Specification-based Automatic Verification of Prolog Programs*

Agostino Cortesi[1] and Baudouin Le Charlier[2] and Sabina Rossi[3]

[1] Dipartimento di Matematica e Informatica, via Torino 155, 30173 Venezia, Italy
[2] Institut d'Informatique, 21 rue Grandgagnage, B-5000 Namur, Belgium
[3] Dipartimento di Matematica, via Belzoni 7, 35131 Padova, Italy

Abstract. The paper presents an analyzer for verifying the correctness of a Prolog program relative to a specification which provides a list of input/output annotations for the arguments and parameters that can be used to establish program termination. The work stems from Deville's methodology to derive Prolog programs that correctly implement their declarative meaning. In this context, we propose an algorithm that combines, adapts, and sometimes improves various existing static analyses in order to verify total correctness of Prolog programs with respect to formal specifications. Using the information computed during the verification process, an automatic complexity analysis can be also performed.

1 Introduction

Logic programming is an appealing programming paradigm since it allows one to solve complex problems in a concise and understandable way, i.e., in declarative style. For efficiency reasons, however, practical implementations of logic programs (e.g., Prolog programs) are not always faithful to their declarative meaning. In his book [11], Deville proposes a methodology for logic program construction that aims at reconciling the declarative semantics with an efficient implementation. The methodology is based on four main development steps:

1. elaboration of a specification, consisting of a description of the relation, type declarations, and a set of behavioural assumptions;
2. construction of a correct logic description, dealing with the declarative meaning only;
3. derivation of a Prolog program;
4. correctness verification of the Prolog code with respect to the specification.

The FOLON environment [8, 12] was designed with the main goal of supporting the automatable aspects of this methodology. In this context, we propose a new analyzer for verifying total correctness of Prolog programs with respect to specifications.

* Partly supported by European Community "HCM-Network, Logic Program Synthesis and Transformation, Contract Nr. CHRX-CT93-00414", and by "MURST NRP, Modelli della Computazione e dei Linguaggi di Programmazione".

The new analyzer is an extension of the analyzers described in [7, 8, 12] which themselves automate part of the methodology described in [11]. The novelty is that we deal with total correctness, while [7, 8, 12] only deal with particular correctness aspects such as mode and type verification. At this aim, we extend the generic abstract domain $\texttt{Pat}(\Re)$ [5, 6, 19] to deal with multiplicity and size relations.

We adapt the framework for termination analysis proposed by De Schreye, Verschaetse and Bruynooghe in [10]: instead of proving termination based on the program code only, we use termination information given in the formal specification; this requires more information from the programmer but allows for more general termination proofs.

The analyzer computes information about the number of solutions using the notion of abstract sequence which was firstly introduced in [15]. This is related to the cardinality analysis described in [2]. However, we do not perform an abstract interpretation in the usual sense since we use the formal specification of the subproblems instead of their code; moreover, we allow the number of solutions to be expressed as a function of the input argument sizes. This enhances the expressiveness of cardinality analysis with respect to [2].

The information about size relations and number of solutions can be also used to perform, in an automatic way, complexity analysis of Prolog programs, in the spirit of the framework proposed by Debray and Lin in [9]. In this paper, we only sketch this application on examples. Notice that, in the context of program development, complexity analysis can be used to choose the most efficient version of a procedure.

The paper is organized as follows. In Section 2, we recall Deville's methodology. In Section 3, we introduce the abstract domains used by the analyzer. In Section 4, we describe the analyzer and give an example to illustrate the main operations. Section 5 contains a formal description of the operations. In Section 6, we discuss the complexity analysis and describe some examples of cost analysis which use information computed by the analyzer. Section 7 concludes the paper.

2 Program Synthesis Methodology Overview

Our work is based on and extends Deville's methodology for logic program development presented in [11]. Let us illustrate it with the construction of the procedure *select*/3 which removes an occurrence of X from the list L containing it and returns the list L_s.

2.1 Specification

The first step consists in specifying the procedure according to a standard specification format. Indeed, we extend the general specification form proposed by Deville in [11] with extra information which will be useful for proving termination of the derived procedures. Argument size relations are specified in the form of a set of inequations (see [9, 10]). They are used to prove termination of the recursive procedures in the spirit of [10]. Moreover, the number of solutions is allowed to be expressed in terms of the input argument sizes.

A specification for *select*/3 is depicted in Figure 1. First, the name and the formal parameters of the procedure are specified. The *type*, the *relation* and the *size relation* information express properties on the formal parameters which are intended to be satisfied after any successful execution. The *size measure* (see [10]) $\|\cdot\|_\lambda$ associates to each term t a natural number $\|t\|_\lambda$ by: $\|[t_1|t_2]\|_\lambda = 1 + \|t_2\|_\lambda$ and $\|t\|_\lambda = 0$ if t is not of the form $[t_1|t_2]$. The *size relation* is a set of linear (in)equations on the formal parameters expressing a relation between the corresponding sizes. In the example, after any successful execution, the size of L is required to be equal to the size of L_s plus 1.

Procedure: $select(X, L, L_s)$
Types: X : Term
L, L_s : Lists
Relation: X is an element of L and L_s is the list L without an occurrence of X
Size measure: $\|\cdot\|_\lambda$
Size relation: $\{L = L_s + 1\}$
Application conditions: $\mathbf{in}(any, gr, any) :: \mathbf{out}(gr, gr, gr)$ $\{0 \le sol \le L\}$ L
$\mathbf{in}(gr, any, gr) :: \mathbf{out}(gr, gr, gr)$ $\{0 \le sol \le L_s + 1\}$ L_s

Fig. 1. Specification for *select*/3

The application conditions consist of three components namely, *directionality, multiplicity* and *size expression*. Each *directionality* specifies the allowed modes of the parameters before the execution (**in** part) and the corresponding modes after a successful execution (**out** part).
For any set of program variables V, we denote by $\mathbf{Exp}_{\mathrm{V}}$ the set of all linear expressions with integer coefficients on variables in V. A *multiplicity* is a set E of (in)equations over $\mathbf{Exp}_{\{sol, X_1, \ldots, X_n\}} \cup \{\infty\}$ where $X_1, \ldots, X_n$ are the formal parameters of the procedure and *sol* is a new variable denoting the cardinality of the answer substitution sequence. For any call to the procedure satisfying the **in** part of the corresponding directionality and producing the sequence S of answer substitutions, the following property is expected to hold: the assignment $\{sol/|S|, X_1/\|t_1\|, \ldots, X_n/\|t_n\|\}$, where $t_1, \ldots, t_n$ are the input values corresponding to the formal parameters, satisfies the set of (in)equations E. The *size expression* is an expression from $\mathbf{Exp}_{\{X_1, \ldots, X_n\}}$[4] associating to each possible call, respecting the corresponding **in** part, a weight obtained by replacing the formal parameters with the size of the corresponding actual values. Such an expression is assumed to be a natural number for any instantiation of the parameters. In order to prove termination of a recursive call we need to prove

[4] It corresponds, in a sense, to the level mapping used by De Schreye *et al.* in [10].

that its weight is smaller than the weight for the initial one[5]. In the example, the weight for the calls of *select*/3 respecting the first (resp. the last) **in** part is given by the size of the actual value corresponding to L (resp. L_s).

2.2 Logic Description

Figure 2 provides a correct logic description for *select*/3, noted $LD(select/3)$.

$$\begin{aligned} select(X, L, L_s) \Longleftrightarrow\ & L = [H|T]\ \wedge list(T) \wedge \\ & ((H = X \wedge L_s = T) \vee\ (L_s = [H_s|T_s] \wedge select(X, T, T_s))). \end{aligned}$$

Fig. 2. $LD(select/3)$

The correctness of a logic description $LD(p/n)$ can be expressed as follows: for any ground n-tuple t, (1) $p(t)$ is a logical consequence of $LD(p/n)$ iff t belongs to the relation and respects the types; (2) $\neg p(t)$ is a logical consequence of $LD(p/n)$ iff either t does not belong to the relation or it does not respect the types.

2.3 Prolog Code Derivation

The next step consists in deriving a Prolog program from the logic description. First, the logic description $LD(p/n)$ is syntactically translated into a Prolog program $LP(p/n)$, in normalized form, whose completion [3] is the logic description again[6]. Figure 3, 4, and 5 depict three different translations of the logic description in Figure 2.

Among the derived procedures, we need to select the ones which are correct with respect to the initial specification. For instance, some of them may not terminate when executed following the left-to-right Prolog search rule.

Among the correct ones, some of them may be more efficient than others. Moreover, they can be simplified in order to gain efficiency. Atoms that surely succeed without instantiating their arguments can be removed; this is typically the case for type checking literals. For instance, procedure $LP_1(select/3)$ is not correct for the second directionality since the literal $list(T)$ could be called with its argument being a variable (e.g., when L is a variable) and then it would produce infinitely many solutions; procedure $LP_2(select/3)$ is correct with respect to the specification in Figure 1; and procedure $LP_3(select/3)$ is obtained from procedure $LP_2(select/3)$ by dropping the last atom $list(T)$. This transformation is made possible by the type information computed during the analysis.

Thus, the analyzer can play three different roles: finding correct permutations, selecting the most efficient one, and, finally, simplifying procedures to make them more efficient.

[5] For more details about these concepts the reader is referred to [1, 10].

[6] Actually, the translation process may produce more than one possible procedure. Different permutations of the clauses and of the literals in the bodies can be returned.

$select(X, L, L_s) :- L = [H|T],\ list(T),\ H = X,\ L_s = T.$
$select(X, L, L_s) :- L = [H|T],\ list(T),\ L_s = [H|T_s],\ select(X, T, T_s).$

Fig. 3. $LP_1(select/3)$

$select(X, L, L_s) :- L = [H|T],\ H = X,\ L_s = T,\ list(T).$
$select(X, L, L_s) :- L = [H|T],\ L_s = [H|T_s],\ select(X, T, T_s),\ list(T).$

Fig. 4. $LP_2(select/3)$

$select(X, L, L_s) :- L = [H|T],\ H = X,\ L_s = T,\ list(T).$
$select(X, L, L_s) :- L = [H|T],\ L_s = [H|T_s],\ select(X, T, T_s).$

Fig. 5. $LP_3(select/3)$

2.4 Correctness Verification

The last step, in Deville's methodology, is the verification of total correctness of the derived program with respect to the specification. In order to be correct, any procedure in the program must satisfy the following criteria: during any execution (based on the SLDNF-resolution) called with arguments respecting at least one **in** part of the corresponding specification and producing the sequence S of answer substitutions,

1. the computation rule is safe, i.e., when selected, the negative literals are ground;
2. any subcall is called with arguments respecting at least one **in** part of its specification;
3. the arguments of the procedure after the execution respect the types, the size relation and the **out** part of each directionality whose **in** part is satisfied by the initial call;
4. S respects the multiplicity of each directionality whose **in** part is satisfied by the initial call;
5. completeness: every computed answer substitution in the SLDNF-tree belongs to S , i.e., it must eventually be caught by the analyzer according to the Prolog search rule;
6. termination: if S is finite then the execution terminates.

When S is finite, points 5 and 6 are satisfied if the execution of each clause terminates. Termination of a clause is achieved when each literal in its body which is not a recursive call terminates, and the weights for the recursive calls are smaller than the weight for the initial call. When S can be infinite, then only completeness has to be verified. A sufficient criterion for completeness is the following: at most one literal in a clause of the procedure produces an infinite number of answer substitutions and either this literal is in the last clause or the following clauses are finitely failed (i.e., they terminate without producing any result); moreover, if a clause contains a literal that produces an infinite number of answer substitutions, then none of the preceding literals in this clause produces more than one solution.

The main advantage provided by the analyzer that we present in this paper is the full automatization of the last step of Deville's methodology described so far, i.e., the verification of total correctness of synthesized programs with respect to the specification. Both the procedures $LP_2(select/3)$ and $LP_3(select/3)$ above can be proved to be correct wrt the specification depicted in Figure 1. In section 4 we will illustrate the behaviour of the analyzer by considering $LP_3(select/3)$.

3 Basic Notions

In this section we briefly describe the abstract domains used by the analyzer. Based on the notion of *abstract substitution*, the concept of *abstract sequence* is introduced to represent a sequence of answer substitutions. Finally, we formalize the program specification through the notion of *behaviour*.
In the following we call I_p the set of indices $\{1, \ldots, p\}$, denoting terms $t_1, \ldots, t_p$. Next notions are parameterized on p, and can be extended to the set of all I_p in the same way as in [16].

Definition 1 (Abstract substitution). An *abstract substitution* β is an element of the generic abstract domain $\mathtt{Pat}(\Re)$ described in [5, 6], i.e., a tuple $\langle sv, frm, \alpha \rangle$ where

1. the same-value component sv assigns an index from I_p to each program variable;
2. the pattern component frm associates with some of the indices in I_p a pattern $f(i_1, \ldots, i_q)$, where f is a q-ary function symbol and $\{i_1, \ldots, i_q\} \subseteq I_p$;
3. α is an element of a domain $\Re$ that gives information on term tuples $\langle t_1, \ldots, t_p \rangle$ about mode, sharing, types, or whatever else.

Example 1. Let $\Re$ be the abstract domain *Mode-Type*[7] described in [14], and consider the (concrete) substitution

$$\{X_1 \mapsto Y * a, X_2 \mapsto a, X_3 \mapsto [\,]\}.$$

This substitution is represented, for instance, by the following abstract substitution:

$$\begin{array}{llll} sv\!: X_1 \mapsto 1 & frm\!: 1 \mapsto 4 * 2 & \alpha : 1/ngv \\ \quad X_2 \mapsto 2 & \quad 2 \mapsto a & \quad 2/gr \\ \quad X_3 \mapsto 3 & \quad 3 \mapsto [\,] & \quad 3/grlist \\ & \quad 4 \mapsto ? & \quad 4/var \end{array}$$

[7] It consists of the elements *gr* denoting a ground term, *any* denoting any term, *var* denoting a variable, *nv* denoting a term which is not a variable, *ngv* denoting a term which is neither ground nor a variable, *grlist* denoting a ground list, *anylist* denoting either a variable or any list.

At each execution point, the analyzer computes (so-called) abstract sequences [2, 15] giving information about variables in the form of an abstract substitution, and also information about the number of solutions in terms of the input argument sizes. Thus, an abstract sequence can be seen as an extension of the abstract substitution notion, where an additional same-value component maintains the parameter assignments at clause entry, and a set of linear (in)equations represents information relative to the size relations among terms and the number of solutions.

Definition 2 (Abstract sequence). An *abstract sequence* B is a tuple of the form $\langle sv_{in}, sv, frm, \alpha, E\rangle$ where

1. $\langle sv_{in}, frm, \alpha\rangle$ and $\langle sv, frm, \alpha\rangle$ are abstract substitutions;
2. the domain of the same-value function sv_{in} is contained in the domain of sv;
3. the *size component* E is a (possibly empty) set of linear (in)equations over $\mathbf{Exp}_{\{sol, sz(1),\ldots,sz(p)\}}$.

Given a size measure $\|\cdot\|$, an abstract sequence $B = \langle sv_{in}, sv, frm, \alpha, E\rangle$ represents the set of all pairs (σ_{in}, S) where $\sigma_{in} \in \gamma(\langle sv_{in}, frm, \alpha\rangle)$ is a concrete substitution and S is a possibly infinite sequence of substitutions such that for all $\sigma \in S$, σ is an instance of σ_{in}, $\sigma \in \gamma(\langle sv, frm, \alpha\rangle)$, and S respects the size and the multiplicity relations expressed in E. The latter condition means that for every $\langle t_1, \ldots, t_p\rangle \in \gamma(\alpha)$ defining a substitution[8] $\sigma \in S$: $(sz(1)/\|t_1\|, \ldots, sz(p)/\|t_p\|, sol/|S|)$ satisfies the set of (in)equations E.

A behaviour for a procedure is a formalization of its specification (excluding the relation part which is formalized by the logic description).

Definition 3 (behaviour). A *behaviour* for a procedure p/n is a triplet of the form $(p, [X_1, \ldots, X_n], Prepost)$ where

1. $X_1, \ldots, X_n$ are distinct variables representing the formal parameters of the procedure p/n;
2. *Prepost* is a set of pairs $\langle B, Se\rangle$ where $B = \langle sv_{in}, sv_{out}, frm, \alpha, E\rangle$ is an abstract sequence and Se is a size expression from $\mathbf{Exp}_{\{X_1,\ldots,X_n\}}$.

Example 2. A behaviour for $select/3$ formalizing the specification depicted in Figure 1 has the form

$$(select, [X_1, X_2, X_3], Prepost = \{\langle B^1, X_2\rangle, \langle B^2, X_3\rangle\})$$

where B^2 (corresponding to the second application condition) is equal to

[8] To be more precise, let $dom(sv) = \{X_1, \ldots, X_n\}$, $\langle t_1, \ldots, t_p\rangle$ is the (unique) tuple of terms such that $\sigma = \{X_1/t_{sv(X_1)}, \ldots, X_n/t_{sv(X_n)}\}$ and $\langle t_1, \ldots, t_p\rangle \in Cc(frm)$, see [16].

$$\begin{array}{llllll}
sv_{in}\colon X_1 \mapsto 1 & sv_{out}\colon X_1 \mapsto 4 & frm\colon 1 \mapsto ? & \alpha : 1/gr & E\colon 0 \le sol \le sz(3)+1 \\
\quad\;\; X_2 \mapsto 2 & \quad\;\; X_2 \mapsto 5 & \quad\;\; 2 \mapsto ? & \quad\; 2/any & \quad\; sz(5) = sz(6)+1 \\
\quad\;\; X_3 \mapsto 3 & \quad\;\; X_3 \mapsto 6 & \quad\;\; 3 \mapsto ? & \quad\; 3/gr & \\
 & & \quad\;\; 4 \mapsto ? & \quad\; 4/gr & \\
 & & \quad\;\; 5 \mapsto ? & \quad\; 5/grlist & \\
 & & \quad\;\; 6 \mapsto ? & \quad\; 6/grlist &
\end{array}$$

and the abstract sequence B^1 (corresponding to the first application condition) can be defined in a similar way.

4 The Analyzer

In this section we first present a clause analyzer for verifying correctness of a clause. This is a refinement and an extension of the analyzer proposed in [7]. Then, we describe a procedure analyzer, based on the clause analyzer, for verifying correctness of a whole procedure.

4.1 The Clause Analyzer

The clause analyzer takes as inputs (1) a clause $cl = p(X_1,\ldots,X_n)\colon -L_1,\ldots,L_f$ where $n, f \geq 0$ and $X_1,\ldots,X_n$ are distinct variables, (2) a behaviour for each subgoal in cl (except $=/2$ but including p/n), (3) an element from the *Prepost* component in the behaviour for p/n of the form $\langle\langle sv_{in}, sv_{out}, frm, \alpha, E\rangle, Se\rangle$, (4) an abstract substitution β on the variables $X_1,\ldots,X_n$ in the head of the clause respecting the precondition $\langle sv_{in}, frm, \alpha\rangle$, i.e., β is smaller than or equal to the abstract substitution $\langle sv_{in}, frm, \alpha\rangle$, according to the partial order defined on $\mathtt{Pat}(\Re)$. It checks the following:

1. for any subcall q/m in the body of cl, let $(q, [X_1,\ldots,X_m], Prepost_q)$ be the behaviour for q/m. The procedure q/m is called with arguments respecting at least one precondition $\langle sv'_{in}, frm', \alpha'\rangle$ in the $Prepost_q$ set;
2. if sol/∞ satisfies the restriction of E to sol, then at most one subcall in the body of cl can produce an infinite number of solutions and none of the preceding literals in this clause produces more than one solution. Otherwise, the execution of cl terminates.

If one of these properties can not be inferred then the clause analyzer fails. Otherwise, it returns an abstract sequence B_{cl} expressing information about types, modes, size relations, number of solutions and termination for the execution of the clause cl.

In order to compute its results, the clause analyzer computes a set of abstract sequences, $B_0, B_1, \ldots, B_f$, on the variables in cl and an abstract sequence B_{out} expressing the same properties as B_f but restricted to the variables $X_1,\ldots,X_n$ in the head of cl. The following main operations are required to analyze a clause.

Initial operation: it extends the abstract substitution β to an abstract sequence B_0 on all the variables in cl.

Derivation operation: it computes the abstract sequence B_i from B_{i-1} and the literal L_i $(1 \leq i \leq f)$. In fact, there are two kinds of derivation operations: the *deriv-unif* (for the $=/2$ literals) and the *deriv-normal* (for procedure calls).

Reduction operation: it computes B_{out} from B_f by restricting it to the variables $X_1, \ldots, X_n$.

Let us illustrate them with an example. Consider the second clause of the procedure $LP_3(select/3)$ and the behaviour for $select/3$ described above. Let $\beta = \langle sv, frm, \alpha, E \rangle$ be an abstract substitution of the form

$$\begin{array}{llll} sv: & X_1 \mapsto 1 & frm: 1 \mapsto ? & \alpha : 1/gr \\ & X_2 \mapsto 2 & \quad 2 \mapsto ? & \quad 2/any \\ & X_3 \mapsto 3 & \quad 3 \mapsto ? & \quad 3/gr \end{array}$$

The clause analyzer computes $B_0, B_1, \ldots, B_3, B_{out}$ as follows:

$$\begin{array}{l} \beta \\ select(X_1, X_2, X_3) \leftarrow B_0 \; X_2 = [X_4|X_5], \\ \qquad\qquad\qquad\qquad\quad B_1 \; X_3 = [X_4|X_6], \\ \qquad\qquad\qquad\qquad\quad B_2 \; select(X_1, X_5, X_6) \; B_3. \\ B_{out} \end{array}$$

Initial Operation. First, the abstract substitution β is extended to an abstract sequence B_0 on all the variables in the clause stating that the new variables are not instantiated and used nowhere else. In the example,

$$\begin{array}{llllll} sv^0_{in}: X_1 \mapsto 1 & sv^0: X_1 \mapsto 1 & frm^0: 1 \mapsto ? & \alpha^0 : 1/gr & E^0 : sol = 1 \\ \quad\;\; X_2 \mapsto 2 & \quad\;\; X_2 \mapsto 2 & \quad 2 \mapsto ? & \quad 2/any & \\ \quad\;\; X_3 \mapsto 3 & \quad\;\; X_3 \mapsto 3 & \quad 3 \mapsto ? & \quad 3/gr & \\ & \quad\;\; X_4 \mapsto 4 & \quad 4 \mapsto ? & \quad 4/var & \\ & \quad\;\; X_5 \mapsto 5 & \quad 5 \mapsto ? & \quad 5/var & \\ & \quad\;\; X_6 \mapsto 6 & \quad 6 \mapsto ? & \quad 6/var & \end{array}$$

Derivation operation. We compute B_1 from B_0 and the literal $X_2 = [X_4|X_5]$. In this case, the *deriv-unif* operation is used. A new term (represented by the index 7) is introduced, representing the result of the unification of X_2 with $[X_4|X_5]$. The sv_{in} component maintain the link with the term denoted by 2 while the actual binding of X_2 after the unification is kept by sv. Since it is not sure that the unification succeeds, the information that a failure may occur is expressed in the multiplicity component. Information on size relations after the unification is expressed by the equation $sz(7) = sz(5) + 1$. Thus, B_1 is

$$
\begin{array}{llllll}
sv^1_{in}: X_1 \mapsto 1 & sv^1: X_1 \mapsto 1 & frm^1: 1 \mapsto ? & \alpha^1: 1/gr & E^1: 0 \leq sol \leq 1 \\
\quad X_2 \mapsto 2 & \quad X_2 \mapsto 7 & \quad 2 \mapsto ? & \quad 2/any & \quad sz(7) = sz(5)+1 \\
\quad X_3 \mapsto 3 & \quad X_3 \mapsto 3 & \quad 3 \mapsto ? & \quad 3/gr & \\
 & \quad X_4 \mapsto 4 & \quad 4 \mapsto ? & \quad 4/any & \\
 & \quad X_5 \mapsto 5 & \quad 5 \mapsto ? & \quad 5/any & \\
 & \quad X_6 \mapsto 6 & \quad 6 \mapsto ? & \quad 6/var & \\
 & & \quad 7 \mapsto [4|5] & \quad 7/nv &
\end{array}
$$

Another derivation operation applies to derive B_2 from B_1 and the literal $X_3 = [X_4|X_6]$. Also in this case, the *deriv-unif* operation is used. Observe that the groundness of the term indexed by 3 propagates to the terms that correspond (through sv) to X_4 and X_6. Moreover, in this case no new term is created, as the term indexed by 3 is ground before the unification step. The abstract sequence B_2 is

$$
\begin{array}{llllll}
sv^2_{in}: X_1 \mapsto 1 & sv^2: X_1 \mapsto 1 & frm^2: 1 \mapsto ? & \alpha^2: 1/gr & E^2: 0 \leq sol \leq 1 \\
\quad X_2 \mapsto 2 & \quad X_2 \mapsto 7 & \quad 2 \mapsto ? & \quad 2/any & \quad sz(7) = sz(5)+1 \\
\quad X_3 \mapsto 3 & \quad X_3 \mapsto 3 & \quad 3 \mapsto [4|6] & \quad 3/gr & \quad sz(3) = sz(6)+1 \\
 & \quad X_4 \mapsto 4 & \quad 4 \mapsto ? & \quad 4/gr & \\
 & \quad X_5 \mapsto 5 & \quad 5 \mapsto ? & \quad 5/any & \\
 & \quad X_6 \mapsto 6 & \quad 6 \mapsto ? & \quad 6/gr & \\
 & & \quad 7 \mapsto [4|5] & \quad 7/nv &
\end{array}
$$

The abstract sequence B_3 is obtained through *deriv-normal* by combining B_2 with the behaviour of *select*/3. In particular, we need to find a pair $\langle B, Se\rangle$ in the *Prepost* component of the behaviour that matches with B_2 regarding *directionality* and *termination*. In our example, this is true when considering the prepost component $\langle B^2, X_3\rangle$ defined at the end of Section 3.
Let $B^2 = \langle sv_{in}, sv_{out}, frm, \alpha, E\rangle$ and let $\tau = \{X_1 \mapsto X_1, X_2 \mapsto X_5, X_3 \mapsto X_6\}$ be the renaming function that is used to restrict B_2 to the clause $select(X_1, X_5, X_6)$.

First, we need to verify directionality: that the abstract substitution obtained by applying τ to $\langle sv^2, frm^2, \alpha^2\rangle$ is at least as precise (i.e., smaller or equal in the ordering on $\mathtt{Pat}(\Re)$) as $\langle sv_{in}, frm, \alpha\rangle$. In particular, if we look at modes, it is easy to verify that

$$
\begin{array}{l}
\mathbf{in}(\alpha^2(sv^2(\tau(X_1))), \alpha^2(sv^2(\tau(X_2))), \alpha^2(sv^2(\tau(X_3)))) = \\
\mathbf{in}(\alpha^2(sv^2(X_1)), \alpha^2(sv^2(X_5)), \alpha^2(sv^2(X_6))) = \\
\mathbf{in}(\alpha^2(1), \alpha^2(5), \alpha^2(6)) = \\
\mathbf{in}(gr, any, gr) = \\
\mathbf{in}(\alpha(sv(X_1)), \alpha(sv(X_2)), \alpha(sv(X_3)))
\end{array}
$$

Second, since sol/∞ is not a solution in E^2, we have to prove termination: according to the size expression $Se = X_3$ in the *Prepost* component selected, we need to verify that the size of the third term of the new activation call is strictly smaller than the size of the third parameter before the execution. Formally, we need to verify that

$$sz(sv^2(\tau(X_3))) < sz(sv_{in}^2(X_3))$$

Indeed,

$$\begin{array}{rll} sz(sv_{in}^2(X_3)) & = sz(3) & \text{by definition of } sv_{in}^2 \\ & = sz(6)+1 & \text{by definition of } E^2 \\ & > sz(6) & \\ & = sz(sv^2(\tau(X_3))) & \text{by definition of } sv^2 \text{ and } \tau. \end{array}$$

Thus, both directionality and termination of the application condition are satisfied. Therefore, we can apply the out-conditions of B^2. In particular, we derive that the terms indexed by 5 and 6 are bound to ground lists, and we get the multiplicity equations $0 \leq sol \leq sz(6)+1$ and $sz(5) = sz(6)+1$. We obtain the following abstract sequence B_3:

$$\begin{array}{llllllllll} sv_{in}^3: & X_1 \mapsto 1 & sv^3: & X_1 \mapsto 1 & frm^3: & 1 \mapsto ? & \alpha^3: & 1/gr & E^3: & 0 \leq sol \leq sz(6)+1 \\ & X_2 \mapsto 2 & & X_2 \mapsto 7 & & 2 \mapsto ? & & 2/any & & sz(7) = sz(5)+1 \\ & X_3 \mapsto 3 & & X_3 \mapsto 3 & & 3 \mapsto [4|6] & & 3/grlist & & sz(3) = sz(6)+1 \\ & & & X_4 \mapsto 4 & & 4 \mapsto ? & & 4/gr & & sz(5) = sz(6)+1 \\ & & & X_5 \mapsto 5 & & 5 \mapsto ? & & 5/grlist & & \\ & & & X_6 \mapsto 6 & & 6 \mapsto ? & & 6/grlist & & \\ & & & & & 7 \mapsto [4|5] & & 7/grlist & & \end{array}$$

Reduction operation. The abstract sequence B_{out} is computed by restricting B_3 to the variables X_1, X_2, X_3. It results in

$$\begin{array}{llllllllll} sv_{in}^o: & X_1 \mapsto 1 & sv^o: & X_1 \mapsto 1 & frm^o: & 1 \mapsto ? & \alpha^o: & 1/gr & E^o: & 0 \leq sol \leq sz(6)+1 \\ & X_2 \mapsto 2 & & X_2 \mapsto 7 & & 2 \mapsto ? & & 2/any & & sz(7) = sz(5)+1 \\ & X_3 \mapsto 3 & & X_3 \mapsto 3 & & 3 \mapsto [4|6] & & 3/grlist & & sz(3) = sz(6)+1 \\ & & & & & 4 \mapsto ? & & 4/gr & & sz(5) = sz(6)+1 \\ & & & & & 5 \mapsto ? & & 5/grlist & & \\ & & & & & 6 \mapsto ? & & 6/grlist & & \\ & & & & & 7 \mapsto [4|5] & & 7/grlist & & \end{array}$$

4.2 The Procedure Analyzer

The clause analyzer is used to define a procedure analyzer for verifying total correctness of a full procedure. It takes as inputs (1) a Prolog procedure P defining a predicate p/n and consisting of a finite sequence of clauses $cl_1, \ldots, cl_m$, (2) a behaviour for each subprocedure in P. For any abstract substitution respecting at least one precondition in the behaviour for p/n,

1. it checks whether for each clause cl_i of P, the clause analyzer does not fail returning the abstract sequence B_{cl_i};

2. it computes the abstract operation $\texttt{CONC}(B_{cl_1}, \ldots, B_{cl_n})$ which approximates the sequence of answer substitutions for the whole procedure (taking into account mutual exclusivity between clauses and incompleteness due to an infinite computation) and verifies the postconditions, i.e., types, modes, size relations, multiplicity. This operation can be seen as an extension of the corresponding operation in [15];
3. if the sequence of answer substitutions can be infinite, then at most one clause of P can produce an infinite number of solutions and if this is not the last clause, then the executions of all the following clauses in P are finitely failed.

If one of these properties can not be inferred then the procedure analyzer fails. This simply means that the correctness of the procedure P cannot be proven by the analyzer.

5 Abstract Operations

In this section[9], we formally describe the main operations used by the clause analyzer, namely `INIT`, `REDUCE`, `DERIV_UNIF`, `DERIV_NORMAL`. For each of this operations we provide a specification and an implementation. The specification is given in terms of the concretization function Cc whereas the implementation applies abstract operations on the $\Re$-domain (corresponding to the α component in the abstract substitutions) defined in [5]. We also assume the existence of operations on the multiplicity component E of the abstract substitutions. We call them $\Im$-operations. If we restrict to sets of linear equations, then we can always reduce E to the canonical form which is supplied by the so-called *row-echelon form*. In this case, the $\Im$-operations are similar to the abstract operations defined by De Schreye, Verschaetse and Bruynooghe in [10]. Notice also that implementations proposed in this section are only tentative since the actual coding of the analyzer is still underway. Future developments may reveal that better implementations are possible.

5.1 Constrained Mapping

The notion of *constrained mapping* (see [17]) is used in the implementation of the `DERIV_UNIF` and `DERIV_NORMAL` operations. Intuitively, let I and J be two sets of indices denoting arbitrary terms and α_I and α_J be two abstract objects parametrized by the indices in I and J, respectively. Let also $tr : I \to J$ be a total function expressing that, for every $i \in I$, the element denoted by i is equal to the element denoted by $tr(i)$. A constrained mapping is a pair $(tr^<, tr^>)$ where $tr^<$ applies to abstract objects parametrized by the indices in J and expresses them in terms of the indices in I, e.g., $tr^<(\alpha_J) = \alpha_I$, while $tr^>$ applies to abstract objects parametrized by the indices in I and expresses them in terms of the indices in J, e.g., $tr^>(\alpha_I) = \alpha_J$.

[9] This section is quite technical, and can be skipped at a first reading.

Let f_1 and f_2 be two functions (tuples are functions) with domains D_1 and D_2, respectively, such that $D_1 \cap D_2 = \emptyset$. We denote $f_1 \square f_2$ the function f with domain $D_1 \cup D_2$ such that for all $d \in D_1 : f(d) = f_1(d)$ and for all $d \in D_2 : f(d) = f_2(d)$.

In the following, since sv, frm, ... are functions over finite domains, they are referred as sets of assignments.

5.2 $\Re$-Operations and $\Im$-Operations

Let us recall from [5, 10] the specification of some abstract operations which are used in the implementation of the analyzer, namely, $\Re$_INTR, $\Re$_PROJ (on the α component, [5]), $\Im$_PROJ and $\Im$_EXTG (on the size component E), and AI_VAR, AI_FUNC, RESTRG and EXTG (on abstract substitutions, [5]),

$\Re$_INTR$(\alpha, p, k) = \alpha'$. Let α be an element of the domain $\Re$ that gives information on term tuples $\langle t_1, \ldots, t_p \rangle$ and $k \geq 0$. This operation introduces k variables in locations $p+1, \ldots, p+k$.

$$\left.\begin{array}{l} \langle t_1, \ldots, t_p \rangle \in Cc(\alpha) \\ Y_1, \ldots, Y_k \text{ are new distinct variables} \end{array}\right\} \Rightarrow \langle t_1, \ldots, t_p, Y_1, \ldots, Y_k \rangle \in Cc(\alpha').$$

$\Re$_PROJ$(\alpha, \{i_1, \ldots, i_n\}) = \alpha'$. Let α be an element of the domain $\Re$ representing term tuples $\langle t_1, \ldots, t_p \rangle$ such that $i_1, \ldots, i_n \in \{1, \ldots, p\}$ and let $\langle j_1, \ldots, j_q \rangle$ be obtained from $\langle 1, \ldots, p \rangle$ by removing the indices $i_1, \ldots, i_n$.

$$\langle t_1, \ldots, t_p \rangle \in Cc(\alpha) \Rightarrow \langle t_{j_1}, \ldots, t_{j_q} \rangle \in Cc(\alpha').$$

$\Im$_PROJ$(E, \{i_1, \ldots, i_n\}) = E'$. Let E be a set of (in)equations over the set $\mathbf{Exp}_{\{sz(1), \ldots, sz(p)\}, sol}$ and let $\langle j_1, \ldots, j_q \rangle$ be obtained from $\langle 1, \ldots, p \rangle$ by removing the indices $i_1, \ldots, i_n$. Let also $x_1, \ldots, x_p, x_{p+1} \in \mathbb{N}$.

$$\left.\begin{array}{l} \langle sz(1)/x_1, \ldots, sz(p)/x_p, sol/x_{p+1} \rangle \\ \text{is a solution of } E \end{array}\right\} \Rightarrow \begin{array}{l} \langle sz(j_1)/x_{j_1}, \ldots, sz(j_q)/x_{j_q}, sol/x_{p+1} \rangle \\ \text{is a solution of } E'. \end{array}$$

The specification for $\Im$_PROJ$(E, sol) = E'$ is similar.

$\Im$_EXTG$(p(X_{i_1}, \ldots, X_{i_n}), E_1, E_2) = E'$. This operation is used to extend the size relation and the number of solutions information for a clause before a literal $p(X_{i_1}, \ldots, X_{i_n})$ (i.e., E_1) with the size relation and the number of solutions information obtained from the behaviour of p/n. Let E_1 and E_2 be two sets of (in)equations over the sets $\mathbf{Exp}_{\{(sz(j))_{j \in I_1}, sol\}}$ and $\mathbf{Exp}_{\{(sz(h))_{h \in I_2}, sol\}}$, respectively. Let also $\{X_1, \ldots, X_m\}$ be the set of all variables occurring in the clause. E' is a set of (in)equations over $\mathbf{Exp}_{\{(sz(k))_{k \in I'}, sol\}}$ such that

$$\left.\begin{array}{l} (\|t_j\|)_{j \in I_1} \square \{sol \mapsto sol_1\} \text{ is a solution of } E_1 \\ (\|s_h\|)_{h \in I_2} \square \{sol \mapsto sol_2\} \text{ is a solution of } E_2 \\ \theta_1 = \{X_1/t_{j_1}, \ldots, X_m/t_{j_m}\},\ j_1, \ldots, j_m \in I_1 \\ \theta_2 = \{X_1/s_{h_1}, \ldots, X_m/s_{h_n}\},\ h_1, \ldots, h_n \in I_2 \\ \exists \sigma :\ \forall j, 1 \leq j \leq n,\ X_j \theta_2 = X_{i_j} \theta_1 \sigma \end{array}\right\} \Rightarrow \begin{array}{l} (\|u_k\|)_{k \in I'} \square \{sol \mapsto sol_1 \times sol_2\} \\ \text{is a solution of } E' \\ \theta_1 \sigma = \{X_1/u_{k_1}, \ldots, X_m/u_{k_m}\}, \\ \{k_1, \ldots, k_m\} \subseteq I' \end{array}$$

Conditions on domains and codomains of substitutions which prevent renaming variable clashes between θ_1 and θ_2 are also required for allowing more precise implementations of the operation.

`AI_VAR`$(\beta, X = t) = \beta'$ and `AI_FUNC`$(\beta, X = t) = \beta'$. The `AI_VAR` operation unifies X and t in the abstract substitution β when t is a variable. The `AI_FUNC` operation is analogous in the case that t is a non-variable term.

$$\left.\begin{array}{l} \theta \in Cc(\beta) \\ \sigma \in mgu(X\theta, t\theta) \end{array}\right\} \Rightarrow \theta\sigma \in Cc(\beta').$$

`RESTRG`$(p(X_{i_1}, \ldots, X_{i_n}), \beta) = \beta'$. The `RESTRG` operation expresses an abstract substitution β on the parameters $X_{i_1}, \ldots, X_{i_n}$ of the call $p(X_{i_1}, \ldots, X_{i_n})$ in terms of the formal parameters $X_1, \ldots, X_n$.

$$\theta \in Cc(\beta) \Rightarrow \{X_1/X_{i_1}\theta, \ldots, X_n/X_{i_n}\theta\} \in Cc(\beta').$$

`EXTG`$(p(X_{i_1}, \ldots, X_{i_n}), \beta_1, \beta_2) = \beta'$. This operation is used to instantiate abstractly a clause substitution (i.e., β_1) with the result of the execution of a procedure call (i.e., β_2). Let β_1 and β_2 be two abstract substitutions such that $\{X_{i_1}, \ldots, x_{i_n}\} \subseteq \{X_1, \ldots, X_m\} = dom(\beta_1)$ and $\{X_1, \ldots, X_n\} = dom(\beta_2)$.

$$\left.\begin{array}{l} \theta_1 \in Cc(\beta_1) \\ \theta_2 \in Cc(\beta_2) \\ \exists\sigma : \forall j, 1 \leq j \leq n,\ X_j\theta_2 = X_{i_j}\theta_1\sigma \end{array}\right\} \Rightarrow \theta_1\sigma \in Cc(\beta').$$

Again, conditions on domains and codomains of substitutions are required.

5.3 The Initial Operation: `INIT`$(\beta, m) = B'$

This operation extends the abstract substitution β on the variables $X_1, \ldots, X_n$ $(0 \leq n)$ in the head of a clause cl to an abstract sequence B' on all the variables $X_1, \ldots, X_m$ $(n \leq m)$ in cl.

Specification. Let $\langle Y_1, \ldots, Y_{m-n}\rangle$ denote new free variables. `INIT`(β, m) produces an abstract sequence B' such that

$$\left.\begin{array}{l} \sigma \in Cc(\beta) \\ \sigma' = \sigma \circ \{X_{n+1}/Y_1, \ldots, X_m/Y_{m-n}\} \end{array}\right\} \Rightarrow (\sigma, \langle\sigma'\rangle) \in Cc(B').$$

Implementation. Let $\beta = \langle sv, frm, \alpha\rangle$ with $dom(frm) = I_p$. The implementation of the `INIT` operation applies the $\Re$-`INTR` operation in order to extend the α component of the abstract substitution. $\mathtt{INIT}(\beta, m) = B'$ where $B' = \langle sv'_{in}, sv', frm', \alpha', E'\rangle$ is defined as follows:

- $sv'_{in} = sv$;
- $sv' = sv \cup \{X_{n+1} \mapsto p+1, \ldots, X_m \mapsto p+m-n\}$;
- $frm' = frm \cup \{p+1 \mapsto ?, \ldots, p+m-n \mapsto ?\}$;
- $\alpha' = \Re\text{-}\mathtt{INTR}(\alpha, p, m-n)$;
- $E' = \{sol = 1\}$.

5.4 The Reduction Operation: $\mathtt{REDUCE}(B, n) = B'$

It computes B' from the abstract sequence B on the variables $X_1, \ldots, X_m$ in the clause cl by restricting it to the variables $X_1, \ldots, X_n$ $(n \leq m)$ in the head.

Specification. Let S and S' be (possibly infinite) sequences of substitutions.

$$\left.\begin{array}{l} (\sigma, S) \in Cc(B) \\ S = \langle \sigma_1, \sigma_2, \ldots\rangle \\ S' = \langle \sigma_{1|\{X_1,\ldots,X_n\}}, \sigma_{2|\{X_1,\ldots,X_n\}}, \ldots\rangle \end{array}\right\} \Rightarrow (\sigma, S') \in Cc(B').$$

Implementation. Let $B = \langle sv_{in}, sv, frm, \alpha, E\rangle$, with $dom(frm) = I_p$. The implementation of the `REDUCE` operation uses the $\Re$-`PROJ` operation on the α component of the abstract sequence B and the $\Im$-`PROJ` operation on the equation set E. $\mathtt{REDUCE}(B, n) = B'$ where $B' = \langle sv'_{in}, sv', frm', \alpha', E'\rangle$ is defined as follows:

- $sv'_{in} = sv_{in}$;
- $sv' = \{(X_i \mapsto j) \in sv \mid 0 \leq i \leq n\}$;
- $frm' = \{(i \mapsto frm_i) \in frm \mid i \in (codom(sv'_{in}) \cup codom(sv'))\} \cup \{(j \mapsto frm_j) \in frm \mid \exists i \in (codom(sv'_{in}) \cup codom(sv')) : j$ is reachable from i by $frm\}$;
- $\alpha' = \Re\text{-}\mathtt{PROJ}(\alpha, I_p \setminus dom(frm'))$;
- $E' = \Im\text{-}\mathtt{PROJ}(E, I_p \setminus dom(frm'))$.

5.5 The Derivation Operation: $\mathtt{DERIV\_UNIF}(B, X = t) = B'$

This operation computes B' from the abstract sequence B and the literal $X = t$.

Specification. Let S and S' be (possibly infinite) sequences of substitutions.

$$\left.\begin{array}{l} (\sigma, S) \in Cc(B) \\ \langle \sigma_{i_1}, \sigma_{i_2}, \ldots\rangle \text{ is the greatest subsequence of } S \\ \text{ s.t. } mgu(X\sigma_{i_j}, t\sigma_{i_j}) \neq \emptyset,\ j \in J \\ \sigma'_{i_j} \in mgu(X\sigma_{i_j}, t\sigma_{i_j}),\ j \in J \\ S' = \langle \sigma_{i_1} \circ \sigma'_{i_1}, \sigma_{i_2} \circ \sigma'_{i_2}, \ldots\rangle \end{array}\right\} \Rightarrow (\sigma, S') \in Cc(B').$$

Implementation. Let $B = \langle sv_{in}, sv, frm, \alpha, E\rangle$. The implementation of the `DERIV_UNIF` operation requires the `AI_VAR` and `AI_FUNC` operations for unifying X and t in the abstract substitution $\langle sv, frm, \alpha\rangle$. If variable X occurs in t, the analyzer fails. Otherwise, $\mathtt{DERIV\_UNIF}(B, X = t) = B'$ where $B' = \langle sv'_{in}, sv', frm', \alpha', E'\rangle$ is defined as follows:

- $sv'_{in} = sv_{in}$;
- $\langle sv', frm', \alpha'\rangle = \begin{cases} \mathtt{AI\_VAR}(\langle sv, frm, \alpha\rangle, X = t) & \text{if } t \text{ is a variable} \\ \mathtt{AI\_FUNC}(\langle sv, frm, \alpha\rangle, X = t) & \text{if } t = f(\ldots); \end{cases}$
- $E' = \begin{cases} (tr\Box\{sol \mapsto sol\})^{>}(E) & \text{if either } X \text{ or } t \text{ are variables in } \alpha \\ convex\_hull((tr\Box\{sol \mapsto sol\})^{>}(E), \{0 \leq sol\}) & \text{otherwise.} \end{cases}$

where $tr : I \to I'$ and $\langle sv, frm, \alpha\rangle$ and $\langle sv', frm', \alpha'\rangle$ are parametrized by the indices in I and I', respectively.

5.6 The Derivation Operation: $\mathtt{DERIV\_NORMAL}(B, Beh_p, \tau) = B'$

This operation computes B' from the abstract sequence B, the behaviour Beh_p of the form $(p, [X_1, \ldots, X_n], Prepost)$ for the procedure call $p(X_{i_1}, \ldots, X_{i_n})$ with $i_1, \ldots, i_n \in \{1, \ldots, m\}$ and the renaming function τ assigning each X_i to X_{i_j} $(1 \leq j \leq n)$. The `DERIV_NORMAL` operation is defined only if there exists at least one $\langle B_i, Se_i\rangle$ in *Prepost* such that B satisfies the corresponding preconditions. Formally, we say that $B = \langle sv_{in}, sv, frm, \alpha, E\rangle$ satisfies the preconditions expressed in $B_i = \langle sv^i_{in}, sv^i, frm^i, \alpha^i, E^i\rangle$ iff

- $\mathtt{RESTRG}(p(X_{i_1}, \ldots, X_{i_n}), \langle sv, frm, \alpha\rangle) \leq \langle sv^i_{in}, frm^i, \alpha^i\rangle$;
- and, in the case of a recursive call,
  $E \Rightarrow sz(sv(\tau(Se_i))) < sz(sv_{in}(Se_i))$ (termination criterion).

Specification. Assume that B satisfies the preconditions corresponding to the subset $\langle B_1, Se_1\rangle, \ldots, \langle B_q, Se_q\rangle$ of *Prepost*. The following holds:

$$\left.\begin{array}{l} (\sigma, S) \in Cc(B) \\ S = \langle \sigma_1, \sigma_2, \ldots\rangle \\ (\sigma^j, S^j) \in Cc(B^j),\ 1 \leq j \leq q \\ S^j = \langle \sigma^j_1, \sigma^j_2, \ldots\rangle,\ 1 \leq j \leq q \\ S' = \langle (\sigma_1 \circ \sigma^1_1 \circ \tau^{-1}), (\sigma_1 \circ \sigma^1_2 \circ \tau^{-1}), .., (\sigma_2 \circ \sigma^1_1 \circ \tau^{-1}), ..\rangle \end{array}\right\} \Rightarrow (\sigma, S') \in Cc(B').$$

Implementation. For all $i = 1, \ldots, q$, let $(tr^{<}_i, tr^{>}_i)$ be a constrained mapping with $tr_i : I_i \to I$ where B_i and B are parametrized by the indices in I_i and I, respectively. The implementation operation uses the greatest lower bound $\sqcap$ operator on abstract substitutions. It is as follows:

- $sv'_{in} = sv_{in}$;
- $\langle sv', frm', \alpha'\rangle = \mathtt{EXTG}(p(X_{i_1}, \ldots, X_{i_n}), \langle sv, frm, \alpha\rangle, \sqcap^q_{j=1}\langle sv^j, frm^j, \alpha^j\rangle)$;
- $E' = \Im\_\mathtt{EXTG}(p(X_{i_1}, \ldots, X_{i_n}), E, \bar{E})$

where $\bar{E} = convex\_hull((tr_1\Box\{sol \mapsto sol\})^{>}(E^1), \ldots, (tr_q\Box\{sol \mapsto sol\})^{>}(E^q))$.

6 Complexity Analysis

The analyzer can be used for performing complexity analysis of Prolog programs in the spirit of [9]. In the context of program development, the complexity analysis is useful to choose the most efficient version of a procedure.

Indeed, we can estimate the time complexity of a procedure by using the information relative to the size relations and the number of solutions computed by the analyzer at each program point. Clearly, it depends on the complexity of each literal called in the body of its clauses. Because of nondeterminism, the cost of such a literal depends on the number of solutions generated by the execution of previous literals in the body. Moreover, the cost of a recursive call depends on the depth of the recursion during the computation, which in turn depends on the size of its input arguments. Called predicates are analyzed before the corresponding callers. If two predicates call each other, then they are analyzed together.

The time complexity function for recursive procedures is given in the form of difference equations which are transformed into closed form functions (when possible) using difference equation solving techniques[10].

Let cl be a clause of the form $H \leftarrow L_1, \ldots, L_f$ $(f \geq 0)$, $\bar{A}$ represents a (chosen) input size for cl, and $\bar{A}_i$ represents a (chosen) input size for L_i. Time complexity of cl can be expressed as

$$t_{cl}(\bar{A}) = \tau + \sum_{i=1}^{f} Max_i(\bar{A}_i)\ t_i(\bar{A}_i)$$

where τ is the time needed to unify the call with the head H of cl, $Max_i(\bar{A}_i)$ is an upper bound to the number of solutions generated by the literals preceding L_i, and $t_i(\bar{A}_i)$ is the time complexity of L_i.

Several criteria may be adopted to select the arguments that provide the input size for a procedure call. In [9], Debray *et al.* choose the arguments which are called with ground mode. Our proposal is to use the size expression information in the specification.

There are also different metrics that can be used as unit of time complexity, e.g., the number of resolutions, the number of unifications, or the number of instructions executed. We believe that the number of unification steps is a reasonable choice for estimating the time complexity of a procedure. In this case, since we assume that programs are normalized, i.e., heads are atoms containing distinct variables only, τ is equal to 1. Moreover, the time needed to solve an equality built-in which is called with at least one variable argument is equal to 1. When L is a ground list, and H and T are variables, three unification steps are needed for executing the built-in $L = [H|T]$. For instance, if $L = [a, b]$ then the following steps are performed: (1) replace $[a, b] = [H|T]$ by two equations $H = a$ and $T = [b]$, (2) bind H to a, (3) bind T to $[b]$. Finally, the time complexity for the built-in $list(T)$ is proportional to the size of its argument T. In fact, the execution of such a built-in has to traverse the whole data structure of the term

[10] For the automatic resolution of difference equations the reader is referred to [4, 13].

T in order to check whether it is a list. In this paper, we assume that the time complexity due to the execution of *list*(T) is equal to the size of T plus 1 (noted $T+1$).

Example 3. Let us now compute and compare the time complexities for the procedures $LP_2(select/3)$ and $LP_3(select/3)$ assuming they are called with the arguments being ground, variable and ground, respectively (they respect the second directionality in the specification depicted in Figure 1).

Complexity analysis for $LP_2(select/3)$. The time complexity for $LP_2(select/3)$ in terms of the size of the input ground argument L_s, noted $t_{select,2}$, can be estimated as follows. First, we compute the time complexity $t^i_{select,2}(0)$ for each clause called with L_s being the empty list. In the first clause, both head unification and the body literals may succeed, whereas, in the second clause, only head unification and the first body literal may succeed. This leads to

$$
\begin{aligned}
t^1_{select,2}(0) &= 4 + (1 + T) \\
&= 5 + L_s \qquad \text{(since, } T = L_s\text{)} \\
&= 5 \qquad \text{(since the size of the empty list is 0)} \\
t^2_{select,2}(0) &= 2.
\end{aligned}
$$

Then, the time complexity $t^i_{select,2}(L_s)$ for each clause called with a non empty list L_s is estimated. Since L_s is ground, the cost of the built-in $L_s = [H|T_s]$ in the second clause is equal to 3. In this case both the size relation and the multiplicity information computed by the analyzer are used.

$$
\begin{aligned}
t^1_{select,2}(L_s) &= 5 + T \\
&= 5 + L_s \quad \text{(since } T = L_s\text{)} \\
t^2_{select,2}(L_s) &= 5 + t_{select,2}(T_s) + (T_s + 1)(T + 1) \\
&= 5 + t_{select,2}(L_s - 1) + L_s(T + 1) \qquad \text{(since } T_s = L_s - 1\text{)} \\
&= 5 + t_{select,2}(L_s - 1) + L_s(L_s + 1) \qquad \text{(since } T = L_s\text{).}
\end{aligned}
$$

This leads to the system

$$
\begin{aligned}
t_{select,2}(0) &= 7 \\
t_{select,2}(L_s) &= 10 + 2L_s + L_s^2 + t_{select,2}(L_s - 1)
\end{aligned}
$$

which can be solved returning the time complexity

$$
t_{select,2} \;\equiv\; \lambda x.\, 7 + \Sigma_{i=1}^{x}(10 + 2i + i^2) \qquad (x \text{ stands for the size of } L_s).
$$

Complexity analysis for $LP_3(select/3)$. We compute now the time complexity for $LP_3(select/3)$. As above, when L_s is the empty list, we obtain

$$
\begin{aligned}
t^1_{select,3}(0) &= 5 \\
t^2_{select,3}(0) &= 2.
\end{aligned}
$$

In the case of a non empty list L_s, we compute

$$t^1_{select,3}(L_s) = 5 + L_s$$

$$\begin{aligned} t^2_{select,3}(L_s) &= 5 + t_{select,3}(T_s) \\ &= 5 + t_{select,3}(L_s - 1) \quad \text{(since } T_s = L_s - 1\text{)}. \end{aligned}$$

We obtain the equation system

$$\begin{aligned} t_{select,3}(0) &= 7 \\ t_{select,2}(L_s) &= 10 + L_s + t_{select,3}(L_s - 1) \end{aligned}$$

which returns the time complexity

$$t_{select,3} \equiv \lambda x.\, 7 + \Sigma^x_{i=1}(10 + i) \quad (x \text{ stands for the size of } L_s).$$

Now, it is easy to compare the time complexities $t_{select,2}$ and $t_{select,3}$ that we have estimated for the procedures $LP_2(select/3)$ and $LP_3(select/3)$, respectively, only considering the second directionality in the specification depicted in Figure 1 with the second argument being a variable. The same reasoning can be applied for the other directionality in the specification of $select/3$. In that case, the time complexity can be estimated in terms of the size of the second argument. By combining these results, $LP_3(select/3)$ results to be more efficient than $LP_2(select/3)$.

7 Conclusion and Future Work

In this paper, an analyzer for Prolog procedures has been presented that verifies total correctness with respect to Devilles's formal specification. An automatic complexity analysis based on the information deduced by the analyzer was also proposed. We are conscious that the effective impact of these ideas can be evaluated only after the full implementation of the analyzer, which is in progress, now. The main goal of the implementation, based on the generic abstract interpretation algorithm `GAIA` [16], is to investigate the practicality of the automatic complexity analysis in the context of a logic procedure synthesizer that derives the most efficient procedure among the set of all correct ones.

References

1. A. Bossi, N. Cocco, and M. Fabris. Norms on Terms and Their Use in Proving Universal Termination of a Logic Program, Theoretical Computer Science, 124:297-328, 1994.
2. C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality Analysis of Prolog. In Proc. Int'l Logic Programming Symposium, (ILPS'94), Ithaca, NY. The MIT Press, Cambridge, Mass., 1994.
3. K.L. Clark. Negation as Failure. In H. Gallaire and J. Minker editors, Advances in Data Base Theory, Plenum Press, New York, pp. 293-322, 1978.
4. J. Cohen and J. Katcoff. Symbolic solution of finite-difference equations. ACM Transactions on Mathematical Software, 3(3):261–271, 1977.

5. A. Cortesi, B. Le Charlier and P. van Hentenryck, Conceptual and Software Support for Abstract Domain Design: Generic Structural Domain and Open Product. Technical Report CS-93-13, Brown University, 1993.
6. A. Cortesi, B. Le Charlier and P. van Hentenryck, Combinations of Abstract Domains for Logic Programming, In Proc. 21th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94), ACM-Press, New York, pp.227-239, 1994.
7. P. De Boeck and B. Le Charlier. Static Type Analysis of Prolog Procedures for Ensuring Correctness. In P. Deransart and J. Maluszyński, editors, Proc. Second Int'l Symposium on Programming Language Implementation and Logic Programming, (PLILP'90), LNCS 456, Springer-Verlag, Berlin, 1990.
8. P. De Boeck and B. Le Charlier. Mechanical Transformation of Logic Definitions Augmented with Type Information into Prolog Procedures: Some Experiments. In Proc. Int'l Workshop on Logic Program Synthesis and Transformation, (LOPSTR'93). Springer Verlag, July 1993.
9. S. K. Debray and N. W. Lin. Cost analysis of logic programs. ACM Transactions on Programming Languages and Systems, 15(5):826–875, 1993.
10. D. De Schreye, K. Verschaetse, and M. Bruynooghe. A Framework for analyzing the termination of definite logic programs with respect to call patterns. In H. Tanaka, editor, FGCS'92, 1992.
11. Y. Deville. *Logic Programming: Systematic Program Development*. Addison-Wesley, 1990.
12. J. Henrard and B. Le Charlier. FOLON: An Environment for Declarative Construction of Logic Programs (extended abstract). In M. Bruynooghe and M. Wirsing, editors, Proc. Fourth Int'l Workshop on Programming Language Implementation and Logic Programming (PLILP'92), LNCS 631 Springer-Verlag, pp. 237-231, 1992.
13. J. Ivie. Some MACSYMA programs for solving recurrence relations. ACM Transactions on Mathematical Software, 4(1):24–33, 1978.
14. B. Le Charlier, and S. Rossi. Automatic Derivation of Totally Correct Prolog Procedures from Logic Descriptions. Research Report RP-95-009, University of Namur.
15. B. Le Charlier, S. Rossi, and P. Van Hentenryck. An Abstract Interpretation Framework which Accurately Handles Prolog Search-Rule and the Cut. In Proc. Int. Logic Programming Symposium, (ILPS'94), Ithaca, NY. The MIT Press, Cambridge, Mass., 1994.
16. B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. ACM Transactions on Programming Languages and Systems, 16(1), pp. 35–101,1994.
17. C. Leclère and B. Le Charlier. Two Dual Abstract Operations to Duplicate, Eliminate, Equalize, Introduce and Rename Place-Holders Occurring Inside Abstract Descriptions. Research Paper N. RP-96-028, University of Namur, Belgium, September 1996.
18. J. W. Lloyd. Foundations of Logic Programming. Springer-Verlag, Berlin, 1987. Second edition.
19. P. van Hentenryck, A. Cortesi, and B. Le Charlier, Evaluation of the Domain Prop. The Journal of Logic Programming 23(3):237-278, 1995.

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