{G(\Phi)}) = \emptyset$.

PROOF Consider a Φ as above. Let $H = G(\Phi)$ and $H' = G'(\Phi)$.

We begin with showing the following property. Let $c \,[\!]\, u$ be a constrained term and V_u be some variable or parameter of G'. If $c \,[\!]\, u\theta \in [\![V_u]\!]_{H'}$ then V_u satisfies the conditions for $\xi(t, G, Y)$ above (for some $n, V_{u_1}, \ldots, V_{u_n}$).

Assume that u is not a variable (otherwise the conditions hold vacuously) and that $c \,[\!]\, u\theta \in [\![V_u]\!]_{H'}$. For V_u being a parameter or Any the conditions trivially hold. Let V_u be a variable of G. We have $V_u \Rightarrow^*_{H'} s$ and $c \,[\!]\, u\theta \in [\![s]\!]$, where $s = f(s_1, \ldots, s_n)$, $u = f(u_1, \ldots, u_n)$ and $V_u \Rightarrow_{H'} f(X_1, \ldots, X_n)$, or s is a base symbol and $V_u \Rightarrow_{H'} s$. Then a rule $V_u > f(X_1, \ldots, X_n)$, respectively $V_u > s$ exists in G; the rule has the required properties.

Now we show that if $c \,[\!]\, t\theta \in [\![Y]\!]_H$ then mapping $\xi(t, G, Y)$ exists and for any subterm u of t, $c \,[\!]\, u\theta \in [\![V_u]\!]_{H'}$. The latter is equivalent to existence of a ground term s such that $V_u \Rightarrow^*_{H'} s$ and $c \,[\!]\, u\theta \in [\![s]\!]$.

The proof is by induction. Let u be a subterm of t and

$$U = \{\, u' \mid u \text{ is a proper subterm of } u',\ u' \text{ is a subterm of } t \,\}.$$

Assume that the required mapping exists on U. (So $c \,[\!]\, u'\theta \in [\![V_{u'}]\!]_{H'}$ for each $u' \in U$ and the conditions for $\xi(t, G, Y)$ are satisfied.) We show that such a mapping exists for $U \cup \{u\}$. It is sufficient to show that $c \,[\!]\, u\theta \in [\![V_u]\!]_{H'}$, then it follows that V_u satisfies the conditions for $\xi(t, G, Y)$ from the property discussed above.

If $u = t$ then $c \,[\!]\, u\theta \in [\![V_u]\!]_{H'}$ obviously holds. Otherwise there exists a subterm $u' = f(u_1, \ldots, u_n)$ of t such that $u = u_i$ for some i, and a ground term s' such that $V_{u'} \Rightarrow^*_{H'} s'$ and $c \,[\!]\, u'\theta \in [\![s']\!]$.

If $V_{u'}$ is a parameter or *Any* then V_u is *Any* and $c\,[\,]\,u\theta \in [\![Any]\!]_{H'}$. The same reasoning is applicable when $V_{u'}$ is a variable of G and $V_{u'} > b \in G$, as then $b = \top$ and $V_u = Any$.

It remains to consider the case of $V_{u'}$ being a variable of G such that $V_{u'} \Rightarrow_{H'} f(V_{u_1}, \ldots, V_{u_n}) \Rightarrow^*_{H'} s' = f(s_1, \ldots, s_n)$. So $V_{u'} > f(V_{u_1}, \ldots, V_{u_n}) \in G$ and $V_{u_i} \Rightarrow^*_{H'} s_i$. From $c\,[\,]\,f(u_1, \ldots, u_n)\theta \in [\![s']\!]$ it follows that $c\,[\,]\,u\theta \in [\![s_i]\!] \subseteq [\![V_u]\!]_{H'}$. This completes the inductive proof.

Thus if $c\,[\,]\,t\theta \in [\![Y]\!]_H$ then $c\,[\,]\,X^i\theta \in [\![V_{X^i}]\!]_{H'}$ for any occurrence X^i of X in t. Hence

$$c\,[\,]\,X^i\theta \in \bigcap_i [\![V_{X^i}]\!]_{G'(\Phi)} \quad \text{and thus} \quad t^{-X}([\![Y]\!]_{G(\Phi)}) \subseteq \bigcap_i [\![V_{X^i}]\!]_{G'(\Phi)}.$$

Notice that if $\xi(t, G, Y)$ does not exist or the intersection above is empty then $c\,[\,]\,t\theta \notin [\![Y]\!]_H$ for any c, θ, and $t^{-Z}[\![Y]\!]_H = \emptyset$ for any variable Z. □

The proposition suggests the following algorithm to compute a set descriptor $t^{-X}(\langle Y, G\rangle)$ giving an approximation of the set $t^{-X}([\![Y]\!]_{G(\Phi)})$.

1. Compute $\xi(t, G, Y)$.
2. For each variable Z with the occurrences $Z^1, \ldots, Z^k$ in t, apply the intersection algorithm for PED grammars (Section 4.4.2) to compute (an approximation of) $\bigcap_i [\![V_{Z^i}]\!]_{G'(\Phi)}$. This results in a grammar $G_Z = G' \dot{\cap} \ldots \dot{\cap} G'$ and a variable $Z' = Z^1 \dot{\cap} \ldots \dot{\cap} Z^k$ such that $\bigcap_i [\![V_{Z^i}]\!]_{G'(\Phi)} \subseteq [\![Z']\!]_{G_Z(\Phi)}$.
3. If $\xi(t, G, Y)$ does not exist or some Z' is nullable in G_Z then return $t^{-X}(\langle Y, G\rangle) = \langle V, \emptyset\rangle$ as the result (because $t^{-Z}(\langle Y, G\rangle) = \emptyset$, for any Z).
4. Otherwise return $t^{-X}(\langle Y, G\rangle) = \langle X', G_X\rangle$.

From the last proposition and the appropriate property of the grammar intersection operation it follows that if the algorithm produces $t^{-X}(\langle Y, G\rangle) = \langle V, H\rangle$ then $t^{-X}([\![Y]\!]_{G(\Phi)}) \subseteq [\![V]\!]_{H(\Phi)}$.

4.4.5 Inclusion checking for parametric sets

The algorithms for checking inclusion of the sets defined by discriminative term grammars can be generalized to extended parametric grammars.

The problem is stated as follows. Let G_1 and G_2 be PED grammars. Let X be a variable in G_1 and let Y be a variable in G_2. We want to check whether $[\![X]\!]_{G_1(\Phi)} \subseteq [\![Y]\!]_{G_2(\Phi)}$ for any valuation Φ such that $G_1(\Phi), G_2(\Phi)$ are parameterless. We will denote this fact by $\langle X, G_1\rangle \sqsubseteq \langle Y, G_2\rangle$ (often abbreviated to $X \sqsubseteq Y$).

We begin with introducing some notions. By $C(X, Y)$ we mean the least set of pairs (of variables or parameters) such that

- $(X, Y) \in C(X, Y)$ and
- if $(X', Y') \in C(X, Y)$, $X' > f(X_1, \ldots, X_n) \in G_1$ and $Y' > f(Y_1, \ldots, Y_n) \in G_2$ then $(X_1, Y_1), \ldots, (X_n, Y_n) \in C(X, Y)$.

An algorithm checking whether $X \sqsubseteq Y$ follows immediately from the following property and from finiteness of $C(X, Y)$.

Proposition 4.13 Let G_1, G_2 be PED grammars and X, Y be variables of, respectively, G_1, G_2. Assume that for each pair $(V, W) \in C(X, Y)$

- if V is a parameter then $V = W$ or rule $W > \top$ is in G_2,
- if V is a variable then
  - for each rule $V > f(V_1, \ldots, V_n) \in G_1$ $(n \geq 1)$ there exists a rule $W > f(\ldots) \in G_2$ or $W > \top \in G_2$, and
  - for each rule $V > c \in G_1$, where c is a constant or base symbol, there exists a $W > c' \in G_2$ such that $[\![c]\!] \subseteq [\![c']\!]$.

Then $X \sqsubseteq Y$.

The reverse implication holds provided G_1 does not have nullable symbols, $[\![c]\!] \neq \emptyset$ for each base symbol c, and if $[\![c]\!] \subseteq [\![W]\!]_{G_2(\Phi)}$ for some Φ, base symbol or constant c and variable W of G_2 then G_2 contains a rule $W > c'$ where $[\![c]\!] \subseteq [\![c]\!]'$. Intuitively, the last condition means that no set $[\![c]\!]$ is described by G_2 by more than one rule.

PROOF Assume that the conditions are satisfied. For any $(V, W) \in C(X, Y)$ and any derivation $V \Rightarrow^*_{G_1} t$, where t is a variable-free term, there exists a derivation $W \Rightarrow^*_{G_2} u$ such that $[\![t]\!]_{G_1(\Phi)} \subseteq [\![u]\!]_{G_2(\Phi)}$ for any Φ. This can be shown by induction on the structure of t. If a constrained term w is in $[\![V]\!]_{G_1(\Phi)}$ then $w \in [\![t]\!]_{G_1(\Phi)}$ for some t as above. Hence $w \in [\![W]\!]_{G_2(\Phi)}$, which completes the "if" part of the proof.

Assume that the conditions are not satisfied, for some pair $(V, W) \in C(X, Y)$. We show that for some parameter valuation Φ there exists a constrained term t such that $t \in [\![V]\!]_{G_1(\Phi)}$ and $t \notin [\![W]\!]_{G_2(\Phi)}$. We enumerate the possible cases, in each of them such Φ and t obviously exist.

If V is a parameter then W is a different parameter or a variable such that $[\![W]\!]_{G_2(\Phi)} \neq [\![\top]\!]$. For V being a variable we have two cases. $V > f(\ldots) \in G_1$ and no $W > f(\ldots)$ is in G_2, or $V > c \in G_1$ and for each $W > c' \in G_2$ $[\![c]\!] \not\subseteq [\![c]\!]'$, hence $[\![c]\!] \cap [\![c]\!]' = \emptyset$ (by our restrictions on base sets).

Now it is easy to construct a $u \in [\![X]\!]_{G_1(\Phi)}$ such that $u \notin [\![Y]\!]_{G_2(\Phi)}$ by induction on the definition of $C(X, Y)$ (on the number of applications of the second rule of the definition of $C(X, Y)$ needed to show that $(V, W) \in C(X, Y)$). □

We illustrate the check by a simple example.

Example 4.14

$$
\begin{array}{llll}
G_1: & Y > cons(\alpha, Z) & \qquad G_2: & X > nil \\
 & Z > nil & & X > cons(\alpha, X) \\
 & Z > cons(\alpha, Y) & &
\end{array}
$$

We want to check the inclusion

$$[\![Y]\!]_{G_1(\Phi)} \subseteq [\![X]\!]_{G_2(\Phi)}$$

for arbitrary parameter valuation Φ such that $G_1(\Phi)$ and $G_2(\Phi)$ are parameterless. For each pair of $C(Y, X)$ the conditions from the proposition are to be checked. $C(Y, X)$ contains $(Y, X), (\alpha, \alpha), (Z, X)$.

Consider (Y, X). For the rule $Y > cons(\alpha, Z) \in G_1$ there exists $X > cons(\alpha, X) \in G_2$. For (α, α) the check is immediate. For (Z, X), the following pairs of rules are found to satisfy the conditions.

$$
\begin{array}{rl}
Z > nil \in G_1, & X > nil \in G_2 \\
Z > cons(\alpha, Y) \in G_1, & X > cons(\alpha, X) \in G_2
\end{array}
$$

So the check is successfully completed.

4.4.6 Set matching

In our approach, a set of allowed calls of a polymorphic procedure will be specified by a set descriptor $\langle Y, G\rangle$ where G is a PED grammar. A particular call t is allowed if there exists a valuation of parameters Φ such that $t \in [\![Y]\!]_{G(\Phi)}$.

A set of actual calls may be described by another set descriptor $\langle X, H\rangle$, where H is a PED grammar which has no parameter common with G.

We want to be sure that all actual calls are allowed. As the specifications are parametric we have to refer to their instances. The question is then, whether for any valuation Ψ of the parameters of H there exists a parameter valuation Φ for G such that $[\![X]\!]_{H(\Psi)} \subseteq [\![Y]\!]_{G(\Phi)}$. Additionally we are interested in obtaining a possibly small set $[\![Y]\!]_{G(\Phi)}$. We will call this a *set matching* problem.

A solution can be obtained by a modification of the set inclusion algorithm discussed above. In this extension the parameters of H are handled as constants while searching for such bindings of the parameters of G that the inclusion holds.

For a given X, H and Y, G the matching algorithm constructs a parameter valuation Φ (possibly containing parameters from H) such that for any Ψ for which $H(\Psi)$ is parameterless

$$[\![X]\!]_{H(\Psi)} \subseteq [\![Y]\!]_{G(\Phi)(\Psi)}.$$

(This is expressed as $\langle X, H\rangle \sqsubseteq \langle Y, G(\Phi)\rangle$ in the notation of the previous section).

To describe matching we recall how the inclusion algorithm works. Applied to X in H and Y in G, it checks the conditions of Proposition 4.13 for each pair $(s,t) \in C(X,Y)$. The difference with the matching algorithm is in the treatment of a (s,t) where t is a parameter (of G). In such case the inclusion algorithm answers "no". In matching we want to instantiate the parameters of G so that inclusion holds. So in this case the matching algorithm binds the parameter t to s (which is a variable or a parameter). Notice that several different bindings for t may be produced since t may appear in several pairs in C.

As $C(X,Y)$ is finite, the checking terminates with failure or success. In the latter case a set of bindings is produced. From these bindings we now construct a parameter valuation Φ. This is done separately for each parameter α. Let $\{\alpha \mapsto s_1, \ldots, \alpha \mapsto s_k\}$ $(k \geq 1)$ be the set of bindings for α produced by the algorithm. The valuation $\Phi(\alpha)$ is constructed by considering the following cases:

- If $k = 1$ then $\Phi(\alpha) = \langle s_1, H\rangle$.
- If $k > 1$ and all s_i are variables of H, then $\Phi(\alpha) = \langle s_1 \dot\cup \ldots \dot\cup s_k, H \dot\cup \ldots \dot\cup H\rangle$.
- Otherwise $k > 1$ and some s_i is a parameter. Then $\Phi(\alpha) = \langle X, \{X > \top\}\rangle$ where X is a new variable.

Let $\alpha_1, \ldots, \alpha_n$ be all the parameters of G that appear in $C(X,Y)$. Applying the above stated rules to each of them we obtain $\Phi = \{\, \alpha_1 \mapsto \Phi(\alpha_1), \ldots, \alpha_n \mapsto \Phi(\alpha_n)\,\}$.

This completes the description of the matching algorithm. It remains to show that if it succeeds then $[\![X]\!]_{H(\Psi)} \subseteq [\![Y]\!]_{G(\Phi)(\Psi)}$, for any parameter valuation Ψ. Assume that $\Phi(\alpha_i) = \langle X_i, G_i\rangle$ (for $i = 1, \ldots, n$) and that X_i was renamed into X_i' while constructing $G(\Phi)$. We apply the inclusion checking algorithm to X, H and $Y, G(\Phi)$ and compare its actions with those of the matching algorithm for X, H and Y, G.

Whenever the matching algorithm produces a pair (s,t) of two variables, the same pair is produced by the inclusion checking algorithm. Whenever the former produces an (s, α_i) then the second produces (s, X_i'). Grammar G_i has been constructed in such a way that $[\![s]\!]_{H(\Psi)} \subseteq [\![X_i]\!]_{G_i(\Psi)}$. As $[\![X_i]\!]_{G_i(\Psi)} = [\![X_i']\!]_{G(\Phi)(\Psi)}$ we have $[\![s]\!]_{H(\Psi)} \subseteq [\![X_i']\!]_{G(\Phi)(\Psi)}$. Hence for each pair (s,t) produced by the inclusion checking algorithm, $[\![s]\!]_{H(\Psi)} \subseteq [\![t]\!]_{G(\Phi)(\Psi)}$. This completes the proof.

Example 4.15 This example illustrates set matching. The parametric grammars H and G specify different variants of lists with elements being triples.

$$
\begin{array}{llcll}
H: & L > nil & \qquad & G: & S > nil \\
 & L > cons(T, L) & & & S > cons(E, S) \\
 & T > t(B, N, \gamma) & & & E > t(\alpha, \alpha, \beta) \\
 & B > tt & & & \\
 & B > ff & & & \\
 & N > nat & & &
\end{array}
$$

We want to match $\langle L, H\rangle$ and $\langle S, G\rangle$. We obtain $C(L, S) = \{(L, S), (T, E), (B, \alpha), (N, \alpha), (\gamma, \beta)\}$. The checks succeed with parameter bindings

$$\{\, \alpha \mapsto B,\ \alpha \mapsto N,\ \beta \mapsto \gamma \,\}.$$

The result is the parameter valuation

$$\Phi = \{\, \alpha \mapsto \langle B \dot{\cup} N, H \dot{\cup} H\rangle,\ \ \beta \mapsto \langle \gamma, H\rangle \,\}$$

5 Locating Program Errors with Parametric Specifications

The call-success semantics discussed in Section 3 describes a program (together with its set of initial goals) by the set of calls and the set of successes. So the information about which successes correspond to which calls is lost. A more precise semantics can be given by replacing the set of successes by the set of pairs of a call and a corresponding success.

The formalism of discriminative grammars does not provide useful approximations of such semantics. If pairs $(call_1, success_1), (call_2, success_2)$ are in such approximation then $(call_1, success_2), (call_2, success_1)$ are there too. Useful approximations can be however provided by parametric discriminative grammars. With such a grammar one can specify a family of specifications. Correctness w.r.t. such a family means the following. Whenever a call is correct w.r.t. some specification from the family then any its success is correct w.r.t. this specification. Additionally, each call is correct w.r.t. some of the specifications.

In this section we address the question of partial correctness of programs w.r.t. parametric specifications. First we state formally the problem and show that it can be re-formulated in terms of parametric set constraints. We show how to employ the constraints to check whether a program is correct w.r.t. a given specification and how to compute a specification for which the program is correct. Then we formalize the notion of error and discuss how the correctness checking procedure locates errors.

5.1 Parametric specifications and program correctness

By a parametric specification we mean a set of specifications.[32] We are interested in specifications given by parametric grammars, this is however insignificant for the purposes of this section. Here we define the notion of correctness for such specifications and prove a sufficient condition for such correctness.

Definition 5.1 Let *Spec* be a parametric specification. A call $c \,[]\, A$ in an LD-derivation is *correct* w.r.t. *Spec* if there exists some $(Pre, Post) \in Spec$ such that $c \,[]\, A \in Pre$. A success $c' \,[]\, A\theta$ corresponding to a call $c \,[]\, A$ is *correct* w.r.t. *Spec* if $c' \,[]\, A\theta \in Post$, for any $(Pre, Post) \in Spec$ such that $c \,[]\, A \in Pre$.

A program P with a set of initial goals $\mathcal{G}$ is *correct* w.r.t. *Spec* iff in any LD-derivation of P starting from a goal from $\mathcal{G}$ all the calls and successes are correct w.r.t. *Spec*. A program P is *correct* w.r.t. *Spec* iff P with the set of initial goals $\bigcup\{\, Pre \mid (Pre, Post) \in Spec \,\}$ is correct w.r.t. *Spec*. □

We impose following restrictions on parametric specifications. If $(Pre, Post)$ is a member of such a specification then $Pre, Post$ are closed under instantiation and $Pre \supseteq Post$.[33] The correctness criterion from Proposition 3.1 can now be generalized.

Theorem 5.1 *Let P be a CLP program, $\mathcal{G}$ a set of atomic initial goals and Spec be a parametric specification. Let each $(Pre, Post) \in Spec$ respect constraints. A sufficient condition for P with $\mathcal{G}$ being correct w.r.t. Spec is:*

1. *For each clause $H \leftarrow B_1, \ldots, B_n$ and any $(Pre_0, Post_0) \in Spec$ there exist $(Pre_1, Post_1), \ldots, (Pre_n, Post_n) \in Spec$ such that for $j = 0, \ldots, n$, any substitution θ and constraint c*

$$\begin{array}{l} \textit{if} \;\; c \,[]\, H\theta \in Pre_0, \;\; c \,[]\, B_1\theta \in Post_1, \;\; \ldots, \;\; c \,[]\, B_j\theta \in Post_j \\ \textit{then} \\ \quad c \,[]\, B_{j+1}\theta \in Pre_{j+1}, \; \textit{if } j < n \\ \quad c \,[]\, H\theta \in Post_0, \; \textit{if } j = n \end{array}$$

2. *Each element of $\mathcal{G}$ is in some Pre, such that $(Pre, Post) \in Spec$.*

(As explained in Section 3.1, the restriction to atomic initial goals is not substantial).

PROOF Consider the i-th goal Q_i of an LD-derivation starting from a goal $Q_0 \in \mathcal{G}$. We show that the call and the successes occurring in Q_i are correct.

[32] Remember that a (non parametric) specification is a pair of sets of (constrained) atoms.

[33] The latter condition is not essential. To abandon it, it is sufficient to replace each $Post_l$ in theorem 5.1 by $Pre_l \cap Post_l$.

The proof is by induction on i. If $i = 0$ then Q_i contains no successes and the call in Q_i is obviously correct.

Let $i > 0$. Consider the call in Q_i. (The case of the goal containing no call is considered later on). Q_i is of the form $c\,[\!]\,(B_{j+1}, \ldots, B_n, \vec{A})\tau$, where $j < n$, for some clause $H \leftarrow B_1, \ldots, B_n$ of P, and the derivation is

$$
\begin{array}{lcl}
 & & \cdots \\
Q_{i_0} & = & c_0\,[\!]\,A, \vec{A} \\
Q_{i_1} & = & c_0\,[\!]\,(B_1, \ldots, B_n, \vec{A})\theta_0 \\
 & & \cdots \\
Q_{i_2} & = & c_1\,[\!]\,(B_2, \ldots, B_n, \vec{A})\theta_0\theta_1 \\
 & & \cdots \\
 & & \cdots \\
Q_{i_{j+1}} & = & c_j\,[\!]\,(B_{j+1}, \ldots, B_n, \vec{A})\theta_0 \cdots \theta_j \\
 & & \cdots
\end{array}
$$

where $i_{j+1} = i$, $\theta_0 \cdots \theta_j = \tau$ and the call $c_{l-1}\,[\!]\,B_l\theta_0 \cdots \theta_{l-1}$ from a goal Q_{i_l} succeeds in the goal $Q_{i_{l+1}}$ (for $l = 1, \ldots, j$). The calls from $Q_{i_0}, \ldots, Q_{i_j}$ are correct, by the inductive assumption. So there exist $(Pre_0, Post_0), \ldots, (Pre_j, Post_j) \in Spec$ such that $c_0\,[\!]\,A \in Pre_0$ and $c_{l-1}\,[\!]\,B_l\theta_0 \cdots \theta_{l-1} \in Pre_l$ for $l = 1, \ldots, j$.

Now we show that the successes of these calls are correct. This means $c_l\,[\!]\,B_l\theta_0 \cdots \theta_l \in Post_l$ for $l = 1, \ldots, j$ and for any $(Pre_0, Post_0), \ldots, (Pre_j, Post_j)$ as above. Notice that this includes the $(Pre_1, Post_1), \ldots, (Pre_j, Post_j)$ from condition 1 of the Theorem.

The successes from $Q_{i_2}, \ldots, Q_{i_j}$ are correct by the inductive assumption. Also the success from $Q_{i_{j+1}}$ of $c_{j-1}\,[\!]\,B_j\theta_0 \cdots \theta_{j-1}$ is correct. To show this remove (the instances of) $B_{j+1}, \ldots, B_n, \vec{A}$ from the goals of the derivation $Q_{i_j}, \ldots, Q_{i_{j+1}}$, obtaining a derivation to which the inductive assumption applies. (The derivation is shorter than i and starts from an atomic goal). Other procedure calls (from goals between Q_{i_j} and $Q_{i_{j+1}}$) may succeed in $Q_{i_{j+1}}$. These successes are correct by the same reasoning.

As all $Pre_l, Post_l$ are instance closed, we have $c_j\,[\!]\,A\tau \in Pre_0$ and $c_j\,[\!]\,B_l\tau \in Post_l$ for $l = 1, \ldots, j$. Moreover, $A\tau = H\tau$, as $A\theta_0 = H\theta_0$. From condition 1 of the Theorem it follows that the call $c_j\,[\!]\,B_{j+1}\tau$ is correct.

It remains to consider the case when Q_i does not contain a call. So Q_i is of the form $c\,[\!]$ and the initial goal Q_0 succeeds in Q_i. Let $Q_0 = c_0\,[\!]\,A$. If A is a constraint then $i = 1$ and $Q_1 = c_0, A\,[\!]$. As the specification respects constraints, the success in Q_1 is in $Post$ whenever $Q_0 \in Pre$ and $(Pre, Post) \in Spec$. If A is not a constraint then we have a derivation as above, with $j = n \geq 0$ (so $Q_i = Q_{i_{n+1}}$), Q_{i_0} being the initial goal (so $i_0 = 0$) and $\vec{A}$ being empty. Reasoning as previously we obtain that the premises of the implication in the Theorem hold. Hence $c_n\,[\!]\,A\tau = c_n\,[\!]\,H\tau \in Post_0$.

As the choice of Pre_0 was arbitrary, this holds for any $(Pre_0, Post_0) \in Spec$ such that $c_0 \,[\!]\, A \in Pre_0$. So the success of $c_0 \,[\!]\, A$ is correct. $\square$

In our approach parametric specifications are given by parametric grammars. We assume that such a grammar G has two distinguished variables $Call, Success$. The specification is then

$$Spec = \{\, ([\![Call]\!]_{G(\Phi)}, [\![Success]\!]_{G(\Phi)}) \mid G(\Phi) \text{ is parameterless} \,\}.$$

We require that each specification $(Pre, Post) \in Spec$ respects constraints. Additionally we require that for each p such that $Success > p(\vec{Y}) \in G$, each parameter occurring in $\langle p(\vec{Y})\rangle_G$ occurs also in $\langle p(\vec{X})\rangle_G$, where $Call > p(\vec{X}) \in G$. Informally, this means that procedure successes may only depend on those parameters on which the corresponding procedure calls depend. This assures that to each Pre there corresponds exactly one $Post$ such that $(Pre, Post) \in Spec$.

Each grammar providing a specification can be seen as consisting of two parts. One is fixed for a given programming language and specifies the semantics of constraint predicates. The second is given by the user and describes the predicates defined by her program. Real CLP languages have built-in predicates, they can be treated by our method like constraint predicates.

5.2 Correctness checking

In this section we discuss checking the verification conditions of Theorem 5.1 with respect to a parametric specification given by a PED grammar. We generalize to such specifications the ideas of Section 3.2.

Similarly as in the parameterless case, each implication from Theorem 5.1 can be expressed by a system $F_j(C)$ of constraints consisting of

$$X \;>\; H^{-X}(Call_0) \,\cap\, \bigcap_{i=1}^{j} B_i{}^{-X}(Success_i) \tag{6}$$

(where $C = H \leftarrow B_1, \ldots, B_n$ is the considered clause, $0 \leq j \leq n$ and k ranges over the occurrences of X in the considered atom) for each variable X occurring in C, and of the

$$\begin{array}{ll} Call_{j+1} > B_{j+1} & \text{if } j < n, \\ Success_0 > H & \text{if } j = n. \end{array} \tag{7}$$

So for the condition 1 from the Theorem to hold it is sufficient that for each choice of $(Pre_0, Post_0) \in Spec$ there exist $(Pre_1, Post_1), \ldots, (Pre_n, Post_n) \in Spec$ such that each constraint system $F_j(C)$ $(j = 0, \ldots, n)$ has a model I in which $I(Call_i) = Pre_i$, $I(Success_i) = Post_i$, for $i = 0, \ldots, n$.

Now assume that the specification is given by a parametric grammar G. A particular $(Pre_0, Post_0)$ is given by a parameterless instance $G(\Phi)$ of G for some parameter valuation Φ: $Pre_0 = [\![Call]\!]_{G(\Phi)}$, $Post_0 = [\![Success]\!]_{G(\Phi)}$. For any such Φ we are looking for $\Phi_1, \ldots, \Phi_n$ describing, respectively, $(Pre_1, Post_1), \ldots, (Pre_n, Post_n)$. As the latter depend on Φ, the grammars of $\Phi_1, \ldots, \Phi_n$ may be parametric, with the parameters originating from grammar $\langle Call \rangle_G$. $\Phi_1, \ldots, \Phi_n$ should be chosen in such a way that for any Φ, each $F_j(C)$ has a model I in which

$$\begin{array}{ll} I(Call_0) = [\![Call]\!]_{G(\Phi)}, & I(Success_0) = [\![Success]\!]_{G(\Phi)}, \\ I(Call_i) = [\![Call]\!]_{G(\Phi_i)(\Phi)}, & I(Success_i) = [\![Success]\!]_{G(\Phi_i)(\Phi)} \end{array} \tag{8}$$

for $i = 1, \ldots, n$. This can be done in the following way.

Assume that $\Phi_1, \ldots, \Phi_j$ $(0 \leq j \leq n)$ have already been found. We show how to check the j-th implication of Theorem 5.1 and, if $j < n$, how to construct Φ_{j+1}. Let $G_0, \ldots, G_j$ be the grammars $G, G(\Phi_1), \ldots, G(\Phi_j)$ with the variables renamed apart such that

1. $Call, Success$ in $G(\Phi_i)$ are renamed into, respectively, $Call_i, Success_i$, for $i = 1, \ldots, j$, and $Call, Success$ in G into $Call_0, Success_0$,
2. no variable occurs in more than one grammar $G_0, \ldots, G_j$ and no variable from clause C occurs in $G_0, \ldots, G_j$.

Now $F_j(C) \cup G_0 \cup \ldots \cup G_j$ is to be converted into a discriminative grammar. For each variable X in the clause, constraint (6) is transformed as described in Section 3.2, by applying generalized projection and intersection operations from Section 4.4.

First for each $A^{-X}(Y)$ occurring in (6), by generalized projection we obtain $\langle X_A, G_A \rangle$ such that $A^{-X}([\![Y]\!]_{(G_0 \cup \ldots \cup G_j)(\Phi)}) \subseteq [\![X_A]\!]_{G_A(\Phi)}$. (Notice that Y is $Call_i$ or $Success_i$, thus $[\![Y]\!]_{(G_0 \cup \ldots \cup G_j)(\Phi)} = [\![Y]\!]_{G_i(\Phi)}$.) Then the intersection operation (followed by appropriate variable renaming) is applied to $\langle X_H, G_H \rangle, \langle X_{B_1}, G_{B_1} \rangle, \ldots, \langle X_{B_j}, G_{B_j} \rangle$, resulting in $\langle X, G_X \rangle$ such that

$$[\![X]\!]_{G_X(\Phi)} \supseteq H^{-X}([\![Call_0]\!]_{(G_0 \cup \ldots \cup G_j)(\Phi)}) \cap \bigcap_{i=1}^{j} B_i{}^{-X}([\![Success_i]\!]_{(G_0 \cup \ldots \cup G_j)(\Phi)})$$

In this way we construct G_X for each variable X of C. A renaming is applied so that the variables of the constructed grammars G_X are distinct and $Call_1, \ldots, Call_n, Success_1, \ldots, Success_n$ do not occur in any G_X. Let $G' = \bigcup_X G_X$. Notice that G' is discriminative and that, for any Φ, the least model of $(G' \cup G_0 \cup \ldots \cup G_j)(\Phi)$ is a model of $\mathcal{C} = F_j(C) - \{(7)\} \cup (G_0 \cup \ldots \cup G_j)(\Phi)$.

Also, the constraint (7) is converted into a discriminative grammar G'' in an obvious way, as described in Section 3.2. Each model of G'' is a model of (7), each model of (7) coincides with some model of G'' on the variables of (7).

Take an arbitrary Φ (such that $(G_0 \cup \ldots \cup G_j)(\Phi)$ is parameterless). Let I_Φ be the least model of $\mathcal{C} = F_j(C) - \{(7)\} \cup (G_0 \cup \ldots \cup G_j)(\Phi)$. We have $I_\Phi(X) \subseteq [\![X]\!]_{G'(\Phi)}$ for any variable X occurring in C, and $I_\Phi(Y) = [\![Y]\!]_{G_i(\Phi)}$ for Y being $Call_i$ or $Success_i$, $i = 1, \ldots, j$. Let us represent (7) as $Y > A$, where Y is $Call_{j+1}$ or $Success_0$ and A is, respectively, B_{j+1} or H. It holds that $[\![Y]\!]_{G'(\Phi) \cup G''} = [\![Y]\!]_{G'(\Phi) \cup \{(7)\}} = [\![A]\!]_{G'(\Phi)} \supseteq I_\Phi(A)$.

If $j = n$ then Y is $Success_0$, A is B_{j+1} and it remains to apply the inclusion algorithm to check whether

$$[\![Success_0]\!]_{G'(\Phi) \cup G''} \subseteq [\![Success]\!]_{G(\Phi)},$$

for any Φ. If yes then I_Φ is a model of (7) (as $[\![Success]\!]_{G(\Phi)} = I_\Phi(Success_0)$), hence a model of $F_j(C)$. It has the required properties, as (8) holds for $i = 1, \ldots, n$.

If $j < n$ then Y is $Call_{j+1}$, A is B_{j+1} and Φ_{j+1} has to be constructed. We have $I_\Phi(B_{j+1}) \subseteq [\![Call_{j+1}]\!]_{G'(\Phi) \cup G''}$, for any Φ. Now we apply the set matching operation of Section 4.4 to obtain Φ_{j+1} such that for any Φ

$$[\![Call_{j+1}]\!]_{G'(\Phi) \cup G''} \subseteq [\![Call]\!]_{G(\Phi_{j+1})(\Phi)}.$$

Take an interpretation I'_Φ such that $I'_\Phi(Call_{j+1}) = [\![Call]\!]_{G(\Phi_{j+1})(\Phi)}$, $I'_\Phi(Success_{j+1}) = [\![Success]\!]_{G(\Phi_{j+1})(\Phi)}$ and $I'_\Phi(V) = I_\Phi(V)$ for any other variable V. For any Φ, I'_Φ is a model of (7) (as $I_\Phi(B_{j+1}) \subseteq I'_\Phi(Call_{j+1})$) and hence of $F_j(C) \cup (G_0 \cup \ldots \cup G_j)(\Phi)$. It also fulfills the requirements (8) for $i = 1, \ldots, j$. If the set matching fails, then the program is not found to be correct.

Computing Φ_{j+1} (or, in the case of $j = n$, performing the inclusion check) completes the iteration step for j. The reasoning above provides a proof for:

Lemma 5.2 If the process described above succeeds producing $\Phi_1, \ldots, \Phi_n$ then the condition 1. from Theorem 5.1 is satisfied, for clause C and the parametric specification given by the parametric grammar G.

If the clause does not satisfy the condition of the Theorem 5.1 then the process of checking is bound to fail. The reverse is not true. The correctness checking of a correct program may fail, due to the fact that the employed intersection and projection operations for parametric grammars are approximate.

Due to similarity of this correctness checking algorithm to that described in Section 3.2, we expect that its complexity is the same.

Example 5.3 Consider the following clause, a part of the "Slowsort" program:

```
slowsort(L,S) :- perm(L,S), sorted(S).
```

For this clause we have the following three systems of constraints (we abbreviate *slowsort* as s, *perm* as p and *sorted* as sd):

$$
\begin{array}{ll}
F_0: & L > s(L,S)^{-L}(Call_0) \\
 & S > s(L,S)^{-S}(Call_0) \\
 & Call_1 > p(L,S)
\end{array}
$$

$$
\begin{array}{ll}
F_1: & L \ > \ s(L,S)^{-L}(Call_0) \ \cap \ p(L,S)^{-L}(Success_1) \\
 & S \ > \ s(L,S)^{-S}(Call_0) \ \cap \ p(L,S)^{-S}(Success_1) \\
 & Call_2 \ > \ sd(S)
\end{array}
$$

$$
\begin{array}{ll}
F_2: & L \ > \ s(L,S)^{-L}(Call_0) \ \cap \ p(L,S)^{-L}(Success_1) \ \cap \ sd(S)^{-L}(Success_2) \\
 & S \ > \ s(L,S)^{-S}(Call_0) \ \cap \ p(L,S)^{-S}(Success_1) \ \cap \ sd(S)^{-S}(Success_2) \\
 & Success_0 \ > \ s(L,S)
\end{array}
$$

A specification is provided by the following parametric grammar G:

$$
\begin{array}{ll}
Call > s(ListN, Any) & Success > s(ListN, ListN) \\
Call > p(List, Any) & Success > p(List, List) \\
Call > sd(ListN) & Success > sd(ListN) \\
ListN > [\,] & List > [\,] \\
ListN > [Nat|ListN] & List > [\alpha|List] \\
Nat > nat & Any > \top
\end{array}
$$

The first step of checking the correctness of the clause w.r.t. the specification deals with F_0. First one uses generalized projection operation to compute $s(L,S)^{-L}(\langle Call_0, G\rangle) = \langle ListN, G\rangle$ and $s(L,S)^{-S}(\langle Call_0, G\rangle) = \langle Any, G\rangle$. We may informally say that the first two rules of F_0 have been transformed into $L > ListN$, $S > Any$.

Then G_L and G_S are respectively $\langle ListN\rangle_G$ and $\langle Any\rangle_G$ (the subsets of G defining $ListN$ and Any), with the variables appropriately renamed. Their union is G':

$$
\begin{array}{ll}
L > [\,] & S > \top \\
L > [Nat'|L] & \\
Nat' > nat &
\end{array}
$$

The grammar G'' is just the last rule of F_0. Matching $\langle Call_1, G' \cup G''\rangle \sqsubseteq \langle Call, G\rangle$ succeeds after checking the pairs $(Call_1, Call), (L, List), (S, Any)$, (Nat', α). The result is $\Phi_1 = \{\, \alpha \mapsto \langle Nat', G' \cup G''\rangle \,\}$. So the first implication of the verification condition is satisfied, provided that $(Call_1, Success_1)$ is defined by $G(\Phi_1)$ (after an appropriate variable renaming).

We briefly outline the remaining two steps. Notice that the results of generalized projections from one step are also used in later steps.

Dealing with F_1 begins with computing two new generalized projections: $p(L,S)^{-L}(\langle Success_1, G_1\rangle) = \langle List_1, G_1\rangle$ and $p(L,S)^{-S}(\langle Success_1, G_1\rangle) = \langle List_1, G_1\rangle$, where G_1 is a renamed $G(\Phi_1)$ and $List_1$ is the renamed $List$.

(We may informally say that the first two rules of F_1 have been transformed into $L > \mathit{ListN} \cap \mathit{List_1}$, $S > \mathit{Any} \cap \mathit{List_1}$.)

Then intersection operation is applied to approximate sets $[\![\mathit{ListN}]\!]_{G(\Phi)} \cap [\![\mathit{List}_1]\!]_{G_1(\Phi)}$ and $[\![\mathit{Any}]\!]_{G(\Phi)} \cap [\![\mathit{List}_1]\!]_{G_1(\Phi)}$, by grammars $\langle \mathit{ListN}\rangle_G \dot\cap G_1$ and $\langle \mathit{Any}\rangle_G \dot\cap G_1$. The grammars are renamed, so that $\mathit{ListN} \dot\cap \mathit{List_1}$ becomes L and $\mathit{Any} \dot\cap \mathit{List}_1$ becomes S, resulting in G'. Matching $\langle \mathit{Call}_2, G' \cup \{\mathit{Call}_2 > sd(S)\}\rangle \sqsubseteq \langle \mathit{Call}, G\rangle$ does not involve any parameter and succeeds, so $\Phi_2 = \emptyset$ and G_2 is G with variables renamed.

Similarly, in the third step the projections related to atom $sd(S)$ result in $\langle \mathit{Any}_2, G_2\rangle$ and $\langle \mathit{ListN}_2, G_2\rangle$. (We may informally say that the first two rules of F_2 have been transformed into $L > \mathit{ListN} \cap \mathit{List_1} \cap \mathit{Any_2}$, $S > \mathit{Any} \cap \mathit{List_1} \cap \mathit{ListN_2}$.) Notice that $[\![\mathit{List}_1]\!]_{G_1} = [\![\mathit{ListN}_2]\!]_{G_2}$. G' obtained in this step is essentially the same as that in the previous one – the sets $[\![L]\!]$ and $[\![S]\!]$ that G' defines are the same as in the previous step. The inclusion check succeeds, which completes checking that the clause is correct.

5.3 Computing parametric specifications

Now we show how to compute a parametric specification approximating the semantics of a given program.

Consider a parametric specification *Spec*. Notice that if the verification conditions of Proposition 3.1 hold for each (non parametric) specification from *Spec* then the conditions of Theorem 5.1 hold, with $(\mathit{Pre}_0, \mathit{Post}_0) = \ldots = (\mathit{Pre}_n, \mathit{Post}_n)$. Thus the program is correct w.r.t. *Spec*. We will use this fact in constructing parametric specifications for a given program. The initial goals are described by a parametric grammar G_0. G_0 also describes the constraint predicates, similarly as in Section 3.3. We are going to construct a parametric grammar G (with the parameters from G_0) such that whenever the initial call is from $[\![\mathit{Call}]\!]_{G_0(\Phi)}$, all the calls and successes are from $[\![\mathit{Call}]\!]_{G(\Phi)}$, $[\![\mathit{Success}]\!]_{G(\Phi)}$, respectively.

To compute G we proceed as in the parameterless case (Section 3.3). The only difference is that the algorithm is now applied to parametric grammars. We require that the description of constraint predicates is parameterless. So whenever a rule $\mathit{Call} > p(Y_1, \ldots, Y_n)$ or $\mathit{Success} > p(Y_1, \ldots, Y_n)$, where p is a constraint predicate, appears in G_0 then $\langle Y_i\rangle_{G_0}$ does not contain any parameters (for $i = 1, \ldots, n$). Obviously, we require that the specification given by $G_0(\Phi)$ respects constraints.

We employ the verification conditions of Proposition 3.1 expressed as the constraint system $\mathbf{C}(P)$ (see Section 3.3). For the grammar G_0 as above, $\mathbf{C}(P)$ is parametric. $\mathbf{C}(P) = \mathbf{C}' \cup G_0$, where $\mathbf{C}'$ is a set of parameterless constraints

$$\mathbf{C}' = \bigcup_{C \in P} \bigcup_j F'_j(C).$$

Consider a parameterless instance $G_0(\Phi)$ of G_0. If I is a model of $\mathbf{C}(P)(\Phi)$

such that $Spec = (I(Call), I(Success))$ respects constraints then the verification conditions of Proposition 3.1 are satisfied, as shown in Section 3.3.

Our goal is to construct a grammar G such that for any Φ (for which $G(\Phi)$ is parameterless) there exists a model I of $\mathbf{C}(P)(\Phi)$ in which $I(Call) = [\![Call]\!]_{G(\Phi)}$ and $I(Success) = [\![Success]\!]_{G(\Phi)}$. This implies that the verification conditions of Proposition 3.1 are satisfied for each specification $([\![Call]\!]_{G(\Phi)}, [\![Success]\!]_{G(\Phi)})$. Hence the verification conditions of Theorem 5.1 are satisfied for the parametric specification

$$\{\, ([\![Call]\!]_{G(\Phi)}, [\![Success]\!]_{G(\Phi)}) \mid G(\Phi) \text{ is parameterless} \,\}$$

given by grammar G, and the program is correct w.r.t. this specification.

To obtain such a grammar we use the iterative procedure of Section 3.3. It starts with G_0 and produces a sequence of grammars G_i. Any parameter appearing in G_i occurs in G_0. The description of the constraint predicates in any G_i is the same as in G_0. The constructed grammars G_i have the following property, for any Φ (such that $G_0(\Phi)$ is parameterless): The constraints $\mathbf{C}'$ are satisfied if the occurrences of *Call* and *Success* in constraints (3) (see Section 3.1) are valuated as in the least model of $G_i(\Phi)$ and the occurrences of *Call* and *Success* in constraints (4) as in the least model of $G_{i+1}(\Phi)$. This follows from the discussion in Sections 3.2, 3.3, which can be repeated for the case of parametric grammars. The difference is that in the parameter free case the operations of intersection and projection are exact while in the parametric case they are approximate. However the conclusions hold in both cases. In particular, if G', G'' are constructed as in Section 3.2 then the least model of $(G \cup G')(\Phi)$ is a model of $F_{j,1(C)} \cup G(\Phi)$ and the least model of $(G' \cup G'')(\Phi)$ is a model of $F_{j,1(C)}$ (for any Φ assigning parameterless grammars to the parameters of G).

As discussed in Section 3.3 it is necessary to apply some technique for enforcing termination while computing fixpoints. As discussed there our prototype implementation uses for that purpose an adaptation of a technique of [Mil99], which extends also to the parametric case.

Now G_i is the required grammar. For any Φ as above there exists a model J of $\mathbf{C}'$ which coincides with the least model of $G_i(\Phi)$ on *Call* and *Success*. An interpretation I in which the variables of $\mathbf{C}'$ are valuated as in J and the variables of G_0, except of *Call*, *Success*, as in the least model of $G_0(\Phi)$, is the required model of $\mathbf{C}(P)(\Phi)$. As explained above, if such model exists then the program is correct w.r.t. the parametric specification given by G_i.

We derive a somehow restricted kind of parametric specifications. Whenever the initial goal is in $[\![Call]\!]_{G(\Phi)}$, all the calls and successes of the computation are, respectively, in $[\![Call]\!]_{G(\Phi)}$, $[\![Success]\!]_{G(\Phi)}$. Thus our approach is unable to construct such parametric specifications that various usages of a predicate in a program are described by different instances of the parametric specification.

5.4 Error detection

The purpose of error diagnosis is to locate the errors in the program. By errors we mean those program fragments that are the reasons that the program is incorrect w.r.t. a given specification. For the semantics chosen in this work, the incorrectness means that some call or success in some computation of the program violates the specification. Such calls or successes will be called error *symptoms*. A pragmatic requirement is that the errors found are as small program fragments as possible.

In traditional approaches, debugging begins with symptoms, obtained from executing the program on some test data. Obviously, only a finite subset of (usually) infinite set of test data can be used. In our approach symptoms are not needed. At the expense of restricting the class of specifications to types defined by parametric discriminative grammars, program correctness can be checked automatically. A successful check is a proof that the program is correct. Equivalently, if the program is incorrect then the check fails; moreover from the correctness checking algorithm we can obtain information locating the errors.

Our correctness checking algorithm uses the sufficient condition of Theorem 5.1. The condition consists of $n+1$ implications for each n-ary clause of the program (and an obvious condition on the initial atomic goals). Each implication concerns a prefix $H \leftarrow B_1, \ldots, B_i$ of a clause $H \leftarrow B_1, \ldots, B_n$ $(1 \leq i \leq n)$.[34] Two implications concern the whole clause $(i = n)$. If the program is incorrect then some of the implications do not hold. The clause prefixes corresponding to these implications will be considered the errors of the program.

Definition 5.4 Let P be a program and *Spec* a parametric specification. An *error* in P (w.r.t. *Spec*) is a prefix $H \leftarrow B_1, \ldots, B_{k+1}$ $(0 \leq k \leq n-1)$ of a clause $H \leftarrow B_1, \ldots, B_n$ of P, or the whole clause $H \leftarrow B_1, \ldots, B_n$ (then $k = n$) such that for some $(Pre_0, Post_0) \in Spec$ and for each $(Pre_1, Post_1), \ldots, (Pre_k, Post_k) \in Spec$ such that the implication of Theorem 5.1 holds[35] for $j = 0, \ldots, k-1$, there exists a substitution θ and constraint c such that $c \,[]\, H\theta \in Pre_0,\ c \,[]\, B_1\theta \in Post_1,\ \ldots,\ c \,[]\, B_k\theta \in Post_k$ and

$$c \,[]\, B_{k+1}\theta \notin Pre_{k+1} \text{ for any } (Pre_{k+1}, Post_{k+1}) \in Spec, \text{ if } k < n,$$
$$c \,[]\, H\theta \notin Post_0, \text{ if } k = n.$$

We say that the *representative* of the error is B_{k+1} when $0 \leq k \leq n-1$, or H when $k = n$. (So it is the atom whose instance is found incompatible with the specification). □

[34] In the notation of Theorem 5.1, $i = j+1$ if $j < n$ and $i = n$ if $j = n$.

[35] This means that for any substitution θ and constraint c

if $c \,[]\, H\theta \in Pre_0,\ c \,[]\, B_1\theta \in Post_1,\ \ldots,\ c \,[]\, B_j\theta \in Post_j$

then $c \,[]\, B_{j+1}\theta \in Pre_{j+1}$

This definition formalizes the intuition of a program fragment being the reason of incorrectness. Such fragments have to be changed in order to obtain a correct program. On the other hand, in a general case there are no semantic criteria to state what in such a fragment has to be changed. In this sense the errors defined above are minimal. What is "the error" from the pragmatic point of view, depends on the programmer's intentions about the exact intended semantics of the program.

Example 5.5 Consider a type specification

$$\begin{array}{lll} \mathit{Call} > m(\mathit{Any}, L) & L > [\,] & \mathit{Any} > \top \\ \mathit{Success} > m(\alpha, L) & L > [\alpha|L] & \end{array}$$

and a clause `m( X, [Y,Z] ) :- m( X, Z ).` The (prefix being the) whole clause is incorrect w.r.t. the specification, as for $j = 0$ the second argument of the call $m(X, Z)\theta$ is, speaking informally, of type α instead of L. We cannot state which atom of the clause is erroneous. To obtain a correct clause one may for instance replace $m(X, [Y, Z])$ by $m(X, [Y|Z])$, or $m(X, Z)$ by $m(X, [Z])$. Only knowing that m is intended to define a list membership relation, makes it possible to decide what is the actual error (w.r.t. the (exact) intended semantics of the program).

Notice that there is at most one error in a given clause, as Definition 5.4 requires that the implications for $j = 0, \ldots, k-1$ hold. Thus according to our definition each proper prefix of an error is not an error. The reason is that if $H \leftarrow B_1, \ldots, B_{j+1}$, $0 \leq j < k$, were an error then we would not have a criterion which $(Pre_{j+1}, Post_{j+1})$ to consider in determining that $H \leftarrow B_1, \ldots, B_{k+1}$ is an error.[36]

We will use the correctness checking procedure from the previous section to locate errors in programs. If a clause contains an error then the procedure will fail. The reverse is not true, correctness checking of a clause not containing an error may fail, due to approximation inaccuracies of the intersection and projection operations.

The correctness checking procedure finds each clause containing an error. Moreover, to a certain extent a clause prefix containing the error is located. If $\Phi_1, \ldots, \Phi_j$ are successfully constructed then each prefix $H \leftarrow B_1, \ldots, B_i$, for $i = 1, \ldots, j$ is not an error. If then constructing of Φ_{j+1} fails, it is possible that some of prefixes $H \leftarrow B_1, \ldots, B_i$, where $i > j$, is an error. If no approximation inaccuracies had appeared then $H \leftarrow B_1, \ldots, B_{j+1}$ would have been an error. The inaccuracies make it possible that some larger prefix is an error or the clause does not contain an error.

[36]Such a criterion may be obtained by setting $Post_{j+1} = \bigcup\{\, Post \mid (Pre, Post) \in Spec \,\}$. We expect however that the definition modified in such way would define errors which do not correspond to an intuitive notion of an error.

6 The prototype diagnosis tool

6.1 The structure of the tool

We implemented a prototype tool that locates errors by checking correctness of a program wrt types specified by PED grammars. Notice that such a grammar may or may not include parameters. As already mentioned in the Introduction, the tool consists of three main components:

- *the type inferencer* – for a given program and parametric entry declaration constructs parametric directional types of the program using the technique of Section 5.3. The types approximate the program semantics.

- *the type checker* – checks correctness of a program wrt to given parametric directional types using the technique of Section5.2

- *the specification editor* – a GUI which makes it possible to specify intended directional types and also to inspect and to re-use in this specification the inferred types.

A diagnosis session starts with type inference. The inferencer may issue some warnings about illegal calls to built in predicates. It happens if the inferred call type for a built-in is not a subtype of the expected one. The expected call types for built-ins are stored in the system library and may be viewed as a part of specification given a priori.

The main part of the session consists in providing/editing by the user a specification of the intended types. The type checker works interactively with the editor. Each verification condition is checked as soon as a sufficient fragment of a specification is provided. The diagnosis relies entirely on the provided types. It does not involve execution of the program and it does not use the inferred types. The role of type inference is auxiliary. As mentioned above, the inferencer may discover certain irregularities in the program and its warnings suggest starting points for the diagnosis. On the other hand, the inferred types may be used as a draft for the specification; this simplifies the task of constructing the specification by the user.

The current version of the tool supports a substantial subset of the CHIP language. It can be easily modified to be used with any Prolog-like language. The prototype has been implemented in SICStus Prolog. A more detailed description of our tool, in its version for parameterless specifications, together with an example error diagnosis session is given in [DMP00a].

6.2 Types

The parametric specifications used by the tool are PED-grammars defined in Section 4.3. For every parameter valuation such a grammar defines a set of constrained terms. A parametric type defined by such a grammar can be seen as a family of sets (of constrained terms).

In the implementation we use the notation as shown in the example below. We write

```
:-typedef tree --> nil; t(elem,tree,tree).
```

to denote the grammar

$$
\begin{aligned}
&Tree > nil\\
&Tree > t(Elem, Tree, Tree)
\end{aligned}
$$

Such a grammar may be a part of a program.

The present version of the tool uses four *base* types:

- `any` denotes $[\![\top]\!]$,
- `nat` denotes the set of natural numbers,
- `anyfd` denotes the set of constrained atoms of the form $x \in \mathrm{FD} \,[]\, x$ where FD is a finite domain, i.e. a finite set of natural numbers[37],
- `int` denotes the set of integers.

The approach to base types in the implementation does not satisfy Requirement 2.13. Namely, sets denoted by `anyfd` and `int` are neither disjoint nor one of them includes the other. This design choice remained from the previous versions of our approach. It is dealt with by some ad hoc modifications of the grammar operations. It will be changed, by adding a base type *neg* of negative numbers and defining the set of integers as the union of $[\![nat]\!]$ and $[\![neg]\!]$.

The type of a top call for a program is provided with `entry` declaration, for instance:

```
:- entry delete(list(A),A,any).
```

Parameters are identifiers written with capital letter (like variables in Prolog). Thus the above declaration says that we intend to delete an element of an arbitrary type `A` (the second argument) from the list of elements of that type (the first argument). The third argument is supposed to be a variable on call, which can be only expressed as `any`.

To make the system interface more user-friendly we introduced a library of type definitions which may be augmented by the user. It contains for instance, a parametric grammar defining type `list(A)`, i.e. lists of elements of type `A`.

Whenever possible, the types computed by the system are presented to the user in terms of those defined in the library or declared by the user. In

[37]We do not distinguish between c and $x \in \{c\} \,[]\, x$.

this way the user faces familiar and meaningful type names instead of artificial ones. For instance, assume that the system has to display a type `t77` together with the grammar rule `t77 --> [];[t78|t77]`. Then it finds that they are an instance of the rules defining `list(A)` and displays `list(t78)` instead.

When providing the specification the user gives intended call and success types for a given program. Formally this means providing grammar rules for *Call* and *Success*. So the grammar providing the specification consists of the rules kept in the library, the grammar rules given in `:-typedef` declarations of the program and the rules for *Call* and *Success* provided by the user during the diagnosis section.

6.3 Inferring and checking types

The type inference algorithm is based on the description of Sections 3.3 and 5.3. It computes an approximation of call-success semantics of a given program. This is done by means of fixed point iteration. The algorithm is implemented in Prolog.

For all programs used in our experiments (up to 230 clauses and 52 predicates) the prototype implementation computes approximations in reasonable time[38].

As already mentioned in Section 3.3, in the parameterless case the algorithm can be seen a method of solving set constraints. However, the solution obtained is in general not the least one because of widening and of the approximate nature of the union operation which is used by the algorithm. Extension to the parametric case introduces additional loss of information caused by the operations on PED grammars discussed in Section 4.4.

The type inferencer is not able to find polymorphic dependencies between variables by itself. The only parameters that may appear during the analysis are those provided by the user in the entry declaration.

As discussed in Section 4.4, the definitions of operations on PED grammars include some arbitrary decisions. The union and the intersection of a type parameter with another type are, respectively, $[\![\top]\!]$ and the other type. The implementation produces a warning whenever these situations appear during type inference.

The rationale behind the warnings is as follows. The type parameter in call specification reflects the intuition that any instance of the parametric type is allowed at call. Normally it means that the analyzed procedure is polymorphic and it is supposed to work for any instance of the parameter. Thus the result of the analysis should be independent on potential instantiations of the parameter. In other words, none of the operation on types

[38]21.88 s in the worst case, running SICStus Prolog, ver.3.8.4 on Sun-Ultra 10/440, with 440 MHz CPU speed and 265 MB RAM.

should touch parameters. If it happens then the procedure may not work as a polymorphic one.

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

As already mentioned, the diagnosis relies on the type specification provided incrementally by the user. The specification process is supported by the possibility to accept some types constructed in the analysis phase as specified ones. This possibility is restricted to the types of the predicates relevant for the diagnosed predicate. Moreover, a heuristics is used to suggest to the user the order of specifying types. Following this order often results in fewer type specifications needed to locate an error. The user may stop the diagnosis with the first error message, which is often obtained without specifying all requested types. The diagnosis process may be continued by specifying all requested types. In this case, the tool will locate all incorrect clause prefixes in the fragment of the program relevant for the diagnosed predicate.

An error message contains an incorrect clause. The incorrect prefix is indicated by referring to its representative (cf. Definition 5.4). The specification provided by the user is stored by the diagnoser and may be re-used during further diagnosis sessions.

6.4 Examples

Below we show some examples illustrating the use of the diagnosis tool. The examples exhibit an advantage of parametric analysis over the non-parametric one.

Consider the following erroneous program:

```
append([],Ys,Ys).
append([H|Xs],Ys,[H,Zs]) :-
   append(Xs,Ys,Zs).
```

The head of the second clause should be `append([H|Xs],Ys,[H|Zs])`. Assume that the `append/3` predicate is supposed to concatenate two lists of any arbitrary type. In the non-parametric framework the best way to express such a type is `list(any)`. After analyzing the program with the following entry point declaration:

```
:-entry append(list(any),list(any),any).
```

the inferred success type is

```
append(list(any),list(any),list(any))
```

The reason for inferring such a (success) type for the third argument of `append/3` is that the type of two-element list originating from the head of the second clause (`[H,Zs]`) has been joined, by means of the upper bound operation, with the type `list(any)` coming from the recursive call of `append/3`. It results in the type `list(any)`. Thus nothing suspicious can be concluded.

On the other hand, if we provide a parametric declaration:

```
:-entry append(list(A),list(A),any).
```

then the inferred success type does not meet our expectations:

```
append(list(A),list(A),list(any))
```

as we would rather wish to have `list(A)` as a result. Moreover, the analyzer warns us that the parameter `A` (originating from the success of the first clause and type `list(A)` of `Ys`) will be approximated by `any` while computing an upper bound with the type `list(A)` (originating from the second clause and the term `[H,Zs]`, in which `Zs` is of type `list(A)`.

After the user has specified the success type, the diagnoser locates the error and reports it by indicating its representative `append([H|Xs],Ys,[H,Zs])`.

The next example is a fragment of a job scheduling program. The fragment sets up precedence constraints among the jobs. A job is described by a term `job(T,P)`, where `T` is a starting time of processing the job and `P` is its duration. As `T` has to be found by the program it is a domain variable; `P` is fixed. The jobs are kept in a list and are identified by the position in it.

The precedence between two jobs is represented as a term `prec(J1,J2)`, with a meaning: `J2` cannot start before `J1` has been completed. All such pairs are kept in the list. The precedence constraints are set up by the procedure `precedences/2` defined below.

```
:-typedef tprec --> prec(nat,nat).
:-typedef tjob --> job(anyfd,nat).

:-entry precedences(list(tprec),list(tjob)).

precedences([],_).
precedences([prec(A,B)|Ps],Jobs) :-
    get_nth(Jobs,A,job(TA,PA)),
    get_nth(Jobs,B,job(TB,_)),
    TB #>= TA + PA,
    precedences(Ps,Jobs).

get_nth([_|X],1,X) :-!. % bug here
get_nth([_|Xs],N,X) :-
    N1 is N - 1,
    get_nth(Xs,N1,X).
```

The `:-typedef` declaration defines new types used in the entry declaration. The first clause defining `get_nth/3` contains a bug, as the first argument of its head should be `[X|_]`.

The inferred success type for `precedences/2` is:

`precedences(t52,list(tjob))`

together with a definition of `t52`:

`t52-->[]`

This means that the procedure may succeed only when the precedence list is empty. If a diagnosis session is started with this predicate the user is asked to provide expected call and success types for `get_nth/3`. Assume they are respectively:

`get_nth(list(A),int,any)`

and

`get_nth(list(A),int,A)`

After this step the diagnoser presents as an error the clause prefix pointed by the representative

`get_nth([_|X],1,X).`

The reason for the error message is that inclusion check of `list(A)` and `A` fails. Notice however, that in non-parametric framework the specification for `get_nth/3` could be `get_nth(list(any),int,any)`, both for calls and successes. In this case the inclusion check of `list(any)` and `any` would succeed, and the bug would not be discovered by the diagnosis.

7 Discussion and Conclusions

7.1 Related Work

This work is directly related to:

- the research on proving partial correctness of logic programs wrt call-success specifications,
- the research on approximating semantics of logic programs by descriptive types based on set constraints and on abstract interpretation.

It extends some of the techniques proposed in these fields to handle parametric polymorphism and constraint domains.

Partial correctness. From [BLR92, Apt93, BC89] and our own previous work [DM88, BM97] we extend to CLP a *directional* view of logic programs in the sense that each predicate is considered a procedure which, when applied to a suitable tuple of call arguments returns upon a success a tuple of computed values. This is formalized by the notion of call-success semantics.

We rely on the proof methods of [DM88, BC89] for proving partial correctness of logic programs wrt call-success specification. We use their modification for CLP described in [DMP00b, DMP00a] and we extend them to deal with parametric specifications. For specifications formulated as *definite set constraints* [HJ90a][39] correctness can be effectively checked by reformulation of the verification conditions of the above mentioned methods, also as definite set constraints. As discussed in Section 3.1 such a reformulation requires specific operation called *generalized projection*, which is a special case of the "quantified set expression" of [HJ94] and "membership expression" of [DTT97, TDT00]. For the reasons discussed in Section 2.1.1 we choose as our specification language a parametric variant of well-known formalism of discriminative regular term grammars[40] see e.g. [DZ92] additionally equipped with basic types for handling constrained terms and atoms of CLP. The same language is used for describing approximations of call-success semantics. Traditionally such approximations are called *descriptive types* of logic programs.

Soundness of our method of type checking is stated by Lemma 3.2. which gives a sufficient condition for correctness of CLP programs for specifications given as term grammars. This result extends then for PED grammars. A recent paper [CGLV00] argues that such sufficient conditions for verification of Logic Programs can be systematically derived if the considered class of specifications is defined as an abstract interpretation domain with Galois connection relating them to a concrete semantics of logic programs. Unfortunately, as shown in [DP98], for our non-parametric specifications such a Galois connection does not exist,[41] so that it is not clear whether the method is applicable.

Types in logic programming. We follow the *descriptive typing* approach where types approximate a posteriori the semantics of untyped programs. The early work on descriptive types [Mis84, FSVY91, YS91] was based on the least model semantics. The problems considered were how to check that the least model semantics is included in a regular set of terms (the type checking problem) and how to approximate it by regular sets (the type inference problem). The regular sets were defined by regular grammars or equivalently by regular unary logic programs [FSVY91]. This approach does not take into account the intended use of the predicates and gives therefore a few possibilities for finding typing errors. The focus is mostly on detecting that for some predicates the inferred types are empty sets in which case the predicates never succeed.

Checking of directional types based on set constraints was discussed in [AL94]. The types used are sets of non-ground terms. They are specified by set constraints together with a lifting function *Sat* that maps a set of

[39] Later studied also by [CP97] and [TDT00].

[40] Such grammars define sets acceptable by deterministic root-to-frontier tree automata. Alternatively, the sets are called tuple-distributive or path-closed.

[41] The abstraction function does not exist, as there does not exist the best approximation of a given set of terms by a regular set of terms. This holds for both kinds of regular sets, those defined by discriminative and by arbitrary term grammars.

ground terms to a set of nonground terms. Type checking is based on the same verification condition we use, which in general form originates from [DM88, BC89] and was specifically formulated for directional type checking in [Apt93]. We also allow nonground types but in contrast to this work we achieve non-groundness not by lifting ground sets but by extending set constraints with constants interpreted as basic nonground types.

Inference of directional types in the framework of set constraints was illustrated by an example in [Hei92a]. (The main topic of the paper are implementation techniques for solving set constraints.) In the example the types are inferred by constructing set constraints analogous to our encoding of verification conditions, and solving them. A more recent work on inference of directional types for logic programs is [CP98]. It rephrases (as Theorem 1) the verification conditions of [Apt93] in model-theoretic setting[42]. Thus, a starting point for type inference are again the verification conditions used both in [AL94] and in our work. In [CP98] the directional types are regular, but in general not discriminative. They are characterized by the least model of a *uniform program* constructed from the original program. The authors are not specific about the algorithms to be used for constructing a representation of the resulting directional types. In contrast to this work we do not construct uniform programs. We encode the verification conditions as set expressions. Directional types are models of these expressions. We restricted our attention to discriminative directional types. This made it possible to extend the type checking and type inference algorithms of [GdW94, Mil99], based on abstract interpretation, to the case of parametric directional types.

Our work follows the idea of using semantic approximations for program verification and for locating errors presented in [BDD$^+$97]. This idea was also used for designing a generic preprocessor for validation and debugging of CLP programs [PBH00b]. The preprocessor verifies various assertions, provided by the user or inferred, in particular also non-parametric discriminative directional types similar to ours.

While most of the papers on types in logic programming claim error detection as their objective, a little attention is usually devoted to locating errors. In this paper we extend our previous approach to locating errors [DMP00b, DMP00a] to the case of polymorphic types. As discussed in Section 6 this gives some more opportunities to locate the reasons of discrepancy between actual program and user expectations.

Parametric Polymorphism in Logic Programming. Use of parametric polymorphic types in logic programming was first suggested in [MO84]. In this approach the function symbols and the predicates of a logic program are supposed to have a priori declared types. The types are used to restrict the syntax of the language to *well-typed* formulae. A compile-time test is then formulated which gives a sufficient condition that

[42]These conditions are stated as *magic transformation* of the original program.

well-typedness is an invariant of goals in all computations. This approach to using types, called *prescriptive* typing has been followed in many papers and in several logic programming languages, most notably Gödel [HL94] and Mercury [SHC96]. Semantically, prescriptive typing corresponds to taking many sorted typed logic as a foundation of logic programming, instead of untyped logic. Our approach is based on untyped logic and our parametric types approximate actual or intended semantics of the program. Thus, our work is in the framework of *descriptive* types, and the vast literature on prescriptive types is not further discussed here. Let us only mention some recent research on this topic [FC01, SFD00, DS01].

In the context of descriptive typing some preliminary ideas on the issue of parametric polymorphism are discussed already in [Mis84] as a possible extension of the presented type checking method for non-directional types. [Zob87] presents a method for deriving "syntactic" polymorphic types. These types are not directional. They are clearly related to term grammars but the paper does not explain the relationship. Our techniques focus on directional types and are based on semantic considerations.

Polymorphic directional types for logic programs discussed in [Boy96] are based on the *annotation method* of [Der93] for proving correctness of logic programs. This method is different from that used in our work and refers to a different semantics. In spite of that the verification conditions have a similar nature to ours and give rise to similar parametric set constraints. Our work goes further in that we use such parametric constraints in a sufficient correctness test, and also for type inference, while the simplification techniques of [Boy96] are rather limited in handling parameters.

The problem of polymorphic directional type checking is also addressed in [RT00] and more recently in [RT01]. This work presents a formal system, where directional well-typing of a logic program for given type specification is defined in terms of proofs constructed from given axioms and typing rules. This is different from our approach where the well typing algorithms are derived from the semantic concept of program correctness and types are understood as families of sets, specified by means of PED-grammars. Thus it seems impossible to compare our type checking algorithms with those discussed in [RT00].

Nevertheless, the semantics of types as sets is also provided in [RT00]. It is done by a fixpoint construction, which for a given alphabet of typed function symbols associates each used type with a subset of the Herbrand universe. In this, rather indirect, way a similar effect is obtained as by our direct specification of types by means of PED-grammars. However, the class of the sets which can be constructed in that way is not precisely characterized. Syntactic restrictions on the way of defining signatures seem to make it somewhat restricted. For example, it is impossible to have nonempty intersection of instances of different polymorphic types, e.g. [] cannot be used for representing both the empty list and the empty tree.

This is a substantial restriction, e.g. one cannot define a type of even length lists.

The soundness theorem of [RT00] relates the directional types of well-typed programs to their declarative semantics, while the types discussed here are related to the call-success semantics. Failure of our type checking algorithm locates potential errors in a fragment of a clause, while a proof failure of [RT00] seems to indicate a whole clause. (At least this issue is not discussed in that paper.) Handling of constraints is not discussed in their work, its main objective is representing different directional types of a predicate by one *main* type.

7.2 Conclusions

We extended the concept of partial correctness for a logic program wrt to a directional type to a concept of partial correctness of a CLP program wrt to a parametric directional specification. We formulated sufficient conditions for the correctness and we encoded them as set constraints. In this way we gave a semantic-based view of parametric polymorphism in constraint logic programs.

We extended the notion of discriminative term grammar to the notion of parametric extended discriminative term grammar (PED grammar). We argued that directional types specified by such grammars are quite useful. On one hand, they make it possible to describe simple approximations of program semantics, easy to provide and to understand by the user. On the other hand, they allow automatic check of the sufficient conditions mentioned above. Using these conditions one can also automatically infer parametric directional types from a parametric entry declaration.

Our type inference techniques extend to CLP and to the parametric types the techniques of [GdW94] corrected by Mildner [Mil99]; they are based on abstract interpretation of logic programs. It seems possible to extend instead some of the set constraint solving techniques. This may be a topic of future work including also a comparison of both extensions.

We developed a prototype tool implementing the proposed algorithms, which can be obtained from the third author. The theoretical result of [CP98] shows that the problem of checking discriminative directional types is not tractable, even in the parameterless case. The complexity of our type checking algorithm is exponential w.r.t. the maximal number of occurrences of a variable in a clause. However our tool turns out to be sufficiently efficient for practical purposes.

Our tool supports a compile-time technique for error location based on checking directional parametric types. Clearly, the class of errors that can be located is restricted to type errors. The check locates those clause prefixes, which cause the type errors. Our approach does not impose any type discipline on the program. It does not require providing all type declarations in advance and often only a few declarations are sufficient to locate an error. The process of specifying declarations is supported by the possibility of inspecting and adopting the inferred types.

Acknowledgments

We want to thank the anonymous referees for insightful comments and suggestions which resulted in substantial improvements of this paper. We acknowledge the contribution of Marco Comini who was largely involved in the development of an early version of the diagnosis tool. We also thank our colleagues from the DiSCiPl Project in which this research has been initiated. In particular we are grateful to Pierre Deransart and Manuel Hermenegildo for fruitful discussions. This work was partly funded by The Swedish Research Council grant 1999-109.

Paper IV

Improving goal directed bottom-up evaluation of logic programs

Jan Carlson and Paweł Pietrzak

Abstract

This paper introduces a new strategy for goal directed bottom-up evaluation of logic programs. The strategy is based on a combination of two known techniques: dividing a program into strongly connected components (SCC) and Induced Magic-sets.

In our approach no magic transformation is needed (as in Induced Magic-sets). We introduce a notion of a *call-success dependency graph* that is constructed directly from the original program. Its SCCs are used to optimize a fixpoint computation (i.e. they coincide with SCCs of a corresponding magic program). The new graph contains substantially fewer edges than standard dependency graph of a magic program, and thus its SCCs are computed faster. We also show how to incorporate SCC-based optimization into the Induced Magic-sets technique. This is achieved by modifying the basic Induced Magic method, so that the fixpoints are computed locally for every SCC.

The impact of our method is illustrated by experimental results.

Keywords: magic-sets, bottom-up evaluation

1 Introduction and motivation

Bottom-up fixpoint evaluation of logic programs has two main applications: answering queries in deductive databases and static program analysis. Computing a fixpoint in such an approach is often inefficient and thus a number of optimization techniques have been proposed. One of them is splitting the predicate dependency graph of the program into *strongly connected components* (SCC for short) [MUV86], and then computing a local fixpoint for each of them.

The bottom-up scheme is often employed in conjunction with the *magic-sets* program transformation [BMSU86, RLK86, Sek89, Ram91], which aims at bottom-up goal-directed evaluation of logic programs. The magic-sets transformation results in a program (called a *magic program*) which is then evaluated bottom-up. This method is devised to simulate top-down execution of the original program with the given top goal. The technique, however, is also known to cause inefficiency problems due to a high degree of overlapping in the clause bodies in the magic program. One possible remedy to this problem is the *Induced Magic-sets* strategy [Cod99].

In this paper we show how to combine SCC-based optimization with Induced Magic. First, we describe a new method for finding SCCs of the predicate dependency graph efficiently. Up to our knowledge, finding SCCs of a magic program has been (so far) performed in a straightforward way (see e.g. [GdW94]), i.e. the magic program is constructed and then its components are computed as in the case of any logic program. However, it turns out that the overlapping of clause bodies in the magic program is reflected in redundant edges in its dependency graph. We exploit this feature by constructing a graph (called a *call-success dependency graph*) which has the same set of SCCs as the dependency graph of the magic program, but contains substantially fewer edges. SCCs of such a graph are then computed much faster then those of the dependency graph of a magic program. The call-success dependency graph can be easily built directly from the original program and thus we do not have to explicitly perform the magic transformation. This makes it possible to combine this technique with Induced Magic-sets, which also does not perform the magic-sets transformation and operates directly on the original program. We present a modification of the Induced Magic algorithm, that for a given program and its (decorated) call-success dependency graph performs a fixpoint evaluation over a given domain.

As indicated by experiments, combining SCC-based optimization with Induced Magic brings significant improvements in terms of efficiency. Moreover, if a widening function is applied, the division of the program into SCCs sometimes leads to an increased accuracy of the analysis. When the widening is applied to an intermediate result of analyzing the whole program, it might give worse over-approximation than when applied to a single SCC. Particularly, if we take advantage of the information that some SCCs are non-recursive.

The rest of the paper is structured as follows. Section 2 surveys briefly basic notions like bottom-up evaluation, the magic transformation and Induced Magic-sets techniques, and dependency graphs. Section 3 introduces a concept of a call-success dependency graph and gives its properties. The next section describes a method that employs the Induced Magic-sets strategy enhanced with SCC optimization. Benchmark results are presented in Section 5. Finally, Section 6 gives some conclusions.

2 Preliminaries

In the following we assume familiarity with basic notions of logic programs as described in [Apt90, Llo87b].

2.1 Bottom-up evaluation

We define bottom-up evaluation in general terms, without determining its exact purpose. Different variants of bottom-up evaluation, such as computing models of datalog programs or approximating declarative semantics [GdW94], are viewed as instances of our bottom-up evaluation concept, making our method applicable to a wider range of tasks.

The fundamental idea of bottom-up evaluation is that every clause c of a logic program defines an operator $O^c : D \rightarrow D$ on a single partially ordered set D, sometimes refered to as the *domain* of the evaluation. Similarly, a program P defines the operator $O^P : D \rightarrow D$, where $O^P(d)$ is defined as the least upper bound of $\{O^c(d) \mid c \in P\}$.

Different instances of bottom-up evaluation use different domains and different definitions of the operations, but the goal is always to find a fixed point of O^P. The evaluation consists of starting from the bottom element of D, $d_0 = \perp$ and iteratively computing $d_i = O^P(d_{i-1})$ until we find a fixed point of O^P, that is a $d_i \in D$ for which $d_i = O^P(d_i)$.

In general, one must ensure that a fixed point is always found in a finite number of steps. This is for example achieved when the domain is finite (or a poset of finite length), or by applying some widening technique [CC92a]. Termination of bottom-up evaluation is outside the scope of this article, since the method presented is equivalent to ordinary bottom-up evaluation in this respect (i.e. our method does not change the termination criteria). For a thorough discussion on the subject see, for example [Nil92].

Although we allow the exact definition of $O^c(d)$ to differ, we require that it can be defined in terms of three operations (denoted $\mathbb{B}$, $\oplus$ and $\mathbb{C}$) as follows.

Definition 2.1 If c is the clause $h \leftarrow b_1, \ldots, b_n$ we define $O^c(d)$ as

$$O^c(d) = \mathbb{C}(h, \bigoplus_{i=1}^{n}(\mathbb{B}(b_i, d)))$$

The three operations are arbitrary, and we only restrict the domains on which they are defined. Let D be a partially ordered set, and let T be the set of all atoms over the current alphabet. Finally, let E contain mappings from variables in the alphabet to elements of some arbitrary domain. Then we have

$$\begin{array}{lcl} \mathbb{B} & : & T \times D \rightarrow E \\ \oplus & : & E \times E \rightarrow E \\ \mathbb{C} & : & T \times E \rightarrow D \end{array}$$

□

The role of $\mathbb{B}$ can be described informally as collecting information about the variables of a body atom with respect to an element from D. The operator $\oplus$ combines the information gained from different atoms, and finally

$\mathbb{C}$ constructs an element in D from the head of a clause and the bindings from the whole body. A graphical presentation of this can be found in Figure 6.

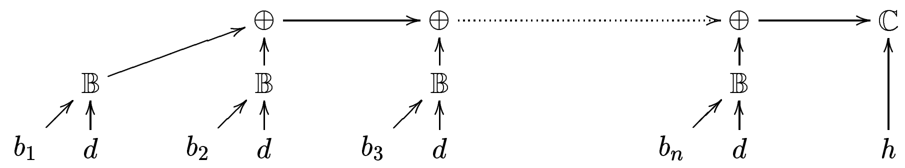

Figure 6: Computation of $O^c(d)$, where c is the clause $h \leftarrow b_1, \ldots, b_n$

Example 2.2 When bottom-up evaluation is used to compute the least Herbrand model of datalog programs, D is the powerset of the Herbrand base, and E is the powerset of all substitutions binding variables to ground terms. The three operations are defined as follows.

$$\begin{array}{lcl} \mathbb{B}(t,d) & = & \{\theta \mid t\theta \in d \ \wedge \ \theta \text{ contains only variables from } t\} \\ \oplus(e_1,e_2) & = & \{\theta_1 \cup \theta_2 \mid \theta_1 \in e_1 \ \wedge \ \theta_2 \in e_2 \ \wedge \\ & & \qquad \wedge \ [(X/t_1 \in \theta_1 \wedge X/t_2 \in \theta_2) \ \Rightarrow \ (t_1 = t_2)]\} \\ \mathbb{C}(t,e) & = & \{t' \mid \theta \in e \ \wedge \ t' \text{ is a ground instance of } t\theta\} \end{array}$$

Let c be the clause $p(X) \ \leftarrow \ q(X,Y), \ r(Y)$ and assume that we have the domain element $d = \{q(a,a), \ q(b,b), \ q(c,d), \ r(b), \ r(c), \ r(d)\}$. Then the computation of $O^c(d)$ would look like this:

$$\begin{array}{rcl} e_1 = \mathbb{B}(q(X,Y),d) & = & \{\{X/a,Y/a\}, \ \{X/b,Y/b\}, \ \{X/c,Y/d\}\} \\ e_2 = \mathbb{B}(r(Y),d) & = & \{\{Y/b\}, \ \{Y/c\}, \ \{Y/d\}\} \\ e_3 = \oplus(e_1,e_2) & = & \{\{X/b,Y/b\}, \ \{X/c,Y/d\}\} \\ O^c(d) = \mathbb{C}(p(X),e_3) & = & \{p(b),p(c)\} \end{array}$$

2.2 Dependency graphs and strongly connected components

As already mentioned, the bottom-up evaluation may be optimized by dividing the program into parts that can be evaluated separately. Those parts are *strongly connected components* of the predicate dependency graph of the program. Below we recall some related definitions.

A digraph (directed graph) G is a pair $(V, \rightarrow)$ of vertices V and edges $\rightarrow$. Let $\overset{+}{\rightarrow}$ denote a transitive closure of $\rightarrow$.

Definition 2.3 A *strongly connected component* of a digraph $G = (V, \rightarrow)$ is a greatest set $\mathcal{C} \subseteq V$ s.t. for every $v_1, v_2 \in \mathcal{C}$ $(v_1 \neq v_2)$ there is a path $v_1 \overset{+}{\rightarrow} v_2$. □

Let $SCC(G)$ denote a set of all SCCs of a digraph G.

Definition 2.4 We say that two SCCs $\mathcal{C}_1$ and $\mathcal{C}_2$ of a given digraph G are in order $\Rightarrow_G$ iff there exist $v_1 \in \mathcal{C}_1$ and $v_2 \in \mathcal{C}_2$ s.t. $v_1 \rightarrow_G v_2$. □

Definition 2.5 A *reduced graph* of a given graph G (denoted $\mathcal{R}(G)$) is a pair $(SCC(G), \Rightarrow_G)$. □

Definition 2.6 We say that a predicate p *directly depends on* a predicate q in a program P iff P contains a clause where p appears in the head and the body contains an atom with the predicate q. A predicate p *depends on* q iff (p, q) is in the transitive closure of "directly depends on". □

Let $Pred(P)$ denote the set of all the predicates occurring in P.

Definition 2.7 A *dependency graph* of a program P (denoted as $Dep(P)$) is a digraph $(Pred(P), \rightarrow_P)$ s.t. $p \rightarrow_P q$ iff q directly depends on p in P. □

Notice, that $\stackrel{+}{\rightarrow}_P$ corresponds to the relation "depends on".

Consider a clause

$$p(\overline{u}_0) \quad \leftarrow \quad q_1(\overline{u}_1), \ldots, q_n(\overline{u}_n) \in P$$

It contributes with n edges to $Dep(P)$:

$$q_1 \rightarrow p, \quad q_2 \rightarrow p, \quad \ldots, \quad q_n \rightarrow p$$

The purpose of introducing the above notions is to apply them to bottom-up evaluation of logic programs. Before the bottom-up iteration begins, the SCCs of $Dep(P)$ are constructed. Computation of the fixed point is divided into a sequence of fixpoint computations: one for each SCC. In such a sub-computation only clauses defining predicates in the current SCC are traversed. The entire algorithm is guided by a topological ordering on $\mathcal{R}(Dep(P))$.

2.3 Magic programs

The major drawback of pure bottom-up approach is that all the facts (answers or consequences of the program) are generated, regardless of the top goal. There is a well-known technique named the *Magic-sets* program transformation [BMSU86, RLK86, Sek89, Ram91], that aims at overcoming this disadvantage. In the context of program analysis we typically need to distinguish between answers and calls.

In this paper we assume the simplest form of Magic-sets technique. We assume the Prolog's computation rule, i.e., the clause bodies are traversed from left-to-right. Moreover, without loss of generality, we assume that the top goal is atomic.

Definition 2.8 Let P be a program and $\leftarrow G$ an initial query. The magic program $P_G^{\mathcal{M}}$ is defined as follows:

- with each predicate symbol p from P we assign two new symbols ${}^{\bullet}p$ and $p^{\bullet}$,
- for every clause

  $$p(\overline{u}_0) \;\leftarrow\; q_1(\overline{u}_1),\; \ldots,\; q_n(\overline{u}_n) \;\in P$$

  there are $n+1$ clauses in $P^{\mathcal{M}}$:

  $$\begin{array}{llll} (1) & {}^{\bullet}q_1(\overline{u}_1) & \leftarrow & {}^{\bullet}p(\overline{u}_0) \\ (2) & {}^{\bullet}q_2(\overline{u}_2) & \leftarrow & {}^{\bullet}p(\overline{u}_0),\; q_1^{\bullet}(\overline{u}_1) \\ & \vdots & & \\ (n) & {}^{\bullet}q_n(\overline{u}_n) & \leftarrow & {}^{\bullet}p(\overline{u}_0),\; q_1^{\bullet}(\overline{u}_1),\; \ldots,\; q_{n-1}^{\bullet}(\overline{u}_{n-1}) \\ (n{+}1) & p^{\bullet}(\overline{u}_0) & \leftarrow & {}^{\bullet}p(\overline{u}_0),\; q_1^{\bullet}(\overline{u}_1),\; \ldots,\; q_{n-1}^{\bullet}(\overline{u}_{n-1}),\; q_n^{\bullet}(\overline{u}_n) \end{array}$$

- let G be of the form $\leftarrow p_0(\overline{u})$, then $P_G^{\mathcal{M}}$ is extended with a seed (fact)

  $${}^{\bullet}p_0(\overline{u}) \;\leftarrow$$

$\square$

We will skip G in $P_G^{\mathcal{M}}$ if the initial call is not relevant.

The main idea behind Magic-sets is to perform bottom-up evaluation of $P_G^{\mathcal{M}}$ and in this way mimic top-down execution of P w.r.t. G. Atoms built of predicate symbols ${}^{\bullet}p$ and $p^{\bullet}$, that appear in the bottom-up semantics of $P_G^{\mathcal{M}}$ correspond respectively to calls and successes of p in the top-down execution of G. In the context of program analysis some precision can be lost, as the Magic-sets method does not keep track of links between a particular call and the corresponding success. For more in-depth discussion on the topic see [CDY94, DR94, Nil95, Bra00].

To see what predicate dependency graphs of magic programs look like, consider a clause of a program P:

$$p(\overline{u}_0) \;\leftarrow\; q_1(\overline{u}_1), \ldots, q_n(\overline{u}_n)$$

This gives $n+1$ clauses in $P^{\mathcal{M}}$ (cf. Definition 2.8) and thus $\frac{(n+1)(n+2)}{2}$ edges in the dependency graph of $P^{\mathcal{M}}$:

$$\begin{array}{lllllll} {}^{\bullet}p \to {}^{\bullet}q_1 & & & & & & \\ q_1^{\bullet} \to {}^{\bullet}q_2, & & {}^{\bullet}p \to {}^{\bullet}q_2 & & & & \\ \vdots & & \vdots & & & & \\ q_{n-1}^{\bullet} \to {}^{\bullet}q_n, & & q_{n-2}^{\bullet} \to {}^{\bullet}q_n, & \ldots, & {}^{\bullet}p \to {}^{\bullet}q_n, & & \\ q_n^{\bullet} \to p^{\bullet}, & & q_{n-1}^{\bullet} \to p^{\bullet}, & \ldots, & q_1^{\bullet} \to p^{\bullet}, & & {}^{\bullet}p \to p^{\bullet} \end{array}$$

$e := \mathbb{B}({}^\bullet h,\ d)$
for $i = 1$ **to** n **begin**
  $O^{m_i}(d) := \mathbb{C}({}^\bullet b_i,\ e)$
  $e := e \ \oplus\ \mathbb{B}(b_i^\bullet,\ d))$
end
$O^{m_{n+1}}(d) := \mathbb{C}(h^\bullet,\ e)$

Figure 7: Induced Magic-sets

2.4 Induced Magic-sets

Magic programs contain clauses whose bodies highly overlap each other. During bottom-up evaluation, this causes redundant computations of identical fragments of the clause bodies, which is an obvious weakness.

One solution to this problem is the Supplementary Magic transformation [Ram91]. A problem with Supplementary Magic is that for every clause whose body is of length n, we introduce n new *supplementary* predicates, which is often not profitable.

Another solution, presented by Michael Codish [Cod99] as Induced Magic-sets, is to process at the same time all the magic clauses originating from the same clause. The idea is best understood by investigating the definition of $O^{m_i}(d)$ for the magic clauses $m_1, \ldots, m_{n+1}$ generated by the single clause $h \leftarrow b_1, \ldots, b_n$.

$$\begin{array}{lcl}
O^{m_1}(d) & = & \mathbb{C}({}^\bullet b_1,\ \ \mathbb{B}({}^\bullet h, d)) \\
O^{m_2}(d) & = & \mathbb{C}({}^\bullet b_2,\ \ \mathbb{B}({}^\bullet h, d) \oplus \mathbb{B}(b_1^\bullet, d)) \\
O^{m_3}(d) & = & \mathbb{C}({}^\bullet b_3,\ \ \mathbb{B}({}^\bullet h, d) \oplus \mathbb{B}(b_1^\bullet, d) \oplus \mathbb{B}(b_2^\bullet, d)) \\
 & \vdots & \\
O^{m_{n+1}}(d) & = & \mathbb{C}(h^\bullet,\ \ \mathbb{B}({}^\bullet h, d) \oplus \mathbb{B}(b_1^\bullet, d) \oplus \mathbb{B}(b_2^\bullet, d) \oplus \cdots \oplus \mathbb{B}(b_n^\bullet, d))
\end{array}$$

By keeping the value of the second argument to $\mathbb{C}$ and using it when processing the next magic clause, redundant computation is avoided.

Codish also points out that this method makes it possible to skip the magic transformation altogether, since the original clause provides the information needed. Figure 7 shows the Induced Magic algorithm for computing the $O^{m_i}(d)$ values above. Note that the input to the algorithm is the original clause $h \leftarrow b_1, \ldots, b_n$.

Induced Magic-sets has been shown to be superior to Supplementary Magic regarding different criteria, such as analysis time and number of unification steps. See [Cod99] for details.

3 Call-success dependency graph

A dependency graph of a magic program has several properties that we exploit in order to calculate SCCs efficiently and thus to optimize program evaluation. The dependency graph for a magic program has a number of "redundant" edges, that correspond to the overlapping parts of magic clauses. Observe also, that it is the reduced dependency graph $\mathcal{R}(Dep(P))$ rather then $Dep(P)$, that is needed to guide the bottom-up evaluation.

In the following we define for a given program P, a *call-success dependency graph* that contains considerably fewer edges than the dependency graph for $P^{\mathcal{M}}$ and has properties that can be used to optimize the fixed point iteration.

Definition 3.1 A *call-success dependency graph* for a given program P (written as $CSDep(P)$) consists of

- a set of vertices $\{{}^{\bullet}p \mid p \in Pred(P)\} \cup \{p^{\bullet} \mid p \in Pred(P)\}$
- a set of edges which
  - for each unit clause $p(\overline{u}) \leftarrow$ includes the edge ${}^{\bullet}p \rightarrow p^{\bullet}$, and
  - for each non-unit clause $p(\overline{u}) \leftarrow q_1(\overline{u}_1), \ldots, q_n(\overline{u}_n) \in P$, includes the following edges:

$$\begin{array}{ccc} {}^{\bullet}p & \rightarrow & {}^{\bullet}q_1, \\ q_1^{\bullet} & \rightarrow & {}^{\bullet}q_2, \\ & \vdots & \\ q_{n-1}^{\bullet} & \rightarrow & {}^{\bullet}q_n, \\ q_n^{\bullet} & \rightarrow & p^{\bullet} \end{array}$$

□

Notice, that every clause from P contributes only with $n+1$ edges in $CSDep(P)$ (compared to $\frac{(n+1)(n+2)}{2}$ in $Dep(P^{\mathcal{M}})$). Before we state the main result of this section we introduce an auxiliary definition.

Definition 3.2 A predicate p is *useful* iff there is a clause $p(\overline{u}) \leftarrow q_1(\overline{u}_1), \ldots, q_n(\overline{u}_n)$ $n \geq 0$, where all $q_1, \ldots, q_n$ are useful. (So each predicate occurring in a unit clause is useful.)

□

Proposition 3.3 If a predicate p is useful then ${}^{\bullet}p \xrightarrow{+}_{CSDep(P)} p^{\bullet}$.

PROOF: By structural induction on definition 3.2.
If there is a unit clause $p(\overline{u}_0) \leftarrow\ \in P$ then the property holds. Otherwise there must be a clause $p(\overline{u}_0) \leftarrow q_1(\overline{u}_1), \ldots, q_n(\overline{u}_n) \in P$, with $q_1, \ldots, q_n$ $(n > 0)$ being useful. By inductive assumption we get ${}^{\bullet}q_i \xrightarrow{+}_{CSDep(P)} q_i^{\bullet}$. Combining this with definition 3.1 we obtain the following path in $CSDep(P)$:

$${}^{\bullet}p \rightarrow {}^{\bullet}q_1 \xrightarrow{+} q_1^{\bullet} \rightarrow \cdots \rightarrow {}^{\bullet}q_n \xrightarrow{+} q_n^{\bullet} \rightarrow p^{\bullet}$$

This completes the proof.

Proposition 3.4 For a given program P, such that all its predicates are useful, the following holds

$$\mathcal{R}(Dep(P^{\mathcal{M}})) = \mathcal{R}(CSDep(P))$$

PROOF: By combining Proposition 3.3 and Definition 3.1 we can reconstruct all the paths of $Dep(P^{\mathcal{M}})$, starting from $CSDep(P)$. Those paths fully determine $\xrightarrow{+}$ and $\Rightarrow$ of both graphs.

The above theorems show that under certain conditions we can employ $\mathcal{R}(CSDep(P))$ to guide the bottom-up evaluation of $P^{\mathcal{M}}$, just in the same way as SCCs of a dependency graph are used to navigate the ordinary bottom-up evaluation.

Example 3.5 Let P be a program

$$\begin{array}{lll} a([\,]) & & \\ a([X|Y]) & \leftarrow & b(X), a(Y), c(X) \\ b(1) & & \\ c(1) & & \end{array}$$

Then $P^{\mathcal{M}}$ contains the following clauses:

$$\begin{array}{lll} a^\bullet([\,]) & \leftarrow & {}^\bullet a([\,]) \\ {}^\bullet b(X) & \leftarrow & {}^\bullet a([X|Y]) \\ {}^\bullet a(Y) & \leftarrow & {}^\bullet a([X|Y]), b^\bullet(X) \\ {}^\bullet c(X) & \leftarrow & {}^\bullet a([X|Y]), b^\bullet(X), a^\bullet(Y) \\ a^\bullet([X|Y]) & \leftarrow & {}^\bullet a([X|Y]), b^\bullet(X), a^\bullet(Y), c^\bullet(X) \\ b^\bullet(1) & \leftarrow & {}^\bullet b(1) \\ c^\bullet(1) & \leftarrow & {}^\bullet c(1) \end{array}$$

The dependency graphs are shown below

$$\begin{tikzcd}
{}^\bullet a \arrow[loop left] \arrow[r] \arrow[d] \arrow[rr, bend left] & a^\bullet \arrow[loop above] \arrow[r] & {}^\bullet c \arrow[d] \\
{}^\bullet b \arrow[r] & b^\bullet \arrow[ul] \arrow[u] \arrow[ur] \arrow[r] & c^\bullet \arrow[ul]
\end{tikzcd}
\qquad
\begin{tikzcd}
{}^\bullet a \arrow[r] \arrow[d] & a^\bullet \arrow[r] & {}^\bullet c \arrow[d] \\
{}^\bullet b \arrow[r] & b^\bullet \arrow[ul] & c^\bullet \arrow[ul]
\end{tikzcd}$$

$Dep(P^{\mathcal{M}})$ $\qquad$ $CSDep(P)$

Both graphs have the same reduced graph that is

$$\{{}^\bullet a, {}^\bullet b, b^\bullet\} \Rightarrow \{a^\bullet, {}^\bullet c, c^\bullet\}$$

Observe that $Dep(P^{\mathcal{M}})$ contains a lot of redundant edges, that could be discarded when computing SCCs.

Next we show that our technique never turns recursive components into non-recursive ones. This is important if we apply an optimization based on the fact that the fixpoint computation of non-recursive components will surely terminate. In such cases enforcing termination by using a widening function may be skipped, as the fixed point of a non-recursive component is found in one iteration. The problem concerns obviously only singleton recursive SCCs (i.e. those containing only one vertex connected to itself), as only one-element SCCs can be non-recursive, and as stated in Proposition 3.4 our technique preserves the structure of reduced graphs (provided that all the predicates in the program are useful).

Proposition 3.6 Let P be a program with all predicate useful. If $\{v\} \in SCC(Dep(P^{\mathcal{M}}))$ and $v \rightarrow_{Dep(P^{\mathcal{M}})} v$ then $v \rightarrow_{CSDep(P)} v$ (and $\{v\} \in SCC(CSDep(P)))$.

PROOF: If v is of the form ${}^\bullet a$ then there must be a clause $a(\overline{u}) \leftarrow a(\overline{u}_0), q_1(\overline{u}_1), \ldots, q_n(\overline{u}_n) \in P$ with all $q_i \neq a$, $1 \leq i \leq n$ (otherwise more elements would belong to the SCC). Thus by the definition 3.1 there is the edge ${}^\bullet a \rightarrow {}^\bullet a$ in $CSDep(P)$. Similarly, if v is of the form $a^\bullet$, then there must be a clause $a(\overline{u}) \leftarrow q_1(\overline{u}_1), \ldots, q_n(\overline{u}_n), a(\overline{u}_{n+1}) \in P$, s.t. $q_i \neq a$, $1 \leq i \leq n$. Hence, there is the edge $a^\bullet \rightarrow a^\bullet$ in $CSDep(P)$.

If a program contains non-useful predicates, the structure of reduced dependency graphs may not be preserved, i.e. $\mathcal{R}(Dep(P^{\mathcal{M}})) \neq \mathcal{R}(CSDep(P))$. This happens if a path ${}^\bullet p \xrightarrow{+} p^\bullet$ is missing for some p, which means that the predicate p is not useful and will never succeed. In fact, this is a rather odd case and may indicate a bug in the program. In particular it happens if p is not defined in P. We can discover this situation at the early stage of analysis and just abandon iteration over SCCs that have no edge incoming from some other component (with an exception of the component containing the initial call).

Example 3.7 Let P be a program

$$
\begin{array}{lll}
a(X) & \leftarrow & b(X), a(X) \\
b(X) & &
\end{array}
$$

It is clear that a never succeeds. The dependency graphs are

$$
\begin{tikzcd}
{}^\bullet a \arrow[loop above] \arrow[r] \arrow[d] & a^\bullet \arrow[loop right] \\
{}^\bullet b \arrow[r] & b^\bullet \arrow[ul] \arrow[u]
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
{}^\bullet a \arrow[d] & a^\bullet \arrow[loop right] \\
{}^\bullet b \arrow[r] & b^\bullet \arrow[ul]
\end{tikzcd}
$$

$Dep(P^{\mathcal{M}})$ $\qquad\qquad$ $CSDep(P)$

In both cases we have two SCC. Evaluation navigated by $Dep(P^{\mathcal{M}})$ will be iterating over $\{a^\bullet\}$. Whereas in $CSDep(P)$ this component has no incoming node and therefore will be omitted during the iteration.

It is also possible to achieve complete equivalence between $\mathcal{R}(Dep(P^{\mathcal{M}}))$ and $\mathcal{R}(CSDep(P))$, that means dropping the condition that every predicate is useful, in proposition 3.4. This could be done just by adding the edge ${}^\bullet h \to h^\bullet$ for each clause in the definition of $CSDep(P)$.

4 Combining Induced Magic-sets with SCCs

Having showed an efficient method to compute SCCs of $P^{\mathcal{M}}$ we proceed by describing how to use them together with the Induced Magic-sets technique.

The original Induced Magic-sets algorithm computes together all magic clauses generated by a single clause in the original program. These clauses, however, might define predicates in more than one SCC. Thus, computing them together would mean that we lose some of the efficiency of the SCC method.

To combine the two methods, we use a slightly different version of Induced Magic-sets. We will still be able to process the original program rather than the magic clauses generated from it, but we need to know how the overlapping magic clauses are distributed over the SCCs, and this information is easily gathered when the SCCs are constructed.

4.1 Decorated call-success dependency graph

In order to keep track on the program structure, we decorate the call-success dependency graph in the following way.

Definition 4.1 Let P be a program and let every clause in P have a unique label. A *decorated call-success dependency graph* of P is a graph $CSDep(P)$ where each vertex v is associated with a set of labels L_v constructed as follows:

For every clause $p(\overline{u}_0) \leftarrow q_1(\overline{u}_1), \ldots, q_n(\overline{u}_n)$, with a label c the vertices are decorated so that:

- $(c.i) \in L_{{}^\bullet q_i}$ for $1 \leq i \leq n$
- $(c.n{+}1) \in L_{p^\bullet}$

□

The intuition of the labels is that the label $(c.i) \in L_a$ corresponds to the ith magic clause (whose head is $a(\overline{u})$) generated from clause c, assuming that the magic clauses are ordered as in Definition 2.8. Thus, L_a describes the magic clauses defining a.

Next, the information of these labels should be passed up to the SCCs. This is done by merging the labels originating from the same clause.

Definition 4.2 For a given program P, every $\mathcal{C} \in SCC(CSDep(P))$ is assigned a set $\mathcal{L}_\mathcal{C}$ of labels on the form $(c.from.to)$ where:

- $from = min\{i \mid (c.i) \in \bigcup_{v \in \mathcal{C}} L_v\}$
- $to = max\{i \mid (c.i) \in \bigcup_{v \in \mathcal{C}} L_v\}$

□

The intention of decorating SCCs is to locate those parts of the original program clauses that are relevant for a given SCC. If the label $(c.i.j)$ is associated with the SCC $\mathcal{C}$ it means that clause c in the original program generates magic clauses defining predicates in $\mathcal{C}$. Furthermore, only the magic clauses number i to j do.

Example 4.3 Consider the following "naive reverse" program:

(1) $nrev([\,],[\,])$
(2) $nrev([X|Xs],R) \leftarrow$
$\quad nrev(Xs,R1),$
$\quad app(R1,[X],R)$

(3) $app([\,],Ys,Ys)$
(4) $app([X|Xs],Ys,[X|Zs]) \leftarrow$
$\quad app(Xs,Ys,Zs)$

Its call-success dependency graph, with decorated vertices, is shown below:

$$\begin{tikzcd}
\underset{\{(2.1)\}}{{}^{\bullet}nrev} \arrow[loop left] \arrow[r] & \underset{\{(1.1),(2.3)\}}{nrev^{\bullet}} \arrow[dl] \\
\underset{\{(2.2),(4.1)\}}{{}^{\bullet}app} \arrow[loop left] \arrow[r] & \underset{\{(3.1),(4.2)\}}{app^{\bullet}} \arrow[u] \arrow[loop right]
\end{tikzcd}$$

The graph has two SCCs, labeled as follows:

$$\underset{\{(2.1.1)\}}{\{{}^{\bullet}nrev\}} \quad \Longrightarrow \quad \underset{\{(1.1.1),(2.2.3),(3.1.1),(4.1.2)\}}{\{nrev^{\bullet}, {}^{\bullet}app, app^{\bullet}\}}$$

The next section will explain in more detail how this information is used during evaluation. For this we need the following proposition:

Proposition 4.4 Consider a program P (containing only useful predicates) and a decorated set $SCC(CSDep(P))$. If there are two different SCCs $\mathcal{C}_1, \mathcal{C}_2 \in SCC(CSDep(P))$, s.t. $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$, $(c.i_1.j_1) \in \mathcal{C}_1$, $(c.i_2.j_2) \in \mathcal{C}_2$ then $j_1 < i_2$.

PROOF: Let assume that $j_1 = i_2$. This means that one predicate belongs to two different components, which is a contradiction. Next assume that $j_1 > i_2$. Let c be $p(\overline{u}) \leftarrow q_1(\overline{u}_1), \ldots, q_n(\overline{u}_n)$. Clearly, there must be a path from ${}^{\bullet}q_{i_2}$ to ${}^{\bullet}q_{j_1}$ (or to $p^{\bullet}$ if $j_1 = n+1$) in $CSDep(P)$. This contradicts the assumption that $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ and that $\mathcal{C}_1$ and $\mathcal{C}_2$ are SCCs.

4.2 Evaluation algorithm

The purpose of constructing labeled SCCs of a program was to be able to combine efficiently the SCC approach with the Induced Magic-set way of computing overlapping magic clauses together. As in the original SCC method, each SCC is evaluated individually, respecting the $\Rightarrow$ ordering. The difference is that the labels give us information about how to compute some magic clauses together, within each SCC, in a way similar to the Induced Magic-sets method.

When processing a SCC $\mathcal{C}$, the labels $\mathcal{L}_{\mathcal{C}}$ and the original program P contain all information needed. During each iteration step all labels in $\mathcal{L}_{\mathcal{C}}$ are considered. A label $(c.i.j)$ indicates that the magic clauses m_i, m_{i+1}, $\ldots$, m_j generated from clause $c \in P$ define a predicate in $\mathcal{C}$. Since these magic clauses overlap, they can be computed together. A simple way of doing this is by traversing the original clause c, much as in the Induced Magic-sets method.

Figure 8 shows the algorithm for Induced Magic-sets restricted to a labeled SCC. Given a set of labels $\mathcal{L}_{\mathcal{C}}$ and a domain element d, the algorithm ends with d$'$ containing the value of $O^M(d)$ (M being the subset of $P^{\mathcal{M}}$ that defines predicates in $\mathcal{C}$). As in the original Induced Magic-set algorithm, the magic clauses are never explicitly generated.

Let c be a label of the clause $h \leftarrow b_1, \ldots, b_n$. For each label $(c.i.j)$, the algorithm traverses the head and the $j-1$ first body literals of c. Before reaching the ith literal, no new facts are generated, since the corresponding magic clauses do not belong to the current SCC. Proposition 4.4 tells that for all $x \leq j-1$, $b_x^{\bullet}$ either have been already computed, or is being computed during the current SCC iteration.

The conditional statement after the loop is needed to take care of the special case when $j = n+1$, since the head of the last magic clause is different from the others.

Example 4.5 Consider the program from example 4.3. During the iteration over the first SCC, the second original clause will be used to gather

$d' := \{\}$
foreach $(c.i.j)$ **in** $\mathcal{L}_{\mathcal{C}}$ **do**
    (c is the label of $h \leftarrow b_1, \ldots b_n$)
    $e := \mathbb{B}(^\bullet h,\ d)$
    for $x = 1$ **to** $j - 1$ **do**
        if $(x \geq i)$ **then** $d' := d' \cup \mathbb{C}(^\bullet b_x,\ e)$
        $e := e\ \oplus \mathbb{B}(b_x^\bullet,\ d))$
    end for
    if $(j < n + 1)$ **then** $d' := d' \cup \mathbb{C}(^\bullet b_j,\ e)$
    else $d' := d' \cup \mathbb{C}(h^\bullet,\ e)$
end foreach

Figure 8: Induced Magic-sets restricted to a labeled SCC.

facts about $^\bullet nrev$, because of the label (2.1.1). Each iteration step would look like this:

$$\begin{array}{lll} e & := & \mathbb{B}(^\bullet nrev([X|Xs], R),\ d) \\ d' & := d' \cup & \mathbb{C}(^\bullet nrev(Xs, R1),\ e) \end{array}$$

In the second SCC, the same original clause will be used to gather facts about $^\bullet app$ and $nrev^\bullet$, because of the label (2.2.3). During each iteration step, the label will be handled as follows:

$$\begin{array}{lll} e & := & \mathbb{B}(^\bullet nrev([X|Xs], R),\ d) \\ e & := e\ \oplus & \mathbb{B}(nrev^\bullet(Xs, R1),\ d) \\ d' & := d' \cup & \mathbb{C}(^\bullet app(R1, [X], R),\ e) \\ e & := e\ \oplus & \mathbb{B}(app^\bullet(R1, [X], R),\ d) \\ d' & := d' \cup & \mathbb{C}(nrev^\bullet([X|Xs], R),\ e) \end{array}$$

Notice that in the first SCC the traversal of the clause stops before the end of the clause is reached. In the second SCC, the first body literal is only used to create variable bindings, and not for creating new facts. This way of restricting the traversal to the current SCC is what makes the algorithm different from the original Induced Magic-sets method.

5 Benchmarks

We have done some experiments concerning computing SCCs and applying SCC-driven strategy to induced magic-sets. All the tests we performed running SICStus Prolog version 3.8.4 on Sun-Ultra 10/440, with 440 MHz cpu speed and 265 MB RAM.

5.1 Computing SCCs

In Table 1 we present experimental results illustrating an advantage of using $CSDep(P)$ and thereby reducing the overhead of computing SCCs. Times presented (in milliseconds) include only computing SCCs of the respective graphs. The figures do not include reading the programs from files and, in the $Dep(P^{\mathcal{M}})$ case, performing magic transformation. SCCs have been computed using SICStus Prolog `ugraphs` library [SIC98].

Program P	Vertices	Edges		Time	
		$Dep(P^{\mathcal{M}})$	$CSDep(P)$	$Dep(P^{\mathcal{M}})$	$CSDep(P)$
`cs_r.pl`	72	230	110	40	20
`disj_r.pl`	62	348	101	27	10
`ga.pl`	76	256	99	33	23
`gabriel.pl`	40	120	67	13	7
`kalah.pl`	90	298	145	37	20
`meta.pl`	18	51	26	3	3
`nand.pl`	86	386	150	57	30
`nandc.pl`	28	101	38	13	7
`neural.pl`	78	289	129	37	20
`sumfac.pl`	12	45	21	3	3
`semi.pl`	44	158	61	27	10
`ronp.pl`	32	108	55	10	7
`qplan.pl`	96	480	197	67	33
`press.pl`	102	334	185	53	37
`peep.pl`	42	117	77	33	27
`treeorder.pl`	12	31	22	3	3
`trs.pl`	74	334	160	60	27
`tsp.pl`	48	188	67	13	7

Table 1: Computing SCCs for $Dep(P^{\mathcal{M}})$ and $CSDep(P)$

The number of edges in $CSDep(P)$ is considerably less than in the $Dep(P^{\mathcal{M}})$ case. This is interesting as the standard algorithm for computing SCC [Tar72] has a linear time complexity w.r.t. number of vertices and edges. This fact is clearly reflected by the columns containing execution times. In the experiments the $CSDep(P)$ contains 53% fewer edges and SCCs are found in 37% shorter time, in average.

5.2 Type analysis

In order to assess the impact of applying SCC to the Induced Magic-sets technique, we have implemented a prototype system that infers descriptive call-success types of a given program. The basic operations on the type domain are taken from [GdW94, DP99]. The tests were performed using semi-naive evaluation strategy.

Program	No. of SCCs	Analysis time	
		No SCC	SCC
`serialize.pl`	10	630	560
`nqueens.pl`	4	290	240
`sumfac.pl`	6	970	780
`qsort.pl`	1	380	380
`cs_r.pl`	23	13300	7850
`disj_r.pl`	19	4940	4060
`dsf.pl`	3	3970	3910
`press.pl`	7	12660	12550
`gabriel.pl`	14	1740	1200
`kalah.pl`	17	11620	10910
`neural.pl`	17	1910	1640
`semi.pl`	9	4680	4630
`ronp.pl`	26	1710	990
`qplan.pl`	8	9560	8840
`peep.pl`	13	4690	4050
`treeorder.pl`	5	290	270
`tsp.pl`	16	4750	2980
`zebra.pl`	6	820	880

Table 2: Type analysis

The results are displayed in Table 2. The first column contains a name of a program P, the second one the numbers of SCCs in $P^{\mathcal{M}}$. The next two columns present times (in milliseconds) of evaluating programs without, and with splitting them into SCCs. Times include reading program files and type inference (thus in the case of SCC-based optimization they include computing SCCs as well). In almost all cases we observe improvement of efficiency. The average evaluation speed-up is 15% (but the maximum is 41%).

The number of SCCs remains in small correlation with the speed-up of analysis. This is due to the fact that in the presence of recursive procedures, predicates of $P^{\mathcal{M}}$ sometimes tend to group into a single big SCC, and few small ones (see [Car00]). In particular, it seems, that in the case of the `zebra.pl` program, predicates did not distribute over SCCs equally enough to compensate the overhead of computing SCCs.

6 Concluding remarks

We have demonstrated a novel method to compute the strongly connected components (SCCs) of a logic program which is the result of a Magic-set transformation. Such programs are characterized by a high degree of overlapping clauses, and the method exploits this fact by building a reduced predicate dependency graph directly from the original program. We have shown that the SCCs of this reduced graph is consistent with those computed in the ordinary way.

Further, we have shown how a labeled dependency graph can be used when evaluating this type of programs bottom-up. Using information from the construction of the dependecy graph, the Induced Magic-sets method can be restricted to clauses (and parts of clauses) that are significant to a single SCC. Thus, we can avoid recomputation of overlapping parts of clauses, whithout loosing the efficiency gained by the SCC approach.

Experimental data show the efficiency of both methods.

Paper V

The TELL diagnosis tool

Paweł Pietrzak

1 Introduction

Constraint logic programming (CLP) is a family of untyped programming languages. Programming practice indicates however that the user has usually "types" in mind when writing a CLP program. Moreover, certain types of parameters are often required by builtin procedures. This paper describes TELL (**T**ype-based **E**rror **L**ocator for constraint **L**ogic programs), a tool that finds type errors in untyped CLP programs.

TELL combines verification techniques together with a language of types, as a specification language. The tool verifies a CLP program w.r.t. a (type) specification given by the user and locates errors if the verification fails. When providing the specification the user may take advantage of results of static analysis (type inference). The results of the analysis can be seen as a draft for the specification being created. The process of writing specification goes interactively. As soon as this is possible, fragments of program clauses are verified. Thus it is likely that the user does not have to provide the entire specification in order to locate a bug. Our approach is based on directional types that make a difference between call and success types. The theoretical principles of TELL, as it was evolving, have been described in [DMP00b, DMP02].

TELL is designed to handle a family of Prolog-like CLP languages. Its architecture makes it possible to easily adjust TELL to a specific dialect of CLP. The current version of TELL handles a considerable subset of SICStus Prolog [SIC98] including finite domain and boolean constraints. We also have made experiments with a version suited for the programming language CHIP [Cos98].

In this paper we describe the architecture of TELL and explain how to use the TELL diagnosis tool to locate errors. In particular, we explain how to add definitions of types, how to write specifications of programs, how to write specifications of new libraries and extend existing ones. It is also shown how to use the GUI during diagnosis sessions. Some implementation issues are discussed. Formal aspects of our approach, i.e. the verification and analysis methods, and the specification language are described elsewhere ([DMP00b, DMP00a, DMP02]). We assume that the reader is familiar with these papers.

In [DMP00a] we have outlined the previous version of the diagnosis tool (as well as some formal issues of the method). After experimenting with the tool, it became clear to us what functionality was missing and what the architecture of the tool should be. This paper discusses our new view of the diagnosis tool. The paper should be read as a design proposal rather than a description of an existing system, since not all of the functions described are implemented now.

The plan of the paper is as follows. In Section 2 we briefly describe the structure of the tool. Section 3 recalls the notion of (descriptive) types and introduces the syntax that is used by TELL. Section 4 explains how to write specifications. Section 5 addresses some issues related to full Prolog and explains how to handle other constraint domains. The architecture of TELL is discussed in Section 6. Features of the user interface are presented in Section 7. We summarize with some conclusions.

2 Overview of TELL

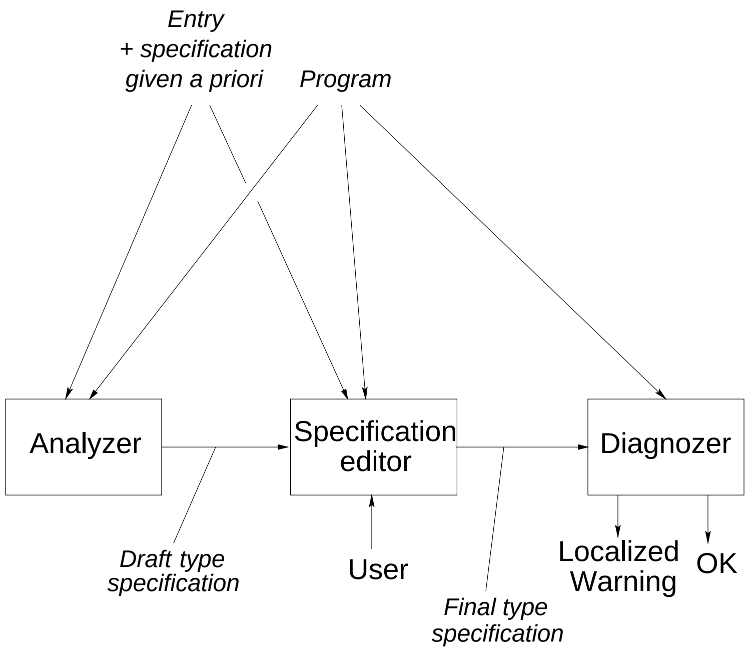

Figure 9: The structure of TELL

The TELL system is built of three major components as depicted in Figure 9:

- **Analyzer** - takes a program and an entry point, i.e. information of the initial calls to the program, and computes (statically) call and success types which approximate the program's semantics; the analyzer uses a specification given a priori to determine types of builtin, library and undefined predicates,

- **Specification editor** - facilitates writing specification; uses the result of the analysis (if it has been performed) as an initial draft of the specification; also uses (a part of) a specification possibly given a priori,

- **Diagnozer** - performs verification of the program w.r.t. the specification and locates errors.

A diagnosis session begins with loading the program which may but need not be analyzed by the analyzer. Then the user, working interactively with the specification editor, constructs the specification. During that process the editor invokes the diagnozer in order to verify the clauses, for which it is possible, i.e. a sufficient part of the specification is provided. The session ends with locating an error, if there is one.

Algorithms used in the components of TELL are described in [DMP02]. The architecture of TELL is further described in Section 6.

3 Types

In our approach we use *descriptive types*. This means that the types approximate the semantics of an untyped program. We consider call-success semantics (with the left-to-right selection rule) (cf. [DMP02] Section 2.3), thus the types are used to describe both call and success patterns. The types are sets of (constrained) terms, as described in Section 2.2 of [DMP02]. We distinguish between *base types* like integers or reals, and *compound types* that are defined by regular term grammars. This formalism makes it possible to specify commonly used types such as lists or trees, and also application specific types. Compound types may be combined with base types (e.g. to define the type consisting of lists of integers). In this section we describe how to define types, how they are stored by TELL, and how they are used. We refer to [DMP02] for further details.

3.1 Base types

The notion of base types is defined in [DMP02], Section 2.2.2. The base types are built into the system in form of definitions kept in a separate file. Every base type is represented by a unique type symbol, called also a type name. In the following description, sample base type will be used: `int` - integers, `float` - float numbers, `anyfd` - finite domain variables[43], `atom` - Prolog atoms. Additionally, the type `any` ($\top$ in [DMP02]), which is a supertype of any type, is always present in the system. The file with base type definitions includes:

[43]It differs from `anyfd` from [DMP02] since finite domains in SICStus range over integer number rather than natural ones.

- the declarations of base type symbols, as illustrated by the following example:

```
base_type(int).
base_type(float).
base_type(anyfd).
base_type(atom).
```

- the declaration of inclusion relations between the base types, e.g. :

```
int < anyfd.
```

  These declarations should satisfy the requirement that every two distinct base types are either disjoint or one is included in the other (Requirement 2.13 of [DMP02]). Otherwise the type algorithms may produce wrong results. For instance, the algorithm computing an upper bound of two types might produce a non-discriminative grammar.

- Prolog procedures for membership testing of every base type:

```
is_in_base(anyfd,C) :- integer(C).
is_in_base(int,C) :- integer(C).
is_in_base(float,C) :- float(C).
is_in_base(atom,C) :- atom(C).
```

  This test is crucial in computing, for instance, an intersection between `int` and a finite type containing integer constants.

This file can be extended with new base types.

3.2 Defining compound types

Besides base types there are compound types defined by parametric type grammars, which are introduced in [DMP02], Section 4.3. Likewise base types, non-base type symbols are written with a lower case letter. A type definition begins with a keyword `typedef` followed by the left hand side of the grammar rule. On the right of `--->` there is a right hand side of the rule. It can be a constant, a function symbol with type names as arguments[44], or a base type name (i.e. is prohibited to write a r.h.s. like `f(blah)`, where `blah` is a constant from the program, not a type name). Alternatively one can write many right hand sides of the rule(s) separated by `;`. For example a grammar defining the type of binary trees with integers in nodes would look like:

[44]We dropped a restriction of Definition 4.5 of [DMP02], where the base type symbols are disallowed under a function symbol in type definitions.

```
typedef inttree ---> void.
typedef inttree ---> t(int,inttree,inttree).
```

or equivalently:

```
typedef inttree ---> void; t(int,inttree,inttree).
```

Type grammars may contain parameters that have to be written with capitals as Prolog variables:

```
typedef list(A) ---> []; [A|list(A)].
typedef tree(A) ---> void; t(A, tree(A), tree(A)).
```

In some circumstances base type name might be mistaken with constants of the programming language. It happens if a base type name is a top-most symbol in the right hand side of the grammar rule, i.e. if it appears where a constant may appear. In theory it is possible to make an assumption that alphabets of a program and type names are disjoint. In practice, symbols like `int` or `float` may appear in the program. To avoid confusion the base type names are to be prefixed by `@`.

For instance, type grammars can be used to define types that are unions of two or more base types:

```
typedef number ---> @int; @float.
typedef atomic ---> @atom; @int; @float.
```

Without `@`, the type `number` would be understood as containing just two constants `int` and `float`.

3.3 Role of type names

We have shown how to define base and compound types. The defined types are given names. TELL uses type names and corresponding definitions for various purposes:

- to refer to the available type definitions before and during a diagnosis session; the type definitions are used by the analyzer and diagnozer; the user may inspect them at any time,

- to display results of analysis - whenever the inferred type is equal to a type defined in the system, then the name of the defined type is displayed; otherwise, the analyzer displays the newly constructed type (with artificial unique name) together with its definition,

- to display type information in an error message.

3.4 Storing type definitions in TELL

TELL supports a layered memory of type definitions. The layers differ in the policy of loading type definitions and keeping them during a diagnosis session. Five layers of definitions are distinguished:

standard definitions - they include base types and all the types that are needed to specify builtin procedures (but not library procedures). They are built into the system and are linked with the language to be analyzed/verified. Standard types are defined together with the specification for builtin predicates, and should be modified / extended by a person who writes this specification. Standard types are present in TELL permanently.

library-specific definitions - they contain types needed in specifications of library predicates. (This specification may use standard types as well.) Library-specific type definitions come together with the specification of the library. They are loaded to TELL on demand, i.e. only if a relevant library is used by the program being diagnosed. They are retracted from the TELL's type memory when the diagnosis session is over.

user's definitions - this is a collection of commonly used types defined by the user. The types are used in writing specification of programs but are not specific to a particular program. They are (re)loaded at the beginning of every session. Another option is to load them and discard on request.

program-specific definitions - these types are application-specific types and come with specifications for individual programs. (Specifications of programs may also use standard, library-specific and user's types.) Program-specific types are defined in a program specification, and therefore are written by the user/programmer. These types are present in TELL only during a diagnosis session with the given program.

synthesized definitions - they are results of the operations on types which are performed mainly during analysis, but also during verification. They are stored in the system temporarily (some of them are kept only in a single iteration of the analysis). The synthesized types that are in the final result of the analysis may become a part of a specification. In this case they become program-specific types.

Normally, only synthesized types are displayed together with their definitions. Otherwise, TELL displays just type names, as it is assumed that the user is familiar with non-synthesized types and does not have to see their definitions. He may however have a look at the definition of any type, upon request.

Whenever possible a synthesized type is replaced with a type known to the user. When looking up a candidate for the replacement TELL checks the type groups in the following order: program-specific types, user's types, library-specific types and eventually standard types. The idea behind this strategy is that the types defined by the user should be preferred over standard or library ones, likely defined by someone else.

TELL requires type names to be unique. While loading type definitions, TELL rises a warning if a conflict in type names occurs, (i.e. if there are two or more definition of a type with the same name) and enforces renaming.

4 Writing specifications

4.1 Specifications of predicates

Types that have been introduced in the previous section are used to specify predicates. For every predicate we have to specify its call and success types (i.e. call and success patterns). They have a form:

Call-Type `=>` *Success-Type.*

The specification of a predicate should be read as an implication: whenever the predicate's call has a form that belongs to *Call-Type* then its success is in *Success-Type*, provided that the call succeeded. For example:

```
qsort(list(number),any) => qsort(list(number),list(number))
```

says that the predicate `qsort/2` should be called with first parameter bound to a list of numbers and second to any term. Type `any` is an implicit way of referring to calls including variables, since this information cannot be explicitly expressed in the type definition formalism. Upon success both arguments are expected to be lists of numbers.

If a specification of a predicate contains parameters then every parameter in the success type must appear in the call type. It is necessary in order to determine the value of the parameter as the predicate is called.

4.2 Specifications of builtins and SICStus libraries

A typical program makes use of builtin predicates and libraries. Therefore, the analyzer and the diagnozer must be given information about call and success types of the builtin and library predicates. This information is provided by specifications. The specification of builtin predicates is always present in the system. The specification of a library *lib* is loaded whenever a directive

```
:-use_module(library(lib))
```

is encountered in the SICStus program. Similar directives occur in other CLP dialects, for which the specification loading procedure should be adjusted. By convention, a name of a file containing the specification is *lib*`.spec.pl`.

A builtin specification, as well as a library specification, consists of call and success types for predicates of the library. They are usually preceded by library-specific type definitions. We illustrate this by a fragment of the specification of SICStus builtin predicates. It includes the following type definition for arithmetic expression.

```
typedef expr --->
     number;
     - expr;
     expr + expr;
     expr - expr;
     expr * expr;
     expr / expr;
     expr // expr;
     expr mod expr.
```

and specification of calls and success types, as illustrated below. The declaration

```
any is expr => number is expr.
```

says that we expect an expression on the right hand side of `is/2` when a call occurs, and any term on the left hand side (a number or a variable is expected but in our type language we have to approximate it by `any`) which upon success will be bound to number.

```
length(any,any) => length(list(any),nat).
```

The above specification reflects behavior of the predicate `length/2`, as if it succeeds the first parameter is a list and the second is its length (a natural number), regardless of the call pattern.

```
var(any) => var(any).
```

This is a test for checking if an argument, which can be anything, is a variable.

```
float(any) => float(float).
```

This is a test for checking if an argument, which can be anything, is a float number. It succeeds iff it is a float number.

```
see(atom) => see(atom).
```

This predicate opens an input file, whose name is a Prolog atom.

```
seen => seen.
```

That predicate closes the current input file, has no parameters.

A specification of libraries has the same form as one for builtins. Below we show a part of a specification of the `clpfd` library of SICStus Prolog.

```
typedef fd_lin_expr --->
     @anyfd;
     int * anyfd;
     int * int;
     fd_lin_expr - fd_lin_expr;
     fd_lin_expr + fd_lin_expr.

typedef fd_expr --->
     @anyfd;
     fd_expr + fd_expr;
     fd_expr - fd_expr;
     fd_expr * fd_expr;
     fd_expr / fd_expr;
     fd_expr mod fd_expr;
     min(fd_expr) ;
     max(fd_expr) ;
     abs(fd_expr).
```

The types `fd_lin_expr` and `fd_expr` denote, respectively, finite domain linear expressions and finite domain expressions. These definitions are used in declaring types for the constraint predicates such as equality and disequality:

```
fd_expr #\= fd_expr => fd_expr #\= fd_expr.
fd_expr #= fd_expr => fd_expr #= fd_expr.
```

As a next example, consider the specification of the predicate `labeling/2`. Its first argument has to be a list of options that specify search strategy. In order to reflect this in the specification a type `labeling_option` is defined:

```
typedef labeling_option ---> leftmost; min; max; ff; ffc;
     step; enum; bisect; up; down; all.
```

The specification of `labeling/2` looks as follows:

```
labeling(list(labeling_option),list(anyfd)) =>
     labeling(list(labeling_option),list(int)).
```

On call, the first argument is supposed to be a list of the options, whereas the second one is a list of finite domain variables. Upon success finite domain variables are instantiated to integer values. This is reflected by the type `list(int)` of the second argument in the success type.

It is unrealistic to assume that a complete specification of builtins and libraries will be ever provided. Thus the system is flexible so it allows incomplete specification. If types of a builtin or library predicate are not given then the system takes type `any` for all the arguments, both in call and success types. This results in a less precise, yet still correct, analysis and verification. Another option would be to rise a warning with a request to update the specification.

4.3 Specifications of programs

A specification of the program has the same parts as one for library or for builtins. It may contain type definitions and has to specify call and success types for the predicates in the program. Additionally the specification of the program includes an `entry` declaration which says what the top goals are. For instance, assume we have a quicksort program, whose main predicate is `quicksort/2` which takes an unsorted list of numbers in the first argument and a free variable in the second one. This is reflected in the following entry point declaration:

```
entry quicksort(list(number),any).
```

Even though the entry declaration is used mainly by the analyzer, it becomes a part of the specification - it is a call type of the top goal predicate.

The system can handle programs which are not completely implemented. In this case the specification should contain type information about predicates that are not defined in the program. If this information is not present in the specification (or if no specification was given) then TELL asks the user to input the missing types. Another possibility is to treat these predicates as builtins with no specification, i.e. to assume types `any` for all of their arguments.

Specifications of programs may be written before the diagnosis, and may be constructed during an interactive diagnosis session as well (cf. Figure 9).

4.4 Role of specifications in analysis and diagnosis

Different parts of specification play different roles during analysis and diagnosis. While analyzing the program, the specification provides information about success types of builtins, predicates defined in libraries or predicates not defined yet. If the success type is parametric, then the call type is used to determine a valuation of the parameters. Consider a slowsort program:

```
:-use_module(library(lists)).

slowsort(L,S) :- permutation(L,S), sorted(S).

sorted([]).
sorted([_]).
sorted([A,B|T]) :-
     A =< B,
     sorted([B|T]).
```

together with the entry declaration:

```
entry slowsort(list(number),any).
```

and the following line in the specification of the `lists` library:

```
permutation(list(A),any) => permutation(list(A),list(A)).
```

Then, during the analysis:

1. the entry point is used to determine the types of `L` and `S` just after calling `slowsort(L,S)`,

2. `permutation/2` is recognized as being imported from the library `lists`,

3. the computed call for `permutation/2`, which is `permutation(list(number),any)` is now matched against the call type from the specification; as a result we obtain a valuation of the parameter `A` (`A` is bound to `number`),

4. now, since the value of the parameter `A` is determined, the success of `permutation/2` can be computed, using the success type from the specification - it is `permutation(list(number),list(number))`,

5. at this point an initial call type of `sorted` can be computed and the analysis proceeds normally, until reaching the fixed point.

The role of the specification in diagnosis is entirely different. The diagnozer constructs verification conditions from the program clauses, and checks if they are correct w.r.t. the specification. The verification algorithm is described in details in [DMP02].

To summarize, the role of specification in the analysis is to complete missing information, like types for builtin predicates, whereas in diagnosis it is verified whether the program conforms the specification.

5 Handling full Prolog

A practical CLP tool should handle a language like SICStus Prolog [SIC98] that conforms to the standard [DEDC96] in high degree.

5.1 Prolog

Even though most of the builtin predicates fit well to our framework, in some of them precision has to be sacrificed, as little information can be determined at compile time. In other cases extra information has to be given in the specification. Below we list some of the problematic cases.

- **Control** structures like "if-then-else" and disjunction are treated correctly in the system: always all possible branches are analyzed, as this is impossible to decide statically which of them will be eventually executed.

  Cuts (`!`) are simply ignored[45] as normally they do not affect type information.

- **Meta predicates** are handled accordingly, provided that the analyzer/diagnozer is given information telling what argument of the meta predicate is an actual goal. This information is given to the system by the predicate `meta_call/2` that has to be defined in the specification for builtins, for example:

  ```
  meta_call(call(A),A).
  meta_call(findall(_,A,_),A).
  ```

  Then TELL knows that, for instance in the call `findall(P,perm(L,P),Ps)` the second argument is a call to the predicate `perm/2`, and concludes that `perm(L,P)` should be further analyzed/diagnosed. Of course, the system can make use of these declaration provided that the goal pattern is known (it is not a variable).

- **Clause retrieval, creation and destruction.** Predicates from this group obviously modify the state of the program and not much can be deduced about them at compile time. If a predicate is dynamic, i.e. its definition is constructed during the execution, then, during analysis, the user will be asked for the types of the predicate. The user's answer is treated as a result of analyzing the predicate definition as it were not missing. The answer is considered as a part of the program specification, and is stored together with it.

[45] Cut is considered as a builtin predicate that cannot cause a run-time error and always succeeds.

- **Term manipulation** also introduces some changes into the program. Terms which show up during execution and are not present in the program text cannot be given types. Therefore they are (over-)approximated by `any`. For example, the only reasonable specification of `functor/3` is:

  ```
  functor(any,any,any) => functor(any,atom,nat)
  ```

  where `nat` stands for natural numbers.

- **Input predicates** clearly cannot give any useful information at compile time. Thus the argument of predicates like `read/1` is of type `any`.

Some problems with writing accurate specifications in our type language come from the fact that many predicates are multi-directional (like `append/3` that can be used for list concatenation as well as for list decomposition). At the moment we are not able to express such a case, which is a limitation of the approach. What can be done now is either to overapproximate all possible uses of the predicate or to choose only one of them.

The current version of TELL does not handle DCG grammars and ignores effect of executing `expand_term/2` predicate when reading the program. This however can be easily incorporated to the system. Problems also occur if operators are defined in the program that extend the standard syntax.

5.2 Handling different constraint domains

Constraint domains are incorporated to SICStus Prolog through libraries. In order to extend the analyzer/diagnozer to handle a constraint domain the following steps have to be taken:

- the user-expert has to identify base types needed, and he has to declare them accordingly, as described in Section 3.1,

- then the specification of the constraint predicates must be written, as for any library, together with grammars defining necessary types.

A sample specification of `clpb` is given in the Appendix 8. It however does not require new base types to be introduced.

6 The architecture

As already pointed, TELL has three main modules: the analyzer, the diagnozer and the specification editor. They use, and in some cases share, software components. The key components are listed below with a note which of the major modules of TELL (**A** - the analyzer, **D** - the diagnozer, **E** - the specification editor) are using them:

operations on types (**A, D**) - they include computing union and intersection, inclusion checking and matching, as defined in Section 4.4 of [DMP02]. Some of them have been implemented by re-using and modifying source code of the analyzer described in [GdW94],

single analysis step (uses the operations on types) (**A, D**) - as explained in [DMP02] this procedure is used both by the analyzer and the diagnozer[46] and has been adopted from [GdW94].

type simplification and presentation (**E**) - the type simplification module decides what name should be used to display a type: user and standard names are preferred over those generated by the type operations (mainly during analysis).

type management (**A, D, E**) - this part is responsible for storing type definitions. Types generated during analysis are stored in new efficient data structures: instead of ordinary lists, as it was implemented by [GdW94], we are using AVL trees; this resulted in about 20% speed-up of analysis, in average. Types from specifications are stored in Prolog database.

widening (**A**) - this is an implementation of one of the widening operations presented in [Mil99] (as the original solution due to [GdW94] caused non-termination in some cases).

The software components and their relationships are depicted in the Figure 10

The TELL diagnosis tool has been implemented in SICStus Prolog [SIC98]. The GUI has been written in Tcl/Tk and uses an interface between Prolog and Tcl/Tk provided by SICStus.

7 Using the TELL interface

The user can communicate with the system through the graphical interface. The interface is shown in Figure 11.

First step that has to be taken by the user is selecting a program to be diagnosed, and loading it together with a file with the program specification, if one exists. By default the file with specification of the program *program*`.pl` has a name *program*`.spec.pl`. Program predicates are listed in window

[46]More systematic treatment of the topic of relating analyzers and verifiers can be found in [CGL01].

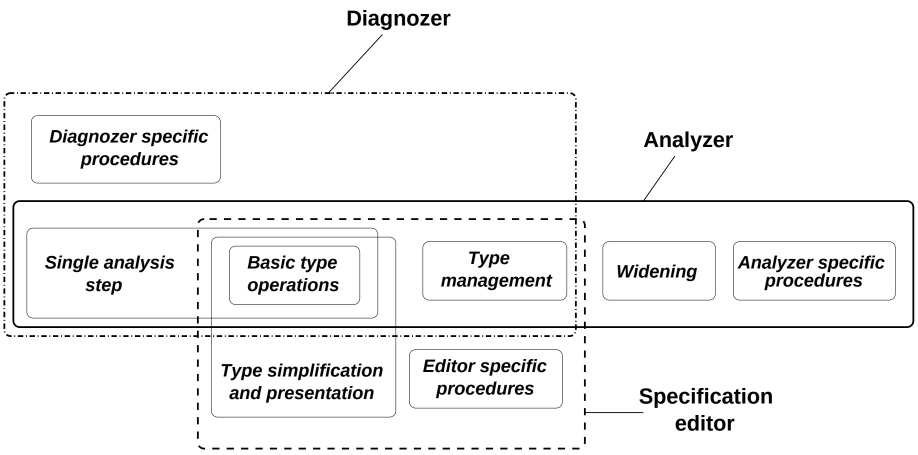

Figure 10: Software components of TELL

Predicates and those call and success types that have been found in the specification are present in User.

Then, the user may (but need not) choose to run the analyzer by pushing the button Analyze. During the analysis the file with the program specification is being looked up for the entry declaration. If the system cannot find one, then the user is asked to input it. If a definition of a predicate is missing (the predicate may be dynamic or not) then the user has to provide its success type (the call type will be computed). The provided type is considered as a part of specification and re-used in the diagnosis phase. The analyzer uses also specifications of builtins and libraries to determine success types of these predicates.

Once the analysis is completed, we can have a look at the inferred types, by selecting a predicate in Predicates. The types are displayed in the left-bottom window named Types. If the type is a synthesized type then its definition is also displayed.

During the analysis the system checks whether calls for the builtin and library predicates, which are in the program, correspond to those expected, i.e. given as call types in the specification. If the computed call type is not a subtype of the call type in the specification then the system displays a warning message saying about possible run-time error. The message includes the builtin predicate, the clause (with original variable names and line numbers in the source code). In addition, there is information about the types of variables that are in the arguments of the call to the builtin.

The user initiates the diagnosis by pointing a predicate to be diagnosed and pushing Diagnose. If the analysis took place, then the inferred types might help in choosing the predicate.

Then, the window Ask shows the list of questions to be asked to the user.

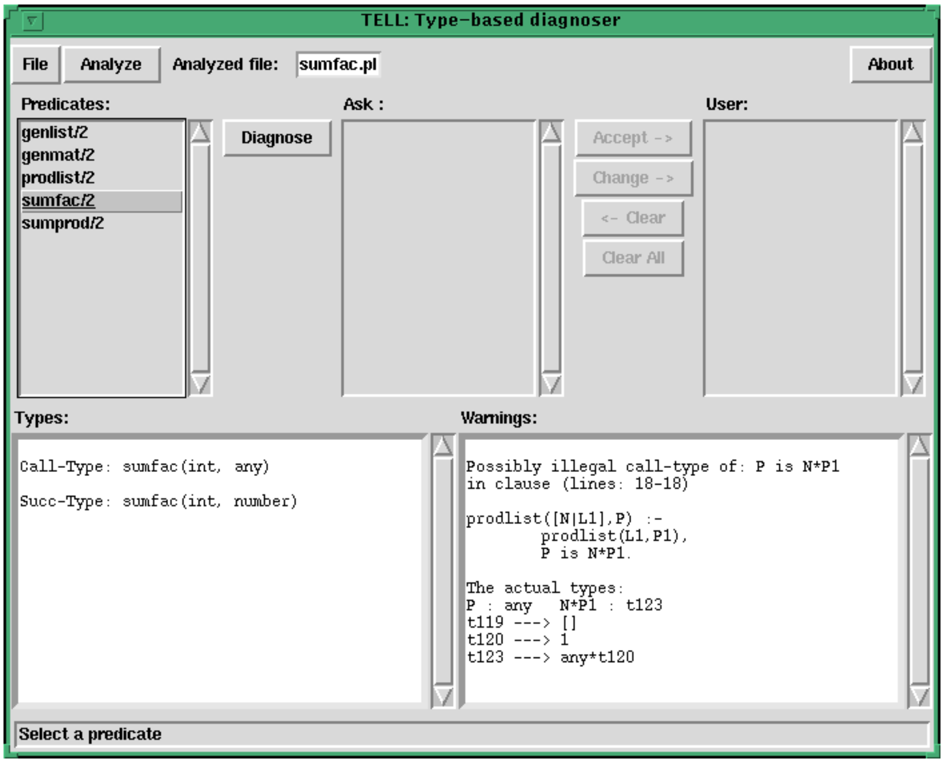

Figure 11: The GUI

Each question is displayed as a procedure name together with a tag (C) when the question is about the call type of the given procedure, or a tag (S) if it is about the success type. The displayed questions concern those types that are not yet specified and are needed to verify the program fragment relevant for the selected predicate. The order in which they appear reflects the following strategy: the more verification conditions contain the given call or success type, the higher is the type's position in the list. Moreover, shorter conditions are preferred over the longer ones (they are counted with a greater weight), as hopefully they will be verified after answering fewer questions.

Then the user is giving a part of the specification that was not defined prior to the diagnosis session (or gives the entire specification if nothing was defined), by answering the questions from the list. If the analysis was performed, the inferred type is a suggestion for the specification and it is displayed in the window Types. The user may accept it by clicking $\boxed{\text{Accept} \rightarrow}$, and the call or success type becomes a part of the specification, or can modify it by choosing $\boxed{\text{Change}}$, and by giving the types for the predicate arguments, as shown in the Figure 12.

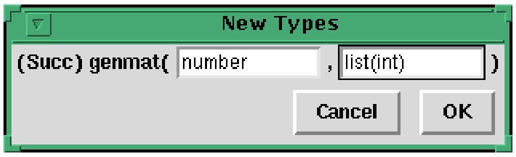

Figure 12: Inputting success types for `genmat/2`

The window User displays information for which predicates the call (success) types have already been specified, either during the session or prior to it. (The syntax is the same as in the Ask window.) If an item in the window User is selected then the definition of the corresponding type is shown in the window Types. It is possible to retract one (← Clear) or all (Clear All) of these types from the specification; then the corresponding questions are re-inserted into the list Ask.

After having introduced some fragments of the type specification we may receive a warning which is a clause prefix[47] incorrect w.r.t. the specification. A sample warning is shown in the Figure 13. Likewise in warnings generated by the analyzer, here we have the original clause with all the information that helps in finding the place in the source code. We also see what call or success type the diagnozer could not verify. There is also information about the actual types of the variables, so the reason of the proof failure can be identified easier.

It may also happen that no warning is generated. Then we are sure that the program is correct w.r.t. the specification.

After the session is completed, the system stores the types given by the user as a specification which can be re-used in further diagnosis sessions.

8 Conclusions

In this paper we have presented main features of the diagnosis tool TELL:

- TELL works statically, i.e. no execution and no test data are needed,
- TELL finds all type errors in the program,
- specification effort is minimized by using results of static analysis and by interactive, demand-driven strategy of entering types (often a partial specification is sufficient to locate an error),
- TELL makes it possible to verify not completely implemented programs (programs with missing fragments).

[47] A prefix of a clause `H:-B1,...,Bn` is any of `H:-B1,...,Bi`, for $\mathtt{i} \leq \mathtt{n}$.

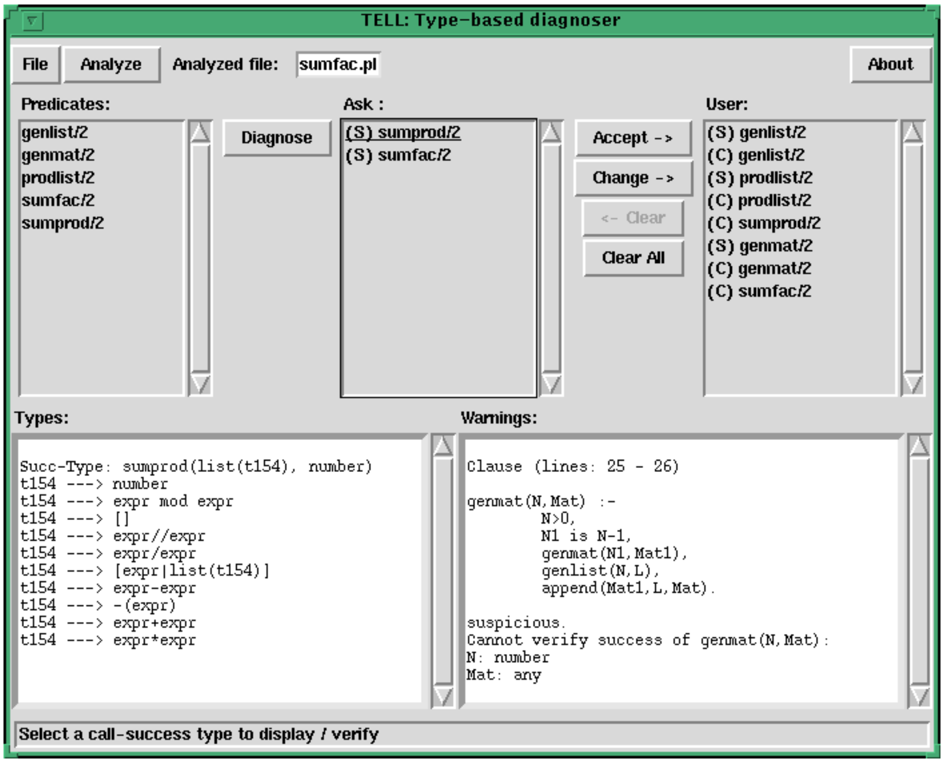

Figure 13: Final step of the session: finding the bug

The architecture of TELL is flexible and allows easy modifications and extensions suitable for other dialects of Prolog/CLP as well as for new libraries. TELL can also be easily customized.

New features of the diagnosis tool, compared to the version of [DMP00a] are:

- handling parametric polymorphism,
- the layered memory of types,
- the user-friendly environment for defining types,
- handling program modules and libraries,
- diagnosing a program without prior analysis.

Future work will focus on completing the implementation and on further development of TELL.

Appendix: A specification of `clpb`

```
typedef truth ---> 0; 1.
typedef cexpr ---> @integer; integer - integer.
typedef bexpr --->
     0;
     1;
     @atom;
     ~ bexpr;
     bexpr * bexpr;
     bexpr + bexpr;
     bexpr # bexpr;
     any ^ bexpr;
     bexpr =:= bexpr;
     bexpr =\= bexpr;
     bexpr =< bexpr;
     bexpr >= bexpr;
     bexpr < bexpr;
     bexpr > bexpr;
     card(list(cexpr),list(bexpr)).

sat(bexpr) => sat(bexpr).
taut(bexpr,any) => taut(bexpr,truth).
labeling(list(any)) => labeling(list(truth)).
```

Bibliography

[AB96] K. Apt and R. Ben-Eliyahu. Meta-variables in logic programming, or in praise of ambivalent syntax. *Fundamenta Informaticae*, 28(1-2):22–36, 1996.

[AL94] A. Aiken and T. K. Lakshman. Directional type checking of logic programs. In B. LeCharlier, editor, *Proc. of the 1st Int. Symposium on Static Analysis*, volume 864 of *LNCS*, pages 43–60. Springer-Verlag, 1994.

[AM94] K. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 3, 1994.

[Apt90] K. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier and The MIT Press, 1990.

[Apt93] K. Apt. Declarative programming in Prolog. In D. Miller, editor, *Proccedings of the International Symposium on Logic Programming*, pages 12–35. The MIT Press, 1993.

[Apt97] K. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[BC89] A. Bossi and N. Cocco. Verifying correctness of logic programs. In J. Díaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development TAPSOFT '89*, volume 352 of *LNCS*, pages 96–110. Springer-Verlag, 1989.

[BCC+97] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. Technical Report CLIP3/97.1, Technical University of Madrid (UPM), 1997.

[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 Kamkar [Kam97], pages 155–169.

[BDM97] J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion based approach. In Kamkar [Kam97], pages 123–140.

[Bei95] C. Beierle. Type inferencing for polymorphic order-sorted logic programs. In L. Sterling, editor, *Logic Programming – Proceedings of the 12th International Conference*, pages 765–779. The MIT Press, 1995.

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

[BLGP$^+$99] F. Bueno, P. López-García, G. Puebla, M. Hermenegildo, and P. Pietrzak. The CHIP Assertion Preprocessor. Technical Report CLIP1/99.1, Technical University of Madrid (UPM), 1999.

[BLR92] F. Bronsard, T. K. Lakshman, and U. Reddy. A framework of directionality for proving termination of logic programs. In K. Apt, editor, *Proceedings of the Joint International Symposium and Conference on Logic Programming*, pages 321–335. The MIT Press, 1992.

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

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM SIGMOD-SIGACT Symposium on Principles of Datebase Systems*, pages 1–15, 1986.

[BO97] F. Benhamou and W. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32(1):1–24, July 1997.

[Boy96] J. Boye. *Directional Types in Logic Programming.* Linköping Studies in Science and Technology, dissertation no. 437, Linköping University, 1996.

[Bra00] S. Brass. SLDMagic - the real magic (with application to Web queries). In J. Lloyd, editor, *Computational Logic - CL2000, The 1st International Conference, Proceedings*, volume 1861 of *LNAI*, pages 1063–1077. Springer-Verlag, 2000.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10(2):91–124, 1991.

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

[Car00] J. Carlson. *Efficient Bottom-up Evaluation of Magic Programs*. Master's thesis, Linköping University, 2000.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[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.

[CC95] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, La Jolla, California, 1995. ACM Press, New York, NY.

[CD93] M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In D. Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 114–129. The MIT Press, 1993.

[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.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997.

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

[CDP99] M. Comini, W. Drabent, and P. Pietrzak. Diagnosis of CHIP programs using type information. In M. C. Meo and M. Vilares Ferro, editors, *APPIA-GULP-PRODE'99, Joint Conference on Declarative Programming*, pages 337–349, 1999.

[CDY94] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Theoretical Computer Science*, 124(1):93–125, 1994.

[CGL01] M. Comini, R. Gori, and G. Levi. How to transform an analyzer into a verifier. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2250 of *LNAI*, pages 595–609. Springer-Verlag, 2001.

[CGLV00] M. Comini, R. Gori, G. Levi, and P. Volpe. Abstract interpretation based verification of logic programs. In S. Etalle and J.-G. Smaus, editors, *Proceedings of the Workshop on Verification of Logic Programs*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2000. Available at URL: `http://www.elsevier.nl/locate/entcs/volume30.html`.

[Cha98] W. Charatonik. Set constraints in some equational theories. *Information and Computation*, 142(1):40–75, 1998.

[Cha00] W. Charatonik. Directional type checking: Beyond discriminative types. In G. Smolka, editor, *Proceedings of the European Symposium on Programming, ESOP'00*, volume 1782 of *LNCS*, pages 72–87. Springer-Verlag, 2000.

[CL00] M. Codish and V. Lagoon. Type dependencies for logic programs using aci-unification. *Theoretical Computer Science*, 238, 2000.

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

[CLMV96] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science, pages 22–50. Springer-Verlag, 1996.

[CLMV99] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.

[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. The MIT Press, 1991.

[Cod99] M. Codish. Efficient goal directed bottom-up evaluation of logic programs. *Journal of Logic Programming*, 38(3):355–370, 1999.

[Com98] M. Comini. *An Abstract Interpretation Framework for Semantics and Diagnosis of Logic Programs.* PhD thesis, Dipartimento di Informatica, Universitá di Pisa, 1998.

[Cos98] Cosytec SA. *CHIP System Documentation*, 1998.

[CP97] W. Charatonik and A. Podelski. Set constraints with intersection. In Glynn Winskel, editor, *Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 362–372. IEEE, June 1997.

[CP98] W. Charatonik and A. Podelski. Directional type inference for logic programs. In G. Levi, editor, *Proceedings of the 5th Static Analysis Symposium (SAS'98)*, volume 1503 of *LNCS*. Springer-Verlag, 1998.

[CP02] J. Carlson and P. Pietrzak. Improving goal directed bottom-up evaluation of logic programs, 2002. Submitted.

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

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard.* Springer-Verlag, 1996.

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

[DHM00] P. Deransart, M. Hermenegildo, and J. Małuszyński, editors. *Analysis and Visualization Tools for Constraint Programming*, volume 1870 of *LNCS*. Springer-Verlag, 2000.

[DJMT97] P. Devienne, Talbot J.-M., and S. Tison. Set-based analysis for logic programming and tree automata. In P. Van Hentenryck, editor, *Proceedings of the Static Analysis Symposium, SAS'97*, volume 1302 of *LNCS*, pages 127–140. Springer-Verlag, 1997.

[DM88] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59(1):133–155, 1988.

[DM93] P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming*. The MIT Press, 1993.

[DM01] W. Drabent and M. Miłkowska. Proving correctness and completeness of normal programs - a declarative approach. In P. Codognet, editor, *Logic Programming, 10th International Conference (ICLP'2001)*, volume 2237 of *LNCS*, pages 284–299. Springer-Verlag, 2001.

[DMP00a] W. Drabent, J. Małuszyński, and P. Pietrzak. Locating type errors in untyped CLP programs. In Deransart et al. [DHM00], chapter 4, pages 121–150.

[DMP00b] W. Drabent, J. Małuszyński, and P. Pietrzak. Type-based diagnosis of CLP programs. *Electronic Notes in Theoretical Computer Science*, 30(4), 2000.

[DMP01] W. Drabent, J. Małuszyński, and P. Pietrzak. Parametric descriptive types for (C)LP. In P. Codognet, editor, *The Tenth International French Speaking Conference on Logic and Constraint Programming*, pages 239–254, Paris, April 2001. Hermes-Science Publications.

[DMP02] W. Drabent, J. Małuszyński, and P. Pietrzak. Using parametric set constraints for locating errors in CLP programs. *Theory and Practice of Logic Programming*, 2(4–5):549–611, 2002.

[DNTM88] W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The use of assertions in algorithmic debugging. In *Proc. of Fifth Generation Computer Systems 88*, pages 573–581, 1988.

[DNTM89] W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. The MIT Press, 1989.

[DP98] W. Drabent and P. Pietrzak. Inferring call and success types for CLP programs. ESPRIT DiSCiPl deliverable, September 1998. Available at URL: `http://discipl.inria.fr/deliverables2.html`.

[DP99] W. Drabent and P. Pietrzak. Type analysis for CHIP. In A. M. Haeberer, editor, *Proceedings of the Seventh International Conference on Algebraic Methotology and Software Technology (AMAST'98)*, volume 1548 of *LNCS*, pages 389–406. Springer-Verlag, 1999.

[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.

[DS01] P. Deransart and J.-G. Smaus. Well-typed logic programs are not wrong. In H. Kuchen and K. Ueda, editors, *Functional and Logic Programming (FLOPS'2001)*, volume 2024 of *LNCS*, pages 280–295. Springer-Verlag, 2001.

[DTT97] P. Devienne, J.-M. Talbot, and S. Tison. Solving classes of set constraints with tree automata. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming - CP97*, volume 1330 of *LNCS*, pages 62–76, Berlin, Germany, October 1997. Springer-Verlag.

[DZ92] P. Dart and J. Zobel. A regular type language for logic programs. In Pfenning [Pfe92], pages 157–187.

[FC01] F. Fages and E. Coquery. Typing constraint logic programs. *Theory and Practice of Logic Programming*, 1(6):751–777, 2001.

[FD93] G. Ferrand and P. Deransart. Proof method of partial correctness and weak completeness for normal logic programs. *Journal of Logic Programming*, 17(2,3 & 4):265–278, 1993.

[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.

[Flo67] R. W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Proceedings if the Symposium in Applied Mathematics of the AMS*, pages 19–32, 1967.

[Frü89] T. Frühwirth. Type inference by program transformation and partial evaluation. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 263–282. The MIT Press, 1989.

[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*, pages 300–309, Amsterdam, 1991. 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. The MIT Press, 1994.

[GP02] J. Gallagher and G. Puebla. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In S. Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002*, volume 2257 of *LNCS*, pages 243–261. Springer-Verlag, 2002.

[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.

[Han91] M. Hanus. Horn clause programs with polymorphic types: Semantics and resolution. *Theoretical Computer Science*, 89:63–106, 1991.

[Han92] M. Hanus. Logic programming with type specifications. In Pfenning [Pfe92], pages 91–140.

[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, 1995.

[Hei92a] N. Heintze. Practical aspects of set based analysis. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming* , pages 765–779. The MIT Press, 1992.

[Hei92b] N. Heintze. *Set based program analysis.* PhD thesis, Carnegie Mellon University, 1992.

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

[HJ90a] N. Heintze and J. Jaffar. A decision procedure for a class of set constraints (extended abstract). In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51, 1990.

[HJ90b] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 197–209, 1990.

[HJ91] N. Heintze and J. Jaffar. A Decision Procedure for a Class of Set Constraints. Technical Report CMU-CS-91-110, Carnegie Mellon University, 1991.

[HJ92] N. Heintze and J. Jaffar. Semantic types for logic programs. In Pfenning [Pfe92], pages 141–155.

[HJ94] N. Heintze and J. Jaffar. Set constraints and set-based analysis. In A. Borning, editor, *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, volume 874 of *LNCS*, pages 281–298. Springer-Verlag, 1994.

[HK88] K. Horiuchi and T. Kanamori. Polymorphic type inference in Prolog by abstract interpretation. In *Proceedings of Conference in Logic Programming*, pages 195–214. Springer, 1988.

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

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 2 edition, 2001.

[Hoa69] A. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):276–280, 1969.

[HPB99] M. Hermenegildo, G. Puebla, and F. Bueno. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In K. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, LNAI, pages 161–192. Springer-Verlag, 1999.

[HT92] P.M. Hill and R.W. Topor. A semantics for typed logic programs. In Pfenning [Pfe92], 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.

[Kåg93] A. Kågedal. Improvements to Compile-Time Analysis for Ground Prolog. In M. Bruynooghe and J. Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP'93)*, volume 724 of *LNCS*, pages 92–107. Springer-Verlag, 1993.

[Kåg95] A. Kågedal. *Exploiting Groundness in Logic Programs.* Linköping Studies in Science and Technology, dissertation no. 383, Linköping University, 1995.

[Kam97] M. Kamkar, editor. *Proceedings of AADEBUG'97 (The Third International Workshop on Automated Debugging).* Linköping University, Sweden, 1997.

[KH85] T. Kanamori and K. Horiuchi. Type inference in Prolog and its application. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 704–404. Morgan Kaufmann, 1985.

[Klu88] F. Kluźniak. Type synthesis for Ground Prolog. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 788–816. The MIT Press, 1988.

[KSF90] M. Kamkar, N. Shahmehri, and P. Fritzson. Bug localization by algorithmic debugging and program slicing. In *Programming Languages Implementation and Logic Programming*

(PLILP'90), volume 456 of *LNCS*, pages 60–74. Springer-Verlag, 1990.

[Laï00] C. Laï. Assertions with constraints for CLP debugging. In Deransart et al. [DHM00], chapter 3, pages 109–120.

[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. The MIT Press, 1991.

[LS88] Y. Lichtenstein and E. Y. Shapiro. Abstract algorithmic debugging. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of Fifth International Conference and Symposium on Logic Programming*, pages 512–531. The MIT Press, 1988.

[Lu98] L. Lu. Polymorphic type analysis of logic programs by abstract interpretation. *Journal of Logic Programming*, 36(1):1–54, 1998.

[Lu00] L. Lu. A precise type analysis of logic programs. In *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practices of Declarative Programming*, pages 214–225. The ACM Press, 2000.

[LV98] G. Levi and P. Volpe. Derivation of proof methods by abstract interpretation. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the Joint International Coneference ALP/PLILP'98*, volume 1490 of *LNCS*, pages 102–117. Springer-Verlag, 1998.

[Mal94] B. Malfon. Characterization of some semantics for logic programs with negation and application to program validation. In M. Bruynooghe, editor, *Logic Programming. Proc. of the 1994 Symposium*, pages 91–105. The MIT Press, 1994.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mil99] P. Mildner. *Type Domains for Abstract Interpretation, A Critical Study.* PhD thesis, Uppsala University, 1999.

[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. A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[MS93] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.

[MS97] K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1997.

[MUV86] K. Morris, J. D. Ullman, and A. Van Gelder. Design overview of the NAIL! system. In E. Y. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, volume 225 of *LNCS*, pages 554–568. Springer-Verlag, 1986.

[Nai91] L. Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10:255–285, 1991.

[Nai96] L. Naish. A declarative view of modes. In M. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 185–199. The MIT Press, 1996.

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

[Nil92] U. Nilsson. *Abstract Interpretations & Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs.* Linköping Studies in Science and Technology, dissertation no. 265, Linköping University, 1992.

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

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[PBH00a] G. Puebla, F. Bueno, and M. Hermenegildo. An assertion language for constraint logic programs. In Deransart et al. [DHM00], chapter 1, pages 23–61.

[PBH00b] G. Puebla, F. Bueno, and M. Hermenegildo. A generic preprocessor for program validation and debugging. In Deransart et al. [DHM00], chapter 2, pages 63–107.

[Per86] L. M. Pereira. Rational debugging in logic programming. In E. Y. Shapiro, editor, *Proceedings of the 3rd International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 203–210. Springer-Verlag, 1986.

[Pfe92] F. Pfenning, editor. *Types in Logic Programming.* The MIT Press, 1992.

[Pie98] P. Pietrzak. *Static Incorrectness Diagnosis of CLP(FD).* Linköping Studies in Science and Technology, Lic. thesis no. 742, Linköping University, 1998.

[PP97] L. Pacholski and A. Podelski. Set constraints - a pearl in research on constraints. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming - CP97*, volume 1330 of *LNCS*, pages 549–561. Springer-Verlag, 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. The MIT Press, 1989.

[PR99] D. Pedreschi and S. Ruggieri. Verification of logic programs. *Journal of Logic Programming*, 39(1–3):125–176, 1999.

[Ram91] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3–4):189–216, 1991.

[Red88] U. Reddy. Notions of polymorphism for preducate logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of Fifth International Conference and Symposium on Logic Programming (Addendum)*, pages 17–34, 1988.

[RLK86] J. Rohmer, R. Lescæur, and J.-M. Kerisit. The Alexander method - a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, 4(3):273–285, 1986.

[RT00] P. Rychlikowski and T. Truderung. *Polymorphic Directional Types For Logic Programming.* PhD thesis, Wrocław University, 2000. `http://www.tcs.uni.wroc.pl/~tt/dirtypes/index.html`.

[RT01] P. Rychlikowski and T. Truderung. Polymorphic directional types for logic programming. In *Proceedings of the 3rd Interna-*

tional ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, pages 61–72. ACM Press, 2001.

[Sek89] H. Seki. On the power of Alexander Templates. In *Proceedings of the 8th SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 150–159, 1989.

[SFD00] J.-G. Smaus, F. Fages, and P. Deransart. Using modes to ensure subject reduction for typed logic programs with subtyping. In S. Kapoor and S. Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science (FST TCS'2000)*, volume 1974 of *LNCS*, pages 214–226. Springer-Verlag, 2000.

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

[Sha91] N. Shahmehri. *Generalized Algorithmic Debugging.* Linköping Studies in Science and Technology, dissertation no. 260, Linköping University, 1991.

[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 constrauction for mode analysis of typed logic programs. Technical report, University of Kent, November 1997.

[SIC98] SICS (Swedish Instituite of Computer Science). *SICStus Prolog User's Manual*, 1998.

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

[Tar72] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[TDT00] J.-M. Talbot, P. Devienne, and S. Tison. Generalized definite set constraints. *Constraints: An International Journal*, 5(1-2):161–202, January 2000.

[Tes96] A. Tessier. Declarative debugging in constraint logic programming. In J. Jaffar and R. Yap, editors, *Asian Computing Science Conference*, volume 1179 of *LNCS*. Springer–Verlag, December 1996.

[TF00] A. Tessier and G. Ferrand. Declarative diagnosis in the CLP scheme. In Deransart et al. [DHM00], chapter 5, pages 151–174.

[TL96] J. Tan and I-P. Lin. Type synthesis for logic programs. In M. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 200–214. The MIT Press, 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. The MIT Press, 1987.