Automated Behavioural Verification of Prolog Programs*

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

[1]Institut d'Informatique, 21 rue Grandgagnage, B-5000 Namur, Belgium

[2]Dip. di Matematica, via Belzoni 7, 35131 Padova, Italy

[3]Dip. di Matematica ed Informatica, via Torino 155, 30173 Venezia, Italy

Abstract

Although Prolog is still the most widely used logic language, it suffers from a number of drawbacks which prevent it from being truely declarative. Several authors have proposed methodologies to reconcile declarative programming with the algorithmic features. The idea is to analyse the logic program with respect to a set of properties such as modes, types and termination in order to ensure that the operational behaviour of the program complies with its logic meaning. In this paper, we present an analyser which allows one to integrate many individual analyses previously proposed in the literature as well as new ones. Conceptually, the analyser is based on the notion of abstract sequence which makes it possible to collect all kinds of desirable information including for instance determinacy and multiplicity of a procedure.

Keywords: Program Verification, Static Analysis, Logic Programming, Prolog.

1 Introduction

The implementation of declarative languages often includes "impure" features which are intended to improve the efficiency of the language but ruin its declarative nature. This is what happens in logic programming with Prolog.

Various forms of program analyses have been investigated by numerous researchers in order to improve on this situation. Some analyses aim at optimizing programs automatically, relieving the programmer from using impure control features. Other analyses attempt to verify that a non declarative implementation of a program in fact behaves accordingly to its declarative meaning. In this paper, we describe an analyser for verifying whether a Prolog program behaves according to a given specification. Our analyser is characterized by the following main features.

1. The analyser is based on the methodology of abstract interpretation in the sense that it approximates concrete executions of programs. However, it differs from

*Extended Abstract of [14].

most abstract interpretation frameworks by the fact that no fixpoint computation is performed. Instead, it makes use of information on possible program executions, namely *behaviour*, provided by the user.

2. The analyser integrates several analyses previously described in the literature. In particular, it allows one to infer not only information like modes, types, sharing, and term sizes which is easily computed within classical abstract interpretation frameworks ([5, 18, 19]), but also information like determinacy and cardinality (i.e., the number of solutions) which is directly related to the operational semantics of Prolog. Our analyser uses a notion of *abstract sequence* which is powerful enough to express and combine all kinds of useful information about Prolog program executions. To the best of our knowledge, no previous framework was able to incorporate such information in a single analysis.
3. The notion of abstract sequence used by the analyser is novel with respect to the one introduced in [4, 16, 17]. It describes a set of pairs of the form $\langle \theta, S \rangle$, where θ and S respectively denote an (input) substitution and the sequence of answer substitutions resulting from executing (part of) a program with this input. Such notion is more "relational" than the one in [4, 16, 17] since it allows us to relate θ and S explicitly, contrary to the old notion which only abstracts sets of sequences. So, we may relate the number of solutions and the size of output terms to the size of input terms in full generality (e.g., we can relate the input and output sizes of the *same*[1] term without requiring any invariance under instantiation). To the best of our knowledge, such generality was not available in previous frameworks for term size analysis.
4. The practical implementation of the analyser is based on the generic system *GAIA* [18]. Although our analyser significantly differs from the fixpoint algorithm of *GAIA*, a large part of the code of *GAIA* can be reused for the algorithms and the domains. At the time of writing, the implementation is still underway.

Our analyser has a number of interesting applications. To cite some, it could be integrated in a programming environment to check the 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 by Deville in [12], it could be integrated in the FOLON environment which was developed for supporting the automatable aspects of Deville's methodology for logic program construction. The automatable aspects of other works on verification (e.g., [1]) can be integrated to our system. 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 our analyser allows one to verify precise relations between the size of the arguments and the number of solutions to a Prolog procedure, it can be used as a basis for an automatic complexity analysis (see [11]).
The paper is organized as follows. Section 2 provides an overview of the analyser. Section 3 defines basic concepts and notations. Section 4 illustrates our domain of abstract sequences. Section 5 describes the analyser. Section 6 concludes the paper.

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

2 Overview of the Analyser

We show on an example the functionalities that we want to achieve. Consider the procedure `select/3` depicted in Figure 1. Declaratively, it defines a relation between

```
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).
```

Figure 1: The Procedure `select/3`

three terms[2] X, L, and LS that holds if and only if 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 declarative specification but instead it checks a number of operational properties which ensure that Prolog actually computes the specified relation (provided that the procedure is "declaratively" correct). So, for the sake of the example, 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 above class of input calls;
2. β_{ref} is an abstract substitution giving an over approximation of the successful input calls, i.e., those that produce at least one solution: here, β_{ref} states that L is a non empty list and is an *exact* description of the set of successful calls;
3. β_{out} is an abstract substitution giving an over approximation of the set of outputs corresponding to the successful calls: here, β_{out} states that X is a ground term and LS is a ground list;
4. 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 case, E_{ref_out} states that the input length of L is equal to the output length of LS plus 1;
5. 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 case, 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 expression must decrease strictly through recursive calls. In our case, se is equal to `L` representing the input length of L, denoted by $\|L\|$.

From the procedure and the above information, the analyser computes a number of abstract sequences: one for every prefix of the body of every clause, one for every clause, and one for the complete procedure. For instance, in our example, the analyser computes an abstract sequence B_1 expressing that, for the specified class of input calls,

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

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 because X and LS are free *and do not share*. Moreover, the analyser needs an abstract sequence describing the behaviour of the procedure `list/1`. The abstract sequence states that, for ground calls, the literal succeeds only for list and exactly once.

The second clause contains a recursive call, which deserves a special treatment. First the analyser is able to infer that the recursive call will be executed at most once and 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. Thus, we can assume 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 TS is a ground list whose size is the same of T minus 1. Putting all pieces together, the analyser computes the abstract sequence B_2, which states that the second clause succeeds only for a list L of at least two elements (and actually succeeds for all of them), the output size of LS is equal to the size of L minus 1, the number of solution is also equal to the size of L minus 1.

The next step is to combine the abstract sequences B_1 and B_2 to get an abstract sequence B_{out} describing the behaviour of the whole procedure. 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, so the procedure succeeds $\|L\|$ times. However, when the length of L is equal to 1, the second clause fails and the first one succeeds once; so the procedure also 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 thus correct.

The previous discussion is intended to give insights into how all kinds of information interact to produce an accurate analysis. The complete formalization of this process is given in [14]. In the rest of the paper we will sketch its main features.

3 Preliminaries

The reader is assumed to be familiar with the basic concepts of logic programming and abstract interpretation [8, 20]. We denote by $\mathcal{T}$ the set of all terms, and by I (possibly subscripted) a set of indices. $\mathcal{T}^I$ is the set of all tuples of terms $\langle t_i \rangle_{i \in I}$ and $\mathcal{T}_I^*$ is the set of all "frames" $f(i_1, \ldots, i_n)$ where f is an n-ary functor and $i_1, \ldots, i_n \in I$. A size measure is a function $\|\cdot\| : \mathcal{T} \to \mathbf{N}$, [3, 10]. 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 sets A and B is an arbitrarily set, denoted by $A + B$, equipped with two injections functions in_A and in_B satisfying the property: for any set C and for any functions $f_A : A \to C$ and $f_B : B \to C$, there exists a unique function $f : A + B \to C$ such that $f_A = f \circ in_A$ and $f_B = f \circ in_B$ (the symbol $\circ$ is the usual function composition). In the following, we denote the function f by $f_A + f_B$.

Let V be a set of variables. We denote by $\mathbf{Exp}_V$ the set of all linear expressions with integer coefficients on the set of variables 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.

Programs are assumed to be normalized. A *normalized program* P is a non empty set of procedures pr. A procedure is a non empty sequence of clauses c. A clause has the form $h\colon -g$ 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 $X_{i_1} = X_{i_2}$, or a built-in $X_{i_1} = f(X_{i_2},\ldots,X_{i_n})$ where f is a functor of arity $n-1$, or a procedure call $p(X_{i_1},\ldots,X_{i_n})$. 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.

A *(program) substitution* θ is a finite set $\{X_1/t_1,\ldots,X_n/t_n\}$ where $X_1,\ldots,X_n$ are distinct program variables and the t_i are terms. $\{X_1,\ldots,X_n\}$ is the domain of θ, denoted by $dom(\theta)$. We say that θ_1 is *more general* than θ_2, denoted by $\theta_2 \leq \theta_1$, iff there exists σ such that $\theta_2 = \theta_1\sigma$. $mgu(t_1,t_2)$ denotes the set of substitutions that are a most general unifier of t_1 and t_2. The *restriction* of θ to a set of variables $D \subseteq dom(\theta)$, 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 $< \theta_1,\ldots,\theta_n >$ $(n \geq 0)$ where the θ_i are (program) substitutions with the same domain D. $Subst(S)$ is the set of all substitutions in S. The symbol :: denotes sequence concatenation.

Our analyser is expected to simulate the concrete semantics of Prolog programs. We refer to the concrete semantics presented in [17] and proven equivalent to Prolog operational semantics in [15]. A complete description of this semantics is not possible here. We just view the concrete semantics for a Prolog program P as 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. Since our analyser relies on an induction argument about the size of input terms, we can restrict to finite sequences[3]. The fact that $\langle\theta,p\rangle$ is mapped to the sequence S is denoted by $\langle\theta,p\rangle \longmapsto S$ meaning that the execution of $p(X_1,\ldots,X_n)\theta$ terminates producing the (finite) sequence of answer substitutions S.

4 Abstract Domains

Our domain of *abstract substitutions* is an instantiation of the generic domain `Pat`($\Re$) [6]; in particular, it is an extension (with type information) of the domain `Pattern` [18]. Our domain of *abstract sequences* is novel and different from [17]. The notion of *behaviour* constitutes the full package of information provided by the user.

Abstract Substitutions An abstract substitution β over variables $X_1,\ldots,X_n$ is a triplet $\langle sv, frm, \alpha\rangle$ where sv is a function from $\{X_1,\ldots,X_n\}$ to a set of indices I, frm is a partial function from I to $\mathcal{T}_I^*$, and α describes modes, types and possible sharing of some terms. An abstract substitution β represents a set of program substitutions $\{X_1/t_1,\ldots,X_n/t_n\}$. β can provide information not only about $t_1,\ldots,t_n$ but also

[3] Note that the semantics in [15] also considers infinite and (so-called) incomplete sequences.

about subterms of them. The terms described in β are denoted by indices.
The *same-value* component sv maps each X_i to the index corresponding to t_i. It may express equality constraints between two variables X_i and X_j, when $sv(X_i) = sv(X_j)$. The value of $frm(i)$, when it is defined, is a term $f(i_1, \ldots, i_n)$, meaning that t_i is of the form $f(t_{i_1}, \ldots, t_{i_n})$. The *abstract tuple* α provides information about modes, types and possible sharing of the t_i. In this paper, we mainly use the modes *ground* and *var* describing, respectively, all ground terms and all terms which are variables. We also consider the types *list*, *anylist*, *any*, describing, respectively, all lists, all terms that can be instantiated to a list, and all terms.

Definition 1 [Abstract Substitution] An *abstract substitution* β over a set of indices I is either $-$ or a triplet of the form $\langle sv, frm, \alpha \rangle$ where

- sv is a function, $sv : \{X_1, \ldots, X_n\} \rightarrow I$ where $\{X_1, \ldots, X_n\} = dom(\beta)$;
- frm is a partial function, $frm : I \not\rightarrow \mathcal{T}_I^*$;
- α is a triplet of the form $\langle mo, ty, ps \rangle$ where mo is a function from I to a set of modes, ty is a function from I to a set of types, and ps is a binary and symmetrical relation on $I \times I$.

β describes the set $Cc(\beta)$ of all concrete substitutions $\theta = \{X_1/t_{i_1}, \ldots, X_n/t_{i_n}\}$ such that there exists a tuple of terms $\langle t_i \rangle_{i \in I}$ satisfying: for all $X_i \in dom(\theta)$, $X_i\theta = t_{sv(X_i)}$; for all $i \in I$, if $frm(i) = f(i_1, \ldots, i_n)$ then $t_i = f(t_{i_1}, \ldots, t_{i_n})$; for all $i \in I$, both $mo(i)$ and $ty(i)$ describe t_i; for all $i, j \in I$, if $Var(t_i) \cap Var(t_j) \neq \emptyset$ then $(i, j) \in ps$.

Obviously, $Cc(-) = \emptyset$. Moreover, in general, the tuple of terms $\langle t_i \rangle_{i \in I}$ is not unique, for a fixed $\theta \in Cc(\beta)$. We denote by $\texttt{DECOMP}(\theta, \beta)$ the set of all such tuples.

Example 2 Consider the behaviour for *select*/3 informally described in Section 2. The abstract substitutions β_{in}, β_{ref} and β_{out} can be represented as follows, where the question mark denotes an undefined value.

$\beta_{in} = \langle sv_{in}, frm_{in}, \alpha_{in} \rangle$ where $\alpha_{in} = \langle mo_{in}, ty_{in}, ps_{in} \rangle$ with

sv_{in} :		frm_{in} :		mo_{in} :		ty_{in} :	
	$X_1 \mapsto 1$		$1 \mapsto ?$		$1 \mapsto var$		$1 \mapsto anylist$
	$X_2 \mapsto 2$		$2 \mapsto ?$		$2 \mapsto ground$		$2 \mapsto any$
	$X_3 \mapsto 3$		$3 \mapsto ?$		$3 \mapsto var$		$3 \mapsto anylist$

$ps_{in} = \{(1,1), (3,3)\}$

$\beta_{ref} = \langle sv_{ref}, frm_{ref}, \alpha_{ref} \rangle$ where $\alpha_{ref} = \langle mo_{ref}, ty_{ref}, ps_{ref} \rangle$ with

sv_{ref} :		frm_{ref} :		mo_{ref} :		ty_{ref} :	
	$X_1 \mapsto 1$		$1 \mapsto ?$		$1 \mapsto var$		$1 \mapsto anylist$
	$X_2 \mapsto 2$		$2 \mapsto [4\|5]$		$2 \mapsto ground$		$2 \mapsto list$
	$X_3 \mapsto 3$		$3 \mapsto ?$		$3 \mapsto var$		$3 \mapsto anylist$
			$4 \mapsto ?$		$4 \mapsto ground$		$4 \mapsto any$
			$5 \mapsto ?$		$5 \mapsto ground$		$5 \mapsto list$

$ps_{ref} = \{(1,1), (3,3)\}$

$\beta_{out} = \langle sv_{out}, frm_{out}, \alpha_{out} \rangle$ where $\alpha_{out} = \langle mo_{out}, ty_{out}, ps_{out} \rangle$ with

sv_{out} :	$X_1 \mapsto 1$	frm_{out} :	$1 \mapsto ?$	mo_{out} :	$1 \mapsto ground$	ty_{out} :	$1 \mapsto any$
	$X_2 \mapsto 2$		$2 \mapsto [4\|5]$		$2 \mapsto ground$		$2 \mapsto list$
	$X_3 \mapsto 3$		$3 \mapsto ?$		$3 \mapsto ground$		$3 \mapsto list$
			$4 \mapsto ?$		$4 \mapsto ground$		$4 \mapsto any$
			$5 \mapsto ?$		$5 \mapsto ground$		$5 \mapsto list$

$ps_{out} = \emptyset$

Abstract Sequences An abstract sequence B is $\langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$ where β_{in}, β_{ref} and β_{out} are abstract substitutions and E_{ref_out} and E_{sol} are so-called size components. β_{in} represents the input; β_{ref} is a refinement of β_{in} and approximates the set of substitutions in $Cc(\beta_{in})$ that surely succeeds; β_{out} describes the output. 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. In this paper, a size component E over I is a system of linear equations and inequations over $\mathbf{Exp}_I$ representing the set $Cc(E)$ of all tuples $\langle n_i \rangle_{i \in I} \in \mathbf{N}^I$ which are solutions of E. In order to distinguish variables of I from integer coefficient and constants in elements of $\mathbf{Exp}_I$, we wrap up each i of I into $\mathbf{sz}(i)$. Moreover, we write (in)equations between double brackets $[\![\cdots]\!]$, meaning that they are syntactic objects. If $f(i) = i'$ and $f(j) = j'$ for a function f, $[\![\mathtt{sz}(f(i)) = \mathtt{sz}(f(j)) + \mathtt{1}]\!]$ represents the syntactical equation $\mathtt{sz}(i') = \mathtt{sz}(j') + \mathtt{1}$.

Definition 3 [Abstract Sequence] An *abstract sequence* B is either $-$ 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}; it is always different from $-$;
- β_{ref} is an abstract substitution over I_{ref} with $Cc(\beta_{ref}) \subseteq Cc(\beta_{in})$;
- 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 $I_{ref} + \{sol\}$;
- if β_{ref} or β_{out} or E_{ref_out} or E_{sol} is equal to $-$ then they are all equal to $-$;
- for all $\theta' \in Cc(\beta_{out})$, $\exists \theta \in Cc(\beta_{ref})$ such that $\theta'_{/dom(\beta_{ref})} \leq \theta$.

B represents the set $Cc(B)$ of all pairs $\langle \theta, S \rangle$ such that: $\theta \in Cc(\beta_{in})$; $Subst(S) \subseteq Cc(\beta_{out})$; 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 $\langle \|t_i\| \rangle_{i \in I_{ref}} + \{sol \mapsto |S|\} \in Cc(E_{sol})$.

Example 4 Let β_{in}, β_{ref}, and β_{out} be the abstract substitutions of the Example 2. Let $in_{ref} : I_{ref} \rightarrow I_{ref} + I_{out}$ and $in_{out} : I_{out} \rightarrow I_{ref} + I_{out}$ 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 $E_{ref\_out} = \{[\![\mathtt{sz}(in_{ref}(2)) = \mathtt{sz}(in_{out}(3)) + \mathtt{1}]\!]\}$ and $E_{sol} = \{[\![sol = \mathtt{sz}(in_{ref}(2))]\!]\}$.

Behaviours A behaviour for a procedure formalizes its behavioural properties.

Definition 5 [Behaviour] A *behaviour* Beh_p for a procedure name $p \in \mathcal{P}$ of arity 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 $\mathbf{Exp}_{\{X_1,\ldots,X_n\}}$.

Example 6 Let B be the abstract sequence of the Example 4. The behaviour for $select/3$ described in Section 2 can be represented by $\{\langle B, X_2\rangle\}$.

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 7 [Consistency of a Set of Behaviours] 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 of B terminates and $\langle \theta, p\rangle \mapsto S$ implies $\langle \theta, S\rangle \in Cc(B)$.

5 The Analyser

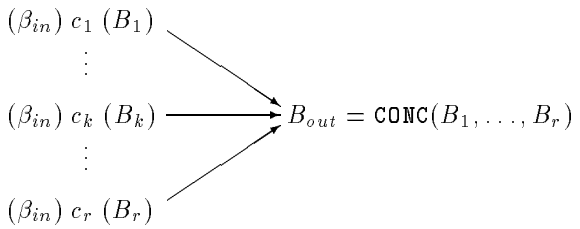
In this section we describe the analyser assuming that the analysed program contains no mutually recursive procedures (see [14]). The description of the analyser is developed in a top-down approach: first, we discuss the analysis of a program, then, the analysis of a procedure, finally, the analysis of a single clause.

Abstract Execution of a Program The analysis of a program is realized by the procedure `analyze_program` which takes as input a program P and a set of behaviours $SBeh$ for P, and returns a boolean value *success*. If *success* is true, then we can say that the program satisfies the specification represented by the behaviours; otherwise, either the program is not correct with respect to the specification, or the analysis is not precise enough to check its correctness. In [14] we proved that if *success* is true, then $SBeh$ is consistent with respect to the concrete semantics of P.

The procedure `analyze_program` uses the function `MAKE_SAT` to build the glb-closure of the set of bahaviours $SBeh$. More 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., $B_1, B_2 \in sat_p \Rightarrow B_1 \sqcap B_2 \in sat_p$.

```
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.
```

Abstract Execution of a Procedure Let $p \equiv c_1, \ldots, c_r$ be a procedure. The result B_{out} of the execution of p with β_{in} (where β_{in} is the input substitution of some abstract sequence B with $\langle B, se \rangle \in Beh_p$) is obtained by "concatenating" the results $B_1, \ldots, B_r$ of the abstract execution of each clause. More precisely, $B_{out} = \texttt{CONC}(B_1, \ldots, B_r)$, the last being a shortcut for $\texttt{CONC}(\ldots \texttt{CONC}(B_{r-1}, B_r) \ldots)$.

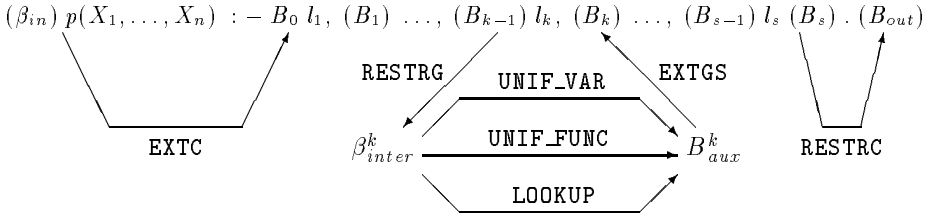


The procedure `analyze_procedure` succeeds if the computed abstract sequence B_{out} is more or equally precise than the abstract sequence B (with input substitution β_{in}).

```
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.
```

The operation `CONC` is the counterpart for abstract sequences of the operation `UNION`, used in [18], which simply collects the information of two abstract substitutions into a single one. `CONC` differs from `UNION` only for the computation of the number of solutions of a procedure which is the sum of the numbers of solutions of its clauses, not an "upper bound" of them. To obtain a good precision, we detect mutual exclusion of clauses [4, 17]. So, in the implementation of the `CONC` operation (see [14]), we compute the greatest lower bound of the β_{ref} of the two abstract sequences: if it is equal to $-$, then the clauses are exclusive, and we only collect the numbers of solutions; otherwise, we compute the sum of the numbers of solutions for this greatest lower bound only.

Abstract Execution of a Clause Let $c \equiv p(X_1, \ldots, X_n)\ \{: -\}\ l_1, \ldots, l_s.$ be a clause and $\langle B, se \rangle$ be an element of Beh_p. Let also β_{in} be the input substitution of B. The execution of the clause c with β_{in} may be computed as depicted below.



Let us see how the abstract sequences $B_0, \ldots, B_s$, and B_{out} are obtained.

- B_0 is obtained from β_{in} through the operation $\texttt{EXTC}(c, \beta_{in})$ which extends the domain of β_{in} to the set of all variables in c and returns an abstract sequence.
- Let $k \in \{1, \ldots, s\}$. The operation used to derive B_k from B_{k-1} depends on the form of the literal l_k. Four cases apply.

  1. l_k is a literal $X_{i_1} = X_{i_2}$. In this case, we restrict the domain of the output β_{out} of B_{k-1} to X_{i_1} and X_{i_2} and rename them into X_1 and X_2, by computing $\beta^k_{inter} = \texttt{RESTRG}(l_k, B_{k-1})$. Then, we execute the unification $B^k_{aux} = \texttt{UNIF\_VAR}(\beta^k_{inter})$. Finally, we compute the effect of this unification on the output substitution β_{out} of B_{k-1}: $B_k = \texttt{EXTGS}(l_k, B_{k-1}, B^k_{aux})$.
  2. l_k is a literal $X_{i_1} = f(X_{i_2}, \ldots, X_{i_n})$. In this case, we use the same process as above where $B^k_{aux} = \texttt{UNIF\_FUNC}(\beta^k_{inter}, f)$.
  3. l_k is a non-recursive call $q(X_{i_1}, \ldots, X_{i_m})$ where $q \neq p$. In this case, we restrict the domain of β_{out} to $X_{i_1}, \ldots, X_{i_m}$ and we rename them into $X_1, \ldots, X_m$ by $\texttt{RESTRG}$. Then, we look at sat and we search sat_q for the most precise abstract sequence B^k_{aux} such that $input(B^k_{aux}) \geq \beta^k_{inter}$. If there is no such abstract sequence, we give up the analysis since there is not enough information in $SBeh$. The search for B^k_{aux} is realized by the operation $\texttt{LOOK\_UP}(\beta^k_{inter}, q, sat)$. Once we have obtained B^k_{aux}, we compute the effect of the call l_k on B_{k-1} by $B_k = \texttt{EXTGS}(l_k, B_{k-1}, B^k_{aux})$.
  4. l_k is a recursive call $p(X_{i_1}, \ldots, X_{i_n})$. In this case, we test whether β^k_{inter} is less precise than β_{in} and, using the operation $\texttt{CHECK\_TERM}(l_k, B_{k-1}, se)$, we check whether 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, with the operation $\texttt{RESTRC}(c, B_s)$ that restricts the output domain of B_s to the variables in the head of c.

```
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_aoux)
   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⟩.
```

The reader may refer to [14] for a detailed description of the algorithms.

6 Conclusions

We have sketched the main theoretical and methodological aspects of an analyser for Prolog programs based on a verification approach. Implementing a complete analyser is a long term project. As we will attempt to complete such a project, we now discuss what remains to be done.

Analysing (Almost) Full Prolog. Since the correctness of our analyser is based on the concrete semantics of [17], all Prolog features that are simple to model in this framework can be easily integrated in the analyser. Arithmetic built-ins, such as `is` and `<`, and test predicates, such as `var` and `ground`, belong to these features. Interestingly, most of them can be handled without additional coding by providing behaviours capturing their operational semantics. The cut is a special control feature which requires to enhance the concrete and abstract domains with so-called "cut information". These aspects have been satisfactory solved in [4, 16, 17]. Thus we can integrate a treatment of the cut based on the same approach into our analyser.

Implementing a Complete Set of Domains. The abstract domain presented in this paper is conceptually generic, since it is based on the approach of [6]. However the particular instance that we have described is able to handle programs dealing with lists accurately, but not other programs. This restriction will be alleviated in the future by integrating a type domain similar to [13], as already described in [7]. We will also improve the treatment of sharing by adding a complementary domain for linearity information. Finally, we will attempt to design more powerful domains for the size components, based on non linear constraints and/or computer algebra.

Extending the Verification Scope of the Analyser. Our analyser assumes that the occur-check [2, 21] is performed during unification. It is nevertheless straightforward to enhance the operations `UNIF_VAR` and `UNIF_FUNC` with an additional result parameter specifying whether the occur-check is needed or not. Classical abstract domains (including sharing and linearity) will then allow us to solve the problem quite satisfactorily.

Finally, our analyser cannot deal with non terminating procedures at all (i.e., it always rejects them), but yet such procedures can sometimes be considered as correct if they "produce all their solutions" [12] or if the user is interested in merely "existential" termination (i.e., the procedures produce at least one solution) [9]. Our analyser can be extended to such situations. This would allow one to transform existentially terminating programs into (universally) terminating programs through automatic introduction of cuts.

References

[1] K.R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[2] K.R. Apt and A. Pellegrini. Why the Occur-Check is Not a Problem. In M. Bruynooghe and M. Wirsing, editors, *PLILP92*, pages 69–86, Leuven, Belgium, 1992. LNCS 631, Springer-Verlag.

[3] A. Bossi, N. Cocco, and M. Fabris. Norms on Terms and their Use in Proving Universal Termination of a Logic Program. *Theoretical Computer Science*, 124(2):297–328, 1994.

[4] C. Braem, B. Le Charlier, S. Modard, and P. Van Hentenryck. Cardinality Analysis of Prolog. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium (ILPS'94)*, Ithaca NY, USA, November 1994. MIT Press.

[5] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.

[6] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combination of Abstract Domains for Logic Programming. In *Proceedings of the 21th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, January 1994.

[7] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Type Analysis of Prolog using Type Graphs. *Journal of Logic Programming*, 23(3):237–278, June 1995.

[8] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[9] D. De Schreye and S. Decorte. Termination of Logic Programs: the Never-Ending Story. *Journal of Logic Programming*, Special anniversary edition, 1994. Accepted for publication.

[10] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A Framework for Analysing the Termination of Definite Logic Programs with respect to Call Patterns. In *Proc. FGCS'92*, pages 481–488, ICOT Tokyo, 1992. ICOT.

[11] S.K. Debray and N.W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.

[12] Y. Deville. *Logic Programming: Systematic Program Development.* MIT Press, 1990.

[13] 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.

[14] 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.

[15] 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.

[16] 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, *Proceedings of the International Logic Programming Symposium (ILPS'94)*, Ithaca NY, USA, November 1994. MIT Press.

[17] 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.

[18] 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.

[19] B. Le Charlier and P. Van Hentenryck. Reexecution in Abstract Interpretation of Prolog. *Acta Informatica*, 32:209–253, 1995.

[20] J.W. Lloyd. *Foundations of Logic Programming.* Springer Series: Symbolic Computation–Artificial Intelligence. Springer-Verlag, second edition, 1987.

[21] K. Marriott and H. Søndergaard. On Prolog and the Occur-Check Problem. *SIGPLAN Notices*, 24(5):76–82, 1989.