Automated Verification of Behavioural Properties of Prolog Programs

B. Le Charlier[1] and C. Leclère[1] and S. Rossi[2] and A. Cortesi[3]

[1] Institut d'Informatique, 21 rue Grandgagnage, B-5000 Namur, Belgium
e-mail: {ble,clc}@info.fundp.ac.be
[2] Dip. di Matematica, via Belzoni 7, 35131 Padova, Italy
e-mail: sabina@math.unipd.it
[3] Dip. di Matematica e Informatica, via Torino 155, 30173 Venezia, Italy
e-mail: cortesi@dsi.unive.it

Abstract. Program verification is a crucial issue in the field of program development, compilation and debugging, in particular for declarative programming. In this paper, we present an analyser for Prolog which aims at verifying whether the execution of a program behaves according to a given specification (behavioural assumptions). The analyser is based on the methodology of abstract interpretation. A novel notion of abstract sequence is introduced, that includes an over-approximatimation of successful inputs (this is useful to detect mutual exclusion of clauses), and expresses size relation information between successful inputs and the corresponding outputs, together with cardinality information in terms of input argument sizes.

1 Introduction

Declarative languages have received great attention in the last years, as they allow the programmer to focus on the description of the problem to be solved while ignoring low level implementation details. Nevertheless, the implementation of declarative languages remains a delicate issue. Very often, languages include additional "impure" features which are intended to improve the efficiency of the programs but do not respect their declarative nature. This is what happens in logic programming with Prolog, where a number of these "impure" features arise, e.g. the incomplete (depth-first) search rule, the non logical negation by failure, the non-logical test predicates like `var`, and the cut.
Many static analysis techniques have been proposed in the literature to improve on this situation. Some analyses aim at optimizing programs automatically, relieving the programmer from using impure control features [8, 16]. Other analyses are designed to verify that a non declarative implementation of a program does behave according to its declarative meaning [6, 9]. These analyses may also be useful to transform a first (declaratively but not operationally correct) version of a program into a both declaratively and operationally correct program.

In this paper, we describe an analyser for Prolog programs which aims at verifying whether the concrete execution of a program behaves according to a

given specification consisting of a set of behavioural assumptions. The analyser is based on the methodology of abstract interpretation [5, 14]: it can be seen as an online approximation of (sets of) concrete program executions. However, instead of performing a fixpoint computation, the analyser makes use of declarations on allowed program executions (*behaviours*) provided by the user. Therefore, the emphasis here is not on the analysis of substitution properties, like modes, sharing and types, that can be automatically inherited from previous works [14], but on the automatic verification of assumptions like on the number of solutions, or on the size relations between input and output substitutions.
Our analyser is built upon the notion of *abstract sequence* [2, 12, 3, 13]. Abstract sequences allow to describe pairs $\langle \theta, S \rangle$, where θ is a substitution and S is the sequence of answer substitutions resulting from executing a program (a procedure, a clause, etc.) with input substitution θ. In this paper, we revisit this notion. The main novelties are:

- the new notion includes an over-approximatimation of successful inputs; this is useful to detect mutual exclusion of clauses;
- it allows to express size relation information between successful inputs and the corresponding outputs;
- it allows to express cardinality information in terms of input argument sizes.

Basically, the new notion of abstract sequence is more "relational", since it may relate the number of solutions and the size of output terms to the size of input terms in full generality. For instance, it may relate the input and output sizes of the same [4] term without requiring any invariance under instantiation.

This paper presents the design of the operations on abstract sequences that are required for computing non trivial information on size relations and cardinality. The interested reader may found more details and correctness proofs in [10]. The practical implementation of the analyser, based on the generic system *GAIA* [14] and on the polyhedron library [17] to manipulate size information is still under progress.

The analyser presented in this paper may apply to any kind of Prolog program (without dynamic predicates such as `assert` and `retract`). To cite some of its applications, it could be integrated in a programming environment to check correctness of Prolog programs and/or to derive efficient Prolog programs from purely logic descriptions. In particular, since our notion of *behaviour* for a procedure subsumes the formal part of the specification schema proposed in [7], it may be integrated in the FOLON environment [6, 9] which was developed for supporting the automatable aspects of Deville's methodology for logic program construction. Our system could support the automatable aspects of other works on verification, e.g., [1]. Moreover, since the information provided by the user is certified by the system, it can be used by a compiler to optimize the object code. Finally, since it may verify precise relations between the size of the arguments and the number of solutions to a procedure, it can be used as a basis for an automatic complexity analysis.

[4] i.e., bound to the same program variable.

The paper is organized as follows. Section 2 provides an overview of the analyser. Section 3 recalls some basic concepts. Section 4 illustrates our domain of abstract sequences. Section 5 describes the analyser. Section 6 discusses the implementation of the unification operation.

2 Overview of the Analyser

In this section, we introduce the main functionalities of the analyser by discussing a simple example. Consider the Prolog procedure `select/3` depicted below.

```
select(X, L, LS):- L=[H|T], H=X, LS=T, list(T).
select(X, L, LS):- L=[H|T], LS=[H|TS], select(X, T, TS).
```

Declaratively, it defines a relation between three terms[5] X, L, and LS that holds if and only if the terms L and LS are lists and LS is obtained by removing one occurrence of X from L. Our analyser is not aimed at verifying this (informal) declarative specification. Instead, it checks a number of operational properties which ensure that the program execution of this program actually computes the specified relation, provided that the procedure is "declaratively" correct (and its operational specification guarantees such correctness). We assume one particular and reasonable class of input calls, i.e., calls such that X and LS are *distinct* variables and L is any ground term (not necessarily a list). For this class of input calls, the user has to provide a description of the expected behaviour of the procedure by means of an abstract sequence B and a size expression se. The abstract sequence B is a tuple $\langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref_out}, E_{sol} \rangle$, where

1. β_{in} is an abstract substitution describing the class of input calls;
2. β_{ref} is an abstract substitution describing an over approximation of the successful input calls, i.e., those that produce at least one solution: in our example, β_{ref} states that L is a non empty ground list;
3. β_{out} is an abstract substitution describing an over approximation of the set of outputs corresponding to the successful calls. In our example, β_{out} states that X is a ground term and LS is a ground list;
4. the component E_{ref_out} describes a relation between the size of the terms occurring in a successful call and the size of the terms returned by the call. In our example, E_{ref_out} states that the input length of L is equal to the output length of LS plus 1;
5. the component E_{sol} describes a relation between the size of the terms occurring in a successful call and the number of solutions returned by the call. In our example, E_{sol} states that the number of solutions is equal to the input length of L.

The size expression se is a positive integer expression over the formal parameters of the procedure denoting the size of the corresponding input terms; this

[5] We use roman letters to denote the values to which program variables are instantiated. Syntactic objects are denoted by typewriter characters.

expression must decrease strictly through recursive calls. In our example, *se* is equal to `L` representing the input length of L, denoted by $\|L\|$. This means that in every recursive call, the length of the list given as second argument decreases strictly.

Starting from B and *se*, the analyser computes a number of abstract sequences, one for every prefix of the body of every clause of the procedure, one for every clause, and, finally, one for the complete procedure. In our example, the analyser computes an abstract sequence B_1 for the first clause, expressing that for the specified class of input calls, the first clause succeeds if and only if L is a non empty list, and it succeeds exactly once. The derivation of this information is possible because the analyser is able to detect that the unification `L=[H|T]` succeeds if and only if L is of the form $[t_1|t_2]$ (not necessarily a list) and that both unifications `H=X` and `LS=T` surely succeed since X and LS are free and do not share.

B_1 is obtained by combining this information with the abstract sequence that describes the behaiviour of the procedure `list/1`. The latter states that, for ground calls, the literal succeeds only for lists, and it succeeds exactly once.

The second clause of `select/3` is treated similarly. Only the recursive call deserves a special treatment. First, the analyser infers that the recursive call will be executed at most once and, in fact, exactly once when L is of the form $[t_1|t_2]$. It also infers that X and TS are distinct variables and that T is ground and strictly smaller than L with respect to the norm $\|\cdot\|$. Thus, it can be assumed by induction that the recursive call satisfies the conditions provided by the user through the abstract sequence B. So, the analyser deduces that the recursive call succeeds only if T is a *non-empty* list and that it returns a number of solutions equal to the length of T. It also infers that X is ground and that TS is a ground list whose size is the same as the size of T minus 1. Putting all pieces together, the analyser computes the abstract sequence B_2 for the second clause, which states that the second clause succeeds only for a list L of at least two elements (and actually succeeds for all of them) and that the output size of LS is equal to the size of L minus 1, i.e., $\|L\| - 1$; moreover, the number of solution is also equal to $\|L\| - 1$.

The last step for the analyser is to combine the abstract sequences B_1 and B_2 to get a new abstract sequence B_{out} describing the behaviour of the whole procedure. Once again, a careful analysis is necessary to get the most precise result. When L is a list of at least two elements, the first clause succeeds once and the second one succeeds $\|L\| - 1$ times. Thus, the procedure succeeds $\|L\|$ times. Otherwise, when the length of L is equal to 1, the second clause fails and the first one succeeds exactly once. Thus, in both cases the procedure succeeds $\|L\|$ times. Hence, putting the abstract sequences B_1 and B_2 together, the analyser is able to reconstruct exactly the information provided by the user, which is automatically verified to be correct.

3 Preliminaries

The reader is assumed to be familiar with the basic concepts of abstract interpretation for logic programming [5]. We denote by $\mathcal{T}$ the set of all terms, and for any set of indices I, we denote by $\mathcal{T}^I$ the set of all tuples of terms $\langle t_i \rangle_{i \in I}$. A size measure (or norm) is a function $\| \cdot \| : \mathcal{T} \to \mathbf{N}$. Here, we refer to the list-length measure defined for any term t by $\|t\| = 1 + \|t_2\|$ if t is of the form $[t_1|t_2]$ and $\|t\| = 0$ otherwise. The *disjoint union* of two (possibly non disjoint) sets A and B is an arbitrarily set $A + B$ equipped with two injections functions in_A and in_B such that: for any set C and for any pair of functions $f_A : A \to C$ and $f_B : B \to C$, there exists a unique function $f : A + B \to C$ with $f_A = f \circ in_A$ and $f_B = f \circ in_B$ (the symbol $\circ$ is the usual function composition). We denote the function f by $f_A + f_B$. For any set V, we denote by $\mathbf{Exp}_V$ the set of all integer linear expressions with variables in V. An element $se \in \mathbf{Exp}_{\{X_1,\ldots,X_m\}}$ can also be seen as a function from $\mathbf{N}^m$ to $\mathbf{N}$. The value of $se(\langle n_1, \ldots, n_m \rangle)$ is obtained by evaluating the expression se where each X_i is replaced by n_i.

A *(normalized) program* P is a non empty set of procedures pr. A procedure is a non empty sequence of clauses c. Each clause has the form h: $-g$ where the head h is of the form $p(X_1, \ldots, X_n)$ and p is a predicate symbol of arity n, whereas the body g is a possibly empty sequence of literals. A literal l is either a built-in of the form $X_{i_1} = X_{i_2}$, or a built-in of the form $X_{i_1} = f(X_{i_2}, \ldots, X_{i_n})$ where f is a functor of arity $n-1$, or an atom $p(X_{i_1}, \ldots, X_{i_n})$. The variables occurring in a literal are all distinct; all clauses of a procedure have exactly the same head. We denote by $\mathcal{P}$ the set of all predicate symbols occurring in the program P. Variables used in the clauses are called *program variables* and are denoted by $X_1, \ldots, X_i, \ldots$.

A *(program) substitution* θ is a finite set $\{X_{i_1}/t_1, \ldots, X_{i_n}/t_n\}$ where variables $X_{i_1}, \ldots, X_{i_n}$ are distinct program variables and the t_i's are terms. The domain of θ, denoted by $dom(\theta)$, is the set of variables $\{X_{i_1}, \ldots, X_{i_n}\}$. Variables occurring in $t_1, \ldots, t_n$ are taken from the set of *standard variables* which is disjoint from the set of program variables. A *standard substitution* σ is a substitution in the usual sense which only uses standard variables. The application of a standard substitution σ to a program substitution $\theta = \{X_{i_1}/t_1, \ldots, X_{i_n}/t_n\}$ is the program substitution $\theta\sigma = \{X_{i_1}/t_1\sigma, \ldots, X_{i_n}/t_n\sigma\}$. We say that θ_1 is *more general* than θ_2, noted $\theta_2 \leq \theta_1$, iff there exists σ such that $\theta_2 = \theta_1\sigma$. $mgu(t_1, t_2)$ denotes the set of standard substitutions that are a most general unifier of t_1 and t_2. The *restriction* of θ to a set of variables D, denoted by $\theta_{/D}$, is such that $dom(\theta_{/D}) = D$ and $X_i\theta = X_i(\theta_{/D})$, for all $X_i \in D$. A *(program) substitution sequence* S is a *finite* sequence $< \theta_1, \ldots, \theta_n >$ of (program) substitutions with the same domain. We denote by $< >$ the empty sequence and by The symbol :: denotes the sequence concatenation.

Our analyser refers to the concrete semantics of Prolog programs presented in [13], which has been proven equivalent to Prolog operational semantics in [11]. In this approach, the concrete semantics for a Prolog program P is a total function from the set of pairs $\langle \theta, p \rangle$, where $p \in \mathcal{P}$ has arity n and $dom(\theta) = \{X_1, \ldots, X_n\}$, to the set of substitution sequences. The fact that $\langle \theta, p \rangle$ is mapped

to the sequence S is denoted by $\langle \theta, p \rangle \longmapsto S$. Here, this means that the execution of $p(X_1, \ldots, X_n)\theta$ terminates and produces the (finite, possibly empty) sequence of answer substitutions S.

4 Abstract Domains

In this section, we describe the abstract objects used by the analyser, namely, *abstract substitutions*, *abstract sequences* and *behaviours*.

Abstract Substitutions. Our domain of *abstract substitutions* is an instantiation of the generic domain $\mathtt{Pat}(\Re)$ [4]. Here, we give only an informal presentation of it and we refer the reader to our previous papers [14, 4] for more details.
An abstract substitution β with domain $\{X_1, \ldots, X_n\}$ describes a set of program substitutions with the same domain giving information not only about the terms to which $X_1, \ldots, X_n$ are bound, but also about some subterms of them. The terms described in β are denoted by indices from a set I. We say that β is an abstract substitution over I. The following properties of terms are captured: the *pattern*, which specifies the main functor of a term as well as the subterms that are its arguments; the *mode* (e.g., `ground`, `var`); the *type* (e.g., `list`); and the *possible sharing* with other subterms. We denote by $Cc(\beta)$ the set of all substitutions described by β, by $\bot$ the abstract substitution describing the empty set, i.e., $Cc(\bot) = \emptyset$, and by $\mathtt{DECOMP}(\theta, \beta)$ the set of all tuples of terms $\langle t_i \rangle_{i \in I}$ respecting the term properties described in β and such that $\theta = \{X_1/t_{i_1}, \ldots, X_n/t_{i_n}\}$ (if $i_1, \ldots, i_n$ denote the terms bound to $X_1, \ldots, X_n$, respectively).

As an example, consider the abstract substitutions β_{in}, β_{ref} and β_{out} informally described in Section 2. They can be represented as follows. The terms bound to the formal parameters `X`, `L` and `LS` are denoted by indices `1`, `2` and `3`, respectively, whereas the subterms of `L` in β_{ref} and β_{out} are denoted by indices `4` and `5`, in

β_{in} : $\mathtt{X_1/var(1)}$, $\mathtt{X_2/ground(2)}$, $\mathtt{X_3/var(3)}$, $\mathtt{noshare(1,3)}$
β_{ref} : $\mathtt{X_1/var(1)}$, $\mathtt{X_2/[ground(4)|ground_list(5)](2)}$, $\mathtt{X_3/var(3)}$, $\mathtt{noshare(1,3)}$
β_{out} : $\mathtt{X_1/ground(1)}$, $\mathtt{X_2/[ground(4)|ground_list(5)](2)}$, $\mathtt{X_3/ground_list(3)}$

Abstract Sequences. We formally describe here our new domain of *abstract sequences*. An abstract sequence B is a tuple $\langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref_out}, E_{sol} \rangle$. The first element β_{in} is the input abstract substitution; β_{ref} is a refinement of β_{in} approximating the set of concrete substitutions in $Cc(\beta_{in})$ that surely succeeds (i.e., whose execution produces at least one result); β_{out} approximates output information about variable instantiation. E_{ref_out} represents size relations between the output and the input arguments (we refer to β_{ref} for the input) whereas E_{sol} expresses the number of solutions in terms of the input argument sizes. The size components E_{ref_out} and E_{sol} are abstract objects representing tuples of natural numbers.

Here, we assume that a size component E over a set of indices I is a system of linear equations and inequations over $\mathbf{Exp}_I$. It represents the set of all tuples

of natural numbers $\langle n_i \rangle_{i \in I} \in \mathbf{N}^I$ which are solutions of E. We denote by $Cc(E)$ this set. In order to distinguish indices of I from integer coefficient and constants, when writing elements of $\mathbf{Exp}_I$, we wrap up each element i of I into the symbol $\mathtt{sz}(i)$. If f is a function from one set of indices to another one, such that $f(i) = i'$ and $f(j) = j'$, the expression $\mathtt{sz}(f(i)) = \mathtt{sz}(f(j)) + 1$ stands for the syntactical equation $\mathtt{sz}(i') = \mathtt{sz}(j') + 1$.

We are now in position to formally define abstract sequences. The symbol *sol* denotes a special index representing the number of substitutions belonging to the approximated sequences.

Definition 1 (Abstract Sequence). An *abstract sequence* B is either $\bot$ or a tuple of the form $\langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref_out}, E_{sol} \rangle$ where

- β_{in} is an abstract substitution over I_{in};
- β_{ref} is an abstract substitution over I_{ref} with $dom(\beta_{ref}) = dom(\beta_{in})$ and $Cc(\beta_{ref}) \subseteq Cc(\beta_{in})$;
- β_{out} is an abstract substitution over I_{out} with $dom(\beta_{out}) \supseteq dom(\beta_{in})$;
- E_{ref_out} is a size component over $I_{ref} + I_{out}$;
- E_{sol} is a size component over $I_{ref} + \{sol\}$;
- for all $\theta' \in Cc(\beta_{out})$, $\exists \theta \in Cc(\beta_{ref})$ such that $\theta'_{/dom(\beta_{ref})} \leq \theta$.

The abstract sequence B represents the set of all pairs $\langle \theta, S \rangle$, noted $Cc(B)$, such that $\theta \in Cc(\beta_{in})$, S is a sequence of substitutions with $Subst(S) \subseteq Cc(\beta_{out})$ and

- if $S \neq <>$ then $\theta \in Cc(\beta_{ref})$;
- for all $\theta' \in Subst(S)$, if $\langle t_i \rangle_{i \in I_{ref}} \in \mathtt{DECOMP}(\theta, \beta_{ref})$ and $\langle s_i \rangle_{i \in I_{out}} \in \mathtt{DECOMP}(\theta', \beta_{out})$ then $\langle \|t_i\| \rangle_{i \in I_{ref}} + \langle \|s_i\| \rangle_{i \in I_{out}} \in Cc(E_{ref_out})$;
- if $\langle t_i \rangle_{i \in I_{ref}} \in \mathtt{DECOMP}(\theta, \beta_{ref})$ then $\langle \|t_i\| \rangle_{i \in I_{ref}} + \{sol \mapsto |S|\} \in Cc(E_{sol})$.

Consider the abstract substitutions β_{in}, β_{ref}, and β_{out} described above where $I_{ref} = I_{out} = \{1, 2, 3, 4, 5\}$. Let $in_{ref} : I_{ref} \to I_{ref} + I_{out}$, $in_{out} : I_{out} \to I_{ref} + I_{out}$ and $in_{sol} : I_{ref} \to I_{ref} + \{sol\}$ be injection functions. The behaviour for the procedure `select/3` described in Section 2 can be expressed in terms of the abstract sequence $B = \langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref_out}, E_{sol} \rangle$ where $E_{ref_out} = \{\mathtt{sz}(in_{ref}(2)) = \mathtt{sz}(in_{out}(3)) + 1\}$ and $E_{sol} = \{sol = \mathtt{sz}(in_{sol}(2))\}$.

Behaviours. A behaviour for a procedure is a formalization of its behavioural properties provided by the user to the system.

Definition 2 (Behaviour). A *behaviour* Beh_p for a procedure p/n is a finite set of pairs $\{\langle B_1, se_1 \rangle, \ldots, \langle B_m, se_m \rangle\}$ where for all $k \in \{1. \ldots, m\}$,

- $B_k = \langle \beta^k_{in}, \beta^k_{ref}, \beta^k_{out}, E^k_{ref_out}, E^k_{sol} \rangle$ is an abstract sequence with $dom(\beta^k_{in}) = dom(\beta^k_{ref}) = dom(\beta^k_{out}) = \{X_1, \ldots, X_n\}$;
- se_k is a positive linear expression from $\mathbf{Exp}_{\{X_1, \ldots, X_n\}}$.

The behaviour for the procedure `select/3` described in Section 2 is simply $\{\langle B, \mathtt{X}_2 \rangle\}$ where B is the abstract sequence defined above.

In the following, we assume that a set of behaviours *SBeh* for a program P contains exactly one behaviour Beh_p for each procedure name $p \in \mathcal{P}$.

Definition 3 (Consistency). We say that a set of behaviours *SBeh* for a program P is *consistent with respect to the concrete semantics of* P iff for all $p \in \mathcal{P}$ and for all $\langle B, se\rangle \in Beh_p$, the execution of the procedure p called with a substitution θ described by the input abstract substitution of B terminates and $\langle \theta, p\rangle \mapsto S$ implies $\langle \theta, S\rangle \in Cc(B)$.

5 The Analyser

Our analyser follows the standard top-down verification technique: for a given program, it analyses each procedure; for each procedure, it analyses each clause; for each clause, it analyses each atom such that if an atom is a procedure call, then it look-up at the behaviour to infer information about its execution.

The algorithm of the analyser is fully presented in the Appendix. We illustrate here its main operations. To simplify the presentation, we assume that programs contain no mutually recursive procedures. We discuss how we remove this simplification at the end of this section.

Analysis of a Program. The analysis of a program is performed by the procedure `analyze_program` which takes as input a program P and a set of behaviours *SBeh* for P, and returns as output a boolean value *success*. If *success* is true, then the program fully satisfies the set of behaviours *SBeh* and, in particular, every procedure call (allowed by *SBeh*) terminates.
The procedure `analyze_program` analyses each procedure $p \in \mathcal{P}$ for each $\langle B, se\rangle \in Beh_p$ in *SBeh*. It also computes the glb-closure of the set of behaviours *SBeh* through the function `MAKE_SAT`. This is useful when analysing an atom which is a procedure call: in that case, a look-up to the set *sat* is performed. Formally, `MAKE_SAT`(*SBeh*) returns a family $sat = \langle sat_p\rangle_{p\in\mathcal{P}}$ of sets of abstract sequences such that for all $p \in \mathcal{P}$, sat_p is the smallest set containing $\{B \mid \exists se : \langle B, se\rangle \in Beh_p\}$ which is closed under glb, i.e., $B_1, B_2 \in sat_p \Rightarrow B_1 \sqcap B_2 \in sat_p$.

Analysis of a Procedure. The result B_{out} of the analysis of a procedure $p \in \mathcal{P}$ consisting of a sequence $c_1, \ldots, c_r$ of clauses is obtained by "concatenating" the results $B_1, \ldots, B_r$ of the abstract execution of each clause through the operation `CONC`, i.e., $B_{out} = \texttt{CONC}(B_1, \ldots, B_r)$, the last being a shortcut for $\texttt{CONC}(\ldots\texttt{CONC}(B_{r-1}, B_r)\ldots)$. Formally, if B_1 and B_2 are the results of two clauses called with the same input β_{in}, then $\texttt{CONC}(B_1, B_2)$ produces an abstract sequence B' such that $\langle \theta, S_1\rangle \in Cc(B_1)$ and $\langle \theta, S_2\rangle \in Cc(B_2)$ imply $\langle \theta, S_1 :: S_2\rangle \in Cc(B')$.

Analysis of a Clause. The analysis of a clause $c \equiv p(X_1, \ldots, X_n)$ `:-` $l_1, \ldots, l_s$. with respect to $\langle B, se\rangle \in Beh_p$ consists in computing the abstract sequences $B_0, \ldots, B_s$, and B_{out} as follows.
Let β_{in} be the input substitution of B. First, B_0 is obtained from β_{in} through the operation `EXTC` which extends the domain of β_{in} to the set of all variables occurring in the clause, and gets an abstract sequence out of it. Then, for all $k \in \{1, \ldots, s\}$, B_k is computed from B_{k-1} and l_k in three steps as follows. Let $X_{i_1}, \ldots, X_{i_n}$ be the variables occurring in l_k. (1) The domain of the output abstract substitution β_{out} of B_{k-1} is restricted to the variables $X_{i_1}, \ldots, X_{i_n}$

and renamed into $X_1, \ldots, X_n$ through the operation RESTRG. (2) The literal l_k is executed with β^k_{inter}: this operation depends on the form of l_k and returns an abstract sequence B^k_{aux}. (3) This result is propagated on B_{k-1} by computing $B_k = \texttt{EXTGS}(l_k, B_{k-1}, B^k_{aux})$. In general, for a literal l with variables $X_{i_1}, \ldots, X_{i_n}$ and abstract sequences B_1 and B_2, $\texttt{EXTGS}(l, B_1, B_2)$ returns an abstract sequence B' such that: if $\langle \theta, < \theta_1, \ldots, \theta_q > \rangle \in Cc(B_1)$, and for all $j \in \{1, \ldots, q\}$ $\theta_j = \{X_1/t^j_1, \ldots, X_m/t^j_m\}$, $\theta'_j = \{X_1/t^j_{i_1}, \ldots, X_n/t^j_{i_n}\}$, $\langle \theta'_j, < \theta^j_1, \ldots, \theta^j_{p_j} > \rangle \in Cc(B_2)$, and for all $k \in \{1, \ldots, p_j\}$ $\exists \sigma^j_k : \ \theta^j_k = \theta'_j \sigma^j_k$ with $dom(\sigma^j_k) \subseteq codom(\theta'_j)$, and $S_j =< \theta_j \sigma^j_1, \ldots, \theta_j \sigma^j_{p_j} >$, then $\langle \theta, S_1 :: \ldots :: S_q \rangle \in Cc(B')$.
The execution of l_k with β^k_{inter} is as follows. We distinguish four cases.

1. If l_k is a built-in of the form $X_{i_1} = X_{i_2}$ then $B^k_{aux} = \texttt{UNIF_VAR}(\beta^k_{inter})$. In general, given an abstract substitution β with $dom(\beta) = \{X_1, X_2\}$, $\texttt{UNIF_VAR}(\beta)$ computes an abstract sequence B' such that for all $\theta \in Cc(\beta)$, if $\sigma \in mgu(X_1\theta, X_2\theta)$ then $\langle \theta, < \theta\sigma > \rangle \in Cc(B)$, otherwise, if $mgu(X_1\theta, X_2\theta) = \emptyset$ then $\langle \theta, < > \rangle \in Cc(B)$.
2. If l_k is of the form $X_{i_1} = f(X_{i_2}, \ldots, X_{i_n})$ then $B^k_{aux} = \texttt{UNIF_FUNC}(\beta^k_{inter}, f)$. This operation is similarly defined to the previous one.
3. If l_k is a non-recursive call $q(X_{i_1}, \ldots, X_{i_m})$ (i.e., $q \neq p$) then the analyser looks at sat, the glb-closed set of behaviours, to find an abstract sequence general enough to give information about this call. More precisely, it searches sat_q for the most precise abstract sequence B^k_{aux} such that β^k_{inter} is more general than the input abstract substitution of B^k_{aux}. This is realized by the operation $\texttt{LOOKUP}(\beta^k_{inter}, q, sat)$.
4. If l_k is a recursive call $p(X_{i_1}, \ldots, X_{i_n})$ then the analyser also applies the operation $\texttt{CHECK_TERM}(l_k, B_{k-1}, se)$ which checks whether the size of the arguments decreases, i.e., for all $\langle \theta, S \rangle \in Cc(B_{k-1})$ and $\theta' \in Subst(S)$,

$$se(\langle \|X_{i_1}\theta'\|, \ldots, \|X_{i_n}\theta'\| \rangle) < se(\langle \|X_1\theta\|, \ldots, \|X_n\theta\| \rangle).$$

The last step is the computation of B_{out} from B_s. This is realized by the operation RESTRC that restricts the output of B_s to the variables in the head of c.

Mutual Recursion. Mutual recursion is treated by extending the termination test to all mutual recursive procedures (above, such a test is applied only to recursive procedures). In practice, if p and q are mutually recursive procedures with $\langle B_p, se_p \rangle \in Beh_p$ and $\langle B_q, se_q \rangle \in Beh_q$, then, at point 3 above, we also check whether the size of the arguments decreases, i.e., for all $\langle \theta, S \rangle \in Cc(B_{k-1})$ and $\theta' \in Subst(S)$, $se_q(\langle \|X_{i_1}\theta'\|, \ldots, \|X_{i_m}\theta'\| \rangle) < se_p(\langle \|X_1\theta\|, \ldots, \|X_n\theta\| \rangle)$. This test ensures that the mutually recursive procedures will not loop infinitely.
Mutual recursive procedures are found out by a first-stage analysis which returns all pairs $(\langle p, B_p, se_p \rangle, \langle q, B_q, se_q \rangle)$ describing possibly mutual recursive calls. The termination test is then realized only when the triplets associated with the subcall and the head call are "mutually recursive".

Correctness of the analyser has been proven in [10]. We recall here the main theorem and we sketch the proof.

Theorem 4. *Let P be a program and SBeh be a set of behaviours for P. The procedure* `analyse_program` *called with P and SBeh as inputs returns a boolean value* success *as output such that if* success *is true, then SBeh is consistent with respect to the concrete semantics of the program P.*

Proof. (sketch) In order to prove the theorem, we first we introduce the notion of *procedure call allowed by SBeh* which is a tuple $t = \langle \theta, B, se, p \rangle$ such that $p \in \mathcal{P}$, $\langle B, se \rangle \in Beh_p$ and θ is described by the input abstract substitution of B. We also introduce a *well-founded relation* on the set of all the allowed procedure calls as follows: for any allowed procedure calls $t = \langle \theta, B, se, p \rangle$ and $t' = \langle \theta', B', se', p' \rangle$, $t < t'$ iff either the procedure p' is used in the definition of p or t' is a (mutually) recursive call that may be reached during the execution of t. In the second case, we also require that the "size" of θ' is (strictly) smaller than the "size" of θ, i.e., $se_{p'}(\langle \|X_{i_1}\theta'\|, \ldots, \|X_{i_m}\theta'\| \rangle) < se_p(\langle \|X_1\theta\|, \ldots, \|X_n\theta\| \rangle)$, to ensure that the execution terminates.

The proof is done by induction on the ordering on allowed procedure calls: we assume that *SBeh* correctly describes the executions of all procedure calls t' such that $t' < t$ and that these executions terminate and we prove that the execution of t terminates and is correcly described by *SBeh*. □

6 Unification Operation

This section is intended to give insights on how abstract operations are implemented and, in particular, how size information is computed. We do it by discussing the implementation of the unification operation `UNIF_VAR`. For a complete description of the abstract operations used by the analyser and the corresponding correctness proofs, the reder is referred to [10].

Unification Operation The unification operation `UNIF_VAR` is used for executing built-ins of the form $X_i = X_j$ with an input abstract substitution β. It returns an abstract sequence $B' = \langle \beta'_{in}, \beta'_{ref}, \beta'_{out}, E'_{ref_out}, E'_{sol} \rangle$.
The principle of the implementation is the following: first, we (re)use the operation $\texttt{UNIF_VAR}_{old}$ on abstract substitutions (see [14]) to compute β'_{out} describing the result of the execution of $X_i = X_j$ called with β. Then, in order to refine β to the set of $\theta \in Cc(\beta)$ for which the unification succeeds, we establish a mapping between the indices of β and the indices of β'_{out} representing the corresponding terms (we call such a mapping *constrained mapping*). Next, based on the above correspondence, we refine the information on terms provided by β, by an operation called $\texttt{REF}_{ref}$ producing β'_{ref}. Finally, we derive constraints between the size of terms in β'_{ref} and β'_{out} as well as constraints on the number of solutions.
Here, we only state the specifications of the operations $\texttt{UNIF_VAR}_{old}$ and $\texttt{REF}_{ref}$. The implementation of `UNIF_VAR` is detailed in terms of such operations.

$\texttt{UNIF_VAR}_{old}(\beta) = \langle \beta_{out}, ss, sf, tr, U \rangle$. This operation is similar [6] to the one de-

[6] Actually, the signature in [14] of this operation is $\texttt{UNIF_VAR}(\beta) = \beta'$, as there was no need there to export sure success/failure information. Adapting that definition to our purposes is straightforward. That's why we call it simply $\texttt{UNIF_VAR}_{old}$.

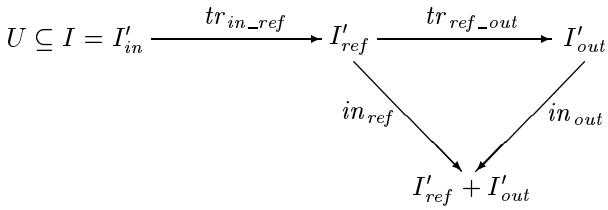
fined in [14]. Given an abstract substitution β with $dom(\beta) = \{X_1, X_2\}$, it returns an abstract substitution β_{out} describing the unification of $X_1\theta$ and $X_2\theta$ for all $\theta \in Cc(\beta)$; two boolean values ss and sf specifying whether sure success or sure failure can be inferred at the abstract level, a constrained mapping tr between the indices of β and the indices of β_{out} representing corresponding terms before and after the unification, and a set of indices U representing the set of terms in θ whose norm is not affected by the instantiation. The latter will allow us to establish precise constraints between the size of terms in β'_{ref} and β'_{out}.

$\mathtt{REF}_{ref}(\beta_1, \beta_2, tr) = \langle \beta', tr' \rangle$. This operation takes as inputs two abstract substitutions β_1 and β_2 and a constrained mapping tr between the indices of β_1 and β_2. It refines the abstract substitution β_1 by keeping substitutions in $Cc(\beta_1)$ that have at least an instance in $Cc(\beta_2)$. It returns an abstract substitution β' and a constrained mapping tr' between the indices of β' and β_2 such that β' is at least as precise as β_1 and $\theta_k \in Cc(\beta_k)$ $(k = 1, 2)$ with $\theta_2 \leq \theta_1$ implies $\theta_1 \in Cc(\beta')$.

$\mathtt{UNIF_VAR}(\beta) = B'$. Let β be an abstract substitution with $dom(\beta) = \{X_1, X_2\}$ and $\langle \beta_{out}, tr, ss, sf, U \rangle = \mathtt{UNIF_VAR}_{old}(\beta)$. B' is defined as follows.

$$
\begin{array}{lll}
\beta'_{in} & = \beta & \\
\beta'_{out} & = \beta_{out} & \\
\langle \beta'_{ref}, tr_{ref_out} \rangle & = \langle \beta'_{in}, tr \rangle & \text{if } ss \\
 & \quad \langle \bot, undef \rangle & \text{if } sf \\
 & \quad \mathtt{REF}_{ref}(\beta'_{in}, \beta'_{out}, tr) & \text{if } \neg ss \text{ and } \neg sf \\
E'_{ref_out} & = \bot & \text{if } sf \\
 & \quad \{\mathtt{sz}(in_{ref}(i)) = \mathtt{sz}(in_{out}(tr_{ref_out}(i))) : & \\
 & \quad i \in tr_{in_ref}(U)\} & \text{otherwise} \\
E'_{sol} & = \{sol = 1\} & \text{if } ss \\
 & \quad \bot & \text{if } sf \\
 & \quad \{0 \leq sol, sol \leq 1\} & \text{if } \neg ss \text{ and } \neg sf.
\end{array}
$$

where tr_{in_ref} is a canonical inclusion, and the following commutative diagram is satisfied by tr_{in_ref}, tr_{ref_out} and the injection functions in_{ref} and in_{out}.



The accuracy of this operation may be improved in practice by using a reexecution strategy [15]: we may repeatedly apply $\mathtt{UNIF_VAR}_{old}$ and $\mathtt{REF}_{ref}$ to β'_{ref} until sure success or sure failure is inferred or β'_{ref} stabilizes.

References

1. K.R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.
2. C. Braem, B. Le Charlier, S. Modard, and P. Van Hentenryck. Cardinality Analysis of Prolog. In M. Bruynooghe, editor, *Proc. of the International Logic Programming Symposium (ILPS'94)*, Ithaca NY, USA, November 1994. MIT Press.
3. A. Cortesi, B. Le Charlier, and S. Rossi. Specification-Based Automatic Verification of Logic Programs. In *Logic Program Synthesis and Transformation. Proceedings of the 6th International Workshop, LOPSTR'96*, volume 1207 of *LNCS*. Springer Verlag, August 1996.
4. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combination of Abstract Domains for Logic Programming. In *Proc. of the 21th ACM Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, January 1994.
5. P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
6. P. De Boeck and B. Le Charlier. Static Type Analysis of Prolog Procedures for Ensuring Correctness. In *PLILP'90*, LNCS 456, Springer-Verlag, pages 222–237, Linköping, Sweden, 1990.
7. Y. Deville. *Logic Programming: Systematic Program Development.* MIT Press, 1990.
8. Thomas W. Getzinger. The Costs and Benefits of Abstract Interpretation-driven Prolog Optimization. In *SAS'94*, LNCS 864, Springer-Verlag, pages 1–25, 1994.
9. J. Henrard and B. Le Charlier. FOLON: An Environment for Declarative Construction of Logic Programs. In M. Bruynooghe and M. Wirsing, editors, *PLILP'92*, LNCS 631, Springer-Verlag, Leuven, 1992.
10. B. Le Charlier, C. Leclère, S. Rossi, and A. Cortesi. Automated Verification of Prolog Programs. Technical Report RP-97-003, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, March 1997.
11. B. Le Charlier and S. Rossi. Sequence-Based Abstract Semantics of Prolog. Technical Report RR-96-001, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, February 1996.
12. B. Le Charlier, S. Rossi, and P. Van Hentenryck. An Abstract Interpretation Framework Which Accurately Handles Prolog Search-Rule and the Cut. In M. Bruynooghe, editor, *ILPS'94*, Ithaca NY, USA, November 1994. MIT Press.
13. B. Le Charlier, S. Rossi, and P. Van Hentenryck. Sequence-Based Abstract Interpretation of Prolog. Technical Report RR-97-001, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, January 1997.
14. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):35–101, January 1994.
15. B. Le Charlier and P. Van Hentenryck. Reexecution in Abstract Interpretation of Prolog. *Acta Informatica*, 32:209–253, 1995.
16. P. Van Roy. 1983–1993 : The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19/20:385–441, 1994.
17. D. K. Wilde. A Library for Doing Polyhedral Operations. Technical Report No. 785, IRISA, Rennes Cedex-France, 1993.

A The Algorithm of the Analyser

This appendix contains the implementation of the three main procedures of our analyser, namely `analyse_program`, `analyse_procedure` and `analyse_clause`.

```
PROCEDURE analyze_program(P, SBeh) =
    success := true
    sat := MAKE_SAT(SBeh)
    for all p ∈ 𝒫,  for all ⟨B, se⟩ ∈ Beh_p
        success := success ∧ analyse_procedure(p, B, se)
    return success.
```

```
PROCEDURE analyze_procedure(p, B, se) =
    for k := 1 to r do
        ⟨success_k, B_k⟩ := analyze_clause(c_k, B, se)
    if there exists k ∈ {1, ..., r} such that ¬success_k,
        then  success := false
        else  B_out := CONC(B_1, ..., B_r)
              success := (B_out ≤ B)
    return success.
```

```
PROCEDURE analyze_clause(c, B) =
    β_in := input(B)
    B_0 := EXTC(c, β_in)
    for k := 1 to s do
        β^k_inter := RESTRG(l_k, B_{k-1})
        if l_k ≡ X_{i_1} = X_{i_2} then B^k_aux := UNIF_VAR(β^k_inter)
        if l_k ≡ X_{i_1} = f(X_{i_2}, ..., X_{i_m}) then B^k_aux := UNIF_FUNC(β^k_inter, f)
        if l_k ≡ q(X_{i_1}, ..., X_{i_m}) and q ≠ p then
                    ⟨B^k_aux, success_k⟩ := LOOK_UP(β^k_inter, q, sat)
        if l_k ≡ p(X_{i_1}, ..., X_{i_m}) then
                    ⟨B^k_aux, success'_k⟩ := LOOK_UP(β^k_inter, q, sat)
                    success_k := success'_k ∧ CHECK_TERM(l_k, B_{k-1}, se)
        B_k := EXTGS(l_k, B_{k-1}, B^k_aux)
    if there exists k such that
           either l_k ≡ q(X_{i_1}, ..., X_{i_m}) ∧ ¬success_k
           or l_k ≡ p(X_{i_1}, ..., X_{i_n}) ∧ (¬success_k ∨ β^k_inter ⋠ β_in)
        then success = false
        else success = true and B_out = RESTRC(c, B_s)
    return⟨success, B_out⟩.
```

This article was processed using the LaTeX macro package with LLNCS style