Specification-based Automatic Verification of Prolog Programs[1]

Bauduin Le Charlier
Institut d'Informatique
21 rue Grandgagnage
B-5000 Namur (Belgium)
ble@info.fundp.ac.be

Sabina Rossi
Dip. di Matematica
via Belzoni 7
35131 Padova (Italy)
sabina@hilbert.math.unipd.it

Agostino Cortesi
Dip. Mat. Appl. e Informatica
via Torino 155
30170 Mestre-Venezia (Italy)
cortesi@dsi.unive.it

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.

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 [10], Deville proposes a methodology for logic program construction that aims at reconciling the declarative semantics with an efficient Prolog implementation. The methodology is based on four main development steps:

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

The FOLON environment [7, 11] 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.

The new analyzer is an extension of the analyzers described in [6, 7, 11] which themselves automate part of the methodology described in [10]. The novelty is that we deal with total correctness, while [6, 7, 11] only deal with particular correctness aspects such as mode and type verification. At this aim, we extend the generic abstract domain $\mathtt{Pat}(\Re)$ [4, 5] to deal with multiplicity and size relations.

We adapt the framework for termination analysis proposed by De Schreye, Verschaetse and Bruynooghe in [9]: 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 introduced in [14]. This is related to the cardinality analysis described in [2]. However, we do not perform an

[1] 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"

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].

Based on the new analyzer, we get almost for free an automatic complexity analysis, in the spirit of the framework proposed by Debray and Lin in [8]. In our context, this analysis is useful to choose the most efficient version of a procedure.

The paper is organized as follows. In Section 1, we recall Deville's methodology. In Section 2, we introduce the abstract domains used by the analyzer. In Section 3, we describe the analyzer and give an example to illustrate the main operations. In Section 4, we discuss the complexity analysis and describe an example of cost analysis which uses information computed by the analyzer. Section 5 concludes the paper.

1 Program Synthesis Methodology Overview

Our work is based on and extends Deville's methodology for logic program development presented in [10]. 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 $\mathtt{L_s}$.

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 [10] 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 [8]). They are used to prove termination of the recursive procedures in the spirit of [9]. 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 relation and the size relation express properties on the formal parameters which are intended to be satisfied after any successful execution. The size measure (see [9]) $\|\cdot\|_\lambda$ associates to each term $\mathtt{t}$ a natural number $\|\mathtt{t}\|_\lambda$ by: $\|[\mathtt{t_1}|\mathtt{t_2}]\|_\lambda = 1 + \|\mathtt{t_2}\|_\lambda$ and $\|\mathtt{t}\|_\lambda = 0$ if $\mathtt{t}$ is not of the form $[\mathtt{t_1}|\mathtt{t_2}]$. The size relation is a set of (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 $\mathtt{L_s}$ plus 1.

Procedure: $\mathtt{select(X, L, L_s)}$
Relation: `X` is an element of `L` and $\mathtt{L_s}$ is the list `L` without an occurrence of `X`
Size measure: $\|\cdot\|_\lambda$
Size relation: $\{\mathtt{L} = \mathtt{L_s} + 1\}$

Application conditions:			
	$\mathbf{in}(\mathtt{any}, \mathtt{gr}, \mathtt{any}) :: \mathbf{out}(\mathtt{gr}, \mathtt{grlist}, \mathtt{grlist})$	$\{0 \leq \mathtt{sol} \leq \mathtt{L}\}$	$\mathtt{L}$
	$\mathbf{in}(\mathtt{gr}, \mathtt{any}, \mathtt{gr}) :: \mathbf{out}(\mathtt{gr}, \mathtt{grlist}, \mathtt{grlist})$	$\{0 \leq \mathtt{sol} \leq \mathtt{L_s} + 1\}$	$\mathtt{L_s}$

Figure 1: Specification for `select/3`

The application conditions consist of three components namely, *directionality, multiplicity* and *size expression*. Each *directionality* specifies the allowed modes[2] of the parameters before the execution (**in** part) and the corresponding modes after a successful execution (**out** part). In order to describe the multiplicity and the size expression components, let us introduce some definitions.
For any set of program variables V, we denote $\mathbf{Exp}_\mathrm{V}$ the set of all expressions on variables in V. A

[2] In this example we restrict our attention to the domain `Mode-Type` [13] (`gr` denotes a ground term, `any` denotes any term, `var` denotes a variable, `grlist` denotes a ground list, `anylist` denotes either a variable or any list). This can be generalized to any information on procedure parameters.

multiplicity is a set E of inequations over $\mathbf{Exp}_{\{sol,X_1,\ldots,X_n\}} \cup \{\infty\}$ where $\mathtt{X_1},\ldots,\mathtt{X_n}$ are the formal parameters of the procedure and *sol* is a new variable denoting the cardinality of the answer substitution set. 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: $sol/|S|$ satisfies the restriction to *sol* of the set of inequations obtained from E by replacing the formal parameters $\mathtt{X_1},\ldots,\mathtt{X_n}$ with the size of the corresponding input values. The *size expression* is an expression from $\mathbf{Exp}_{\{\mathtt{X_1},\ldots,\mathtt{X_n}\}}$[3] 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 a weight is assumed to be not affected by any further instantiation of the parameters. In order to prove termination of a recursive call we need to prove that its weight is smaller than the weight for the initial one[4]. 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. $\mathtt{L_s}$). Since this value is required to be ground, according to the specified directionality, such a weight is guaranteed to be invariant for any further instantion of the parameters.

Logical Description

A correct logic description for `select/3`, noted `LD(select/3)`, is

$$\mathtt{select(X,L,L_s)} \iff \mathtt{L} = [\mathtt{H}|\mathtt{T}] \wedge ((\mathtt{H} = \mathtt{X} \wedge \mathtt{L_s} = \mathtt{T} \wedge \mathtt{list(T)}) \vee (\mathtt{L_s} = [\mathtt{H_s}|\mathtt{T_s}] \wedge \mathtt{select(X,T,T_s)})).$$

The correctness of a logic description `LD(p/n)` can be expressed as follows: for every ground n-tuple term 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.

Prolog Code Derivation

The next step consists in deriving a Prolog program from the logic description. The logic description `LD(p/n)` is syntactically translated into a Prolog program `LP(p/n)` whose completion [16] is the logic description again. In the example, the following Prolog program `LP(select/3)` is obtained.

```
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).
```

Correctness Verification

The last step is the verification of total correctness of the program with respect to the specification. To be correct that procedure has to respect 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 reached according to Prolog search rule);
6. termination: if S is finite then the execution terminates.

[3] It corresponds, in a sense, to the natural level mapping used by De Schreye *et al.* in [9].
[4] For more details about these concepts the reader is referred to [1, 9].

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 sequence 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 sequence of answer substitutions, then none of the preceeding literals in this clause produces more than one solution.

The main advantage provided by the analyzer that we present in this paper is the fully automatization of the last step of Deville's methodology described so far, i.e. the verification of total correctness of synthetized programs with respect to the formal specification. In section 3 we will illustrate the behaviour of the analyzer by proving that the procedure LP(select/3) is correct wrt the specification depicted in Figure 1.

2 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 solution set cardinality. Finally, we formalize the program specification through the notion of *behaviour*.
In the following we call I_p 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 [15].

Definition 2.1 [abstract substitution]
An *abstract substitution* β is an element of the generic abstract domain Pat($\Re$) described in [4, 5], i.e. a tuple $\langle frm, sv, \alpha \rangle$ where the pattern component frm associates with some of the indices in I_p a pattern $f(i_1, \ldots, i_q)$, where f a q-ary function symbol and $\{i_1, \ldots, i_q\} \subseteq I_p$; the same-value component sv assigns a subterm to each variable in the substitution; and 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 2.1 Let $\Re$ be the abstract domain Mode-Type described in [13], and consider the (concrete) substitution

$$\{\texttt{X1} \mapsto \texttt{Y} * \texttt{a}, \texttt{X2} \mapsto \texttt{a}, \texttt{X3} \mapsto [\,]\}.$$

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

sv:		frm:		α	
	X1 $\mapsto$ 1		1 $\mapsto$ 4 * 2		1/ngv
	X2 $\mapsto$ 2		2 $\mapsto$ a		2/gr
	X3 $\mapsto$ 3		3 $\mapsto$ []		3/grlist
			4 $\mapsto$?		4/var

At each execution point, the analyzer computes (so-called) abstract sequences (see [2, 14]) 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 assignements at clause entry, and a set of linear (in)equations represents size relations among terms and solution set cardinality.

Definition 2.2 [abstract sequence]
An *abstract sequence* $\mathcal{B}$ is a tuple $\langle frm, sv_{in}, sv, \alpha, \texttt{E} \rangle$ where

1. $\langle frm, sv_{in}, \alpha \rangle$ and $\langle frm, sv, \alpha \rangle$ are abstract substitutions.
2. the domain of the same-value function sv_{in} is contained in the domain of sv;

3. the *size component* $\mathtt{E}$ is a (possibly empty) set of (in)equations over $\mathbf{Exp}_{\{sol,\mathtt{sz(1)},\ldots,\mathtt{sz(p)}\}}$.

Given a size measure $\|\cdot\|$, an abstract sequence $\mathcal{B}$= $\langle frm, sv_{in}, sv, \alpha, \mathtt{E}\rangle$ represents the pairs $(\sigma_{in}, \mathtt{S})$ of concrete substitutions such that $\sigma_{in} \in \gamma(\langle frm, sv_{in}, \alpha\rangle)$ and for all $\sigma \in \mathtt{S}$, σ is an instance of $\sigma_{\mathtt{in}}$, $\sigma \in \gamma(\langle frm, sv, \alpha\rangle)$, and $\mathtt{S}$ respects the multiplicity relations expressed by $\mathtt{E}$, i.e. for every $\langle t_1, \ldots, t_p\rangle \in \gamma(\alpha)$: $(\mathtt{sz(1)}/\|\mathtt{t_1}\|, \ldots, \mathtt{sz(p)}/\|\mathtt{t_p}\|, sol/|\mathtt{S}|)$ satisfies the set of inequations $\mathtt{E}$.

A behaviour for a procedure is a formalization of its specification (excluding the relation part).

Definition 2.3 [behaviour]
A *behaviour* for a procedure $\mathtt{p/n}$ is a 4-tuple of the form $(\mathtt{p}, [\mathtt{X_1}, \ldots, \mathtt{X_n}], \mathtt{S}, \mathtt{Prepost})$ where

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

Example 2.2 A behaviour for $\mathtt{select/3}$ formalizing the specification depicted in Figure 1 has the form

$$(\mathtt{select}, [\mathtt{X_1}, \mathtt{X_2}, \mathtt{X_3}], \mathtt{S} = \{\mathtt{X_2} = \mathtt{X_3} + 1\}, \mathtt{Prepost} = \{\langle \mathcal{B}^-, \mathtt{X_2}\rangle, \langle \mathcal{B}^+, \mathtt{X3}\rangle\})$$

where $\mathcal{B}^+$ (corresponding to the second application condition) is equal to

sv_{in}:	$\mathtt{X1} \mapsto 1$	sv_{out}:	$\mathtt{X1} \mapsto 4$	frm:	$1 \mapsto ?$	α :	$1/\mathtt{gr}$	E:	$0 \leq sol \leq \mathtt{sz(3)} + 1$
	$\mathtt{X2} \mapsto 2$		$\mathtt{X2} \mapsto 5$		$2 \mapsto ?$		$2/\mathtt{any}$		$\mathtt{sz(5)} = \mathtt{sz(6)} + 1$
	$\mathtt{X3} \mapsto 3$		$\mathtt{X3} \mapsto 6$		$3 \mapsto ?$		$3/\mathtt{gr}$		
					$4 \mapsto ?$		$4/\mathtt{gr}$		
					$5 \mapsto ?$		$5/\mathtt{grlist}$		
					$6 \mapsto ?$		$6/\mathtt{grlist}$		

and the abstract sequence $\mathcal{B}^-$ (corresponding to the first application condition) can be defined in a similar way.

3 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 [6]. Then, we describe a procedure analyzer, based on the clause analyzer, for verifying correctness of a whole procedure.

3.1 The Clause Analyzer

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

1. for any subcall to a procedure $\mathtt{q/m}$ in the body of $\mathtt{CL}$, let $(\mathtt{q}, [\mathtt{X_1}, \ldots, \mathtt{X_m}], \mathtt{S_q}, \mathtt{Prepost_q})$ be the behaviour for $\mathtt{q/m}$. The procedure $\mathtt{q/m}$ is called with arguments respecting at least one precondition $\langle frm', sv'_{in}, \alpha'\rangle$ in the $\mathtt{Prepost_q}$ set;
2. the arguments of $\mathtt{p}$ after the execution of the clause $\mathtt{CL}$ respect the size relation and $\langle frm, sv_{out}, \alpha\rangle$;

3. if sol/∞ satisfies the restriction of E to sol, then at most one subcall in the body of CL can produce infinite solutions and none of the preceeding 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 (in)equation set $\mathtt{E_{CL}}$ expressing information about the length of the sequence of answer substitutions and the termination of the execution of the clause CL.

In order to compute its results, the clause analyzer computes a set, $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_\mathtt{f}$, of abstract sequences on the variables in CL and an abstract sequence $\mathcal{B}_\mathtt{out}$ expressing the same properties as $\mathcal{B}_\mathtt{f}$ but restricted to the variables $\mathtt{X_1}, \ldots, \mathtt{X_n}$ in the head of CL. The following main operations are required to analyze a clause.

- **Initial operation**: it extends β to an abstract sequence $\mathcal{B}_0$ on all variables in CL.
- **Derivation operation**: it computes $\mathcal{B}_\mathtt{i}$ from $\mathcal{B}_{\mathtt{i}-1}$ and $\mathtt{L_i}$ ($1 \leq \mathtt{i} \leq \mathtt{f}$). In fact, there are two kinds of derivation operations: the *deriv-unif* (for the = /2 literals) and the *deriv-normal* (for the other literals).
- **Reduction operation**: it computes $\mathcal{B}_\mathtt{out}$ from $\mathcal{B}_\mathtt{f}$ by restricting it to the variables $\mathtt{X_1}, \ldots, \mathtt{X_n}$.
- **Exit operation**: it verifies whether $\mathcal{B}_\mathtt{out}$ respects the size relation in E and $\langle frm, sv_{out}, \alpha \rangle$.

Let us illustrate them with an example. Consider the second clause of the procedure select/3 and the behaviour for select/3 described above. Let $\beta = \langle frm^*, sv^*, \alpha^*, \mathtt{E}^* \rangle$ be an abstract substitution with $frm^* = \{1 \mapsto ?, 2 \mapsto ?, 3 \mapsto ?\}$, $sv^* = \{\mathtt{X1} \mapsto 1, \mathtt{X2} \mapsto 2, \mathtt{X3} \mapsto 3\}$, $\alpha^* = \{1/\mathtt{gr}, 2/\mathtt{any}, 3/\mathtt{gr}\}$, $\mathtt{E}^* = \{sol = 0\}$. The clause analyzer camputes $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_4$ as follows:

β
select(X1,X2,X3) $\leftarrow$ $\mathcal{B}_0$ X2=[X4|X5],
$\mathcal{B}_1$ X3=[X4|X6],
$\mathcal{B}_2$ select(X1,X5,X6) $\mathcal{B}_3$.
$\mathcal{B}_4$.

Initial Operation First, β is extended to an abstract substitution $\mathcal{B}_0$ on all the variables in the clause stating that the new variables are not instantiated and used nowhere else. In the example,

sv^0_{in}:	sv^0:	frm^0:	α^0:	$\mathtt{E}^0$:
X1 $\mapsto$ 1	X1 $\mapsto$ 1	1 $\mapsto$?	1/gr	$sol = 0$
X2 $\mapsto$ 2	X2 $\mapsto$ 2	2 $\mapsto$?	2/any	
X3 $\mapsto$ 3	X3 $\mapsto$ 3	3 $\mapsto$?	3/gr	
	X4 $\mapsto$ 4	4 $\mapsto$?	4/var	
	X5 $\mapsto$ 5	5 $\mapsto$?	5/var	
	X6 $\mapsto$ 6	6 $\mapsto$?	6/var	

Derivation operation: We compute $\mathcal{B}_1$ from $\mathcal{B}_0$ and the literal X2 = [X4|X5]. 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 X2 with [X4|X5]. The sv_{in} component maintain the link with the term denoted by 2 while the actual binding of X2 after unification is kept by sv. Since it is not sure that the unification succeeds, the information that a failure could occur is expressed in the multiplicity component. Information on the size relations holding after the unification is expressed by the equation sz(7)=sz(5)+1. Thus, $\mathcal{B}_1$ is

sv^1_{in}:	sv^1:	frm^1:	α^1:	$\mathtt{E}^1$:
X1 $\mapsto$ 1	X1 $\mapsto$ 1	1 $\mapsto$?	1/gr	$0 \leq sol \leq 1$
X2 $\mapsto$ 2	X2 $\mapsto$ 7	2 $\mapsto$?	2/any	sz(7)=sz(5) + 1
X3 $\mapsto$ 3	X3 $\mapsto$ 3	3 $\mapsto$?	3/gr	
	X4 $\mapsto$ 4	4 $\mapsto$?	4/any	
	X5 $\mapsto$ 5	5 $\mapsto$?	5/any	
	X6 $\mapsto$ 6	6 $\mapsto$?	6/var	
		7 $\mapsto$ [4|5]	7/ngv	

Another derivation operation applies to derive $\mathcal{B}_2$ from $\mathcal{B}_1$ and the literal `X3 = [X4|X6]`. Also in this case, the *deriv-unif* operation is used. Observe that in this case, the groundness of term indexed by `3` propagates to the terms that correspond (through sv) to `X4,X6`. Observe that no new term is created, because the term indexed by `3` was ground already. The abstract sequence $\mathcal{B}_2$ is

sv^2_{in}:		sv^2:		frm^2:		α^2 :		$\mathtt{E}^2$:	
	$\mathtt{X1} \mapsto 1$		$\mathtt{X1} \mapsto 1$		$1 \mapsto ?$		$1/\mathtt{gr}$		$0 \leq sol \leq 1$
	$\mathtt{X2} \mapsto 2$		$\mathtt{X2} \mapsto 7$		$2 \mapsto ?$		$2/\mathtt{any}$		`sz(7)=sz(5) + 1`
	$\mathtt{X3} \mapsto 3$		$\mathtt{X3} \mapsto 3$		$3 \mapsto [4\|6]$		$3/\mathtt{gr}$		`sz(3)=sz(6) + 1`
			$\mathtt{X4} \mapsto 4$		$4 \mapsto ?$		$4/\mathtt{gr}$		
			$\mathtt{X5} \mapsto 5$		$5 \mapsto ?$		$5/\mathtt{any}$		
			$\mathtt{X6} \mapsto 6$		$6 \mapsto ?$		$6/\mathtt{gr}$		
					$7 \mapsto [4\|5]$		$7/\mathtt{ngv}$		

The abstract sequence $\mathcal{B}_3$ is obtained through *deriv-normal* by combining $\mathcal{B}_2$ with the behaviour of `select/3`. In particular, we need to find a pair $\langle \mathcal{B}, \mathtt{Se} \rangle$ in the `Prepost` component of the behaviour that matches with $\mathcal{B}_2$ regarding *directionality* and *termination*. In our example, this is true when considering the prepost component $\langle \mathcal{B}^+, \mathtt{X_3} \rangle$ defined at the end of Section 2.
Let $\mathcal{B}^+ = \langle frm, sv_{in}, sv_{out}\alpha, \mathtt{E} \rangle$ and let $\tau = \{\mathtt{X1} \mapsto \mathtt{X1}, \mathtt{X2} \mapsto \mathtt{X5}, \mathtt{X3} \mapsto \mathtt{X6}\}$ be the renaming function that is used to restrict $\mathcal{B}_2$ to the clause `select(X1,X5,X6)`.

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

$$\begin{array}{l} \mathbf{in}(\alpha^2(sv^2(\tau(\mathtt{X1}))), \alpha^2(sv^2(\tau(\mathtt{X2}))), \alpha^2(sv^2(\tau(\mathtt{X3})))) = \\ \mathbf{in}(\alpha^2(sv^2(\mathtt{X1})), \alpha^2(sv^2(\mathtt{X5})), \alpha^2(sv^2(\mathtt{X6}))) = \\ \mathbf{in}(\alpha^2(\mathtt{1}), \alpha^2(\mathtt{5}), \alpha^2(\mathtt{6})) = \\ \mathbf{in}(\mathtt{gr}, \mathtt{any}, \mathtt{gr}) = \\ \mathbf{in}(\alpha(sv(\mathtt{X1}), \alpha(sv(\mathtt{X2}), \alpha(sv(\mathtt{X3})) \end{array}$$

Second, since sol/∞ is not a solution in $\mathtt{E}^2$, we have to prove termination: according to the size expression `Se = X3` 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

$$\mathtt{sz}(sv^2(\tau(\mathtt{X3}))) < \mathtt{sz}(sv^2_{in}(\mathtt{X3}))$$

Indeed,

$$\begin{array}{rcll} \mathtt{sz}(sv^2_{in}(\mathtt{X3})) & = & \mathtt{sz}(\mathtt{3}) & \text{by definition of } sv^2_{in} \\ & = & \mathtt{sz}(\mathtt{6}) + \mathtt{1} & \text{by multiplicity in } \mathcal{B}_2 \\ & > & \mathtt{sz}(\mathtt{6}) & \\ & = & \mathtt{sz}(sv^2(\tau(\mathtt{X3}))). & \end{array}$$

Thus, both directionality and termination of the application condition are satisfied. Therefore, we can apply the out-conditions of $\mathcal{B}$. In particular, we derive that the terms indexed by `5` and `6` are bound to ground lists, and we get the multiplicity equations $0 \leq sol \leq \mathtt{sz}(\mathtt{6}) + \mathtt{1}$ and $\mathtt{sz}(\mathtt{5}) = \mathtt{sz}(\mathtt{6}) + \mathtt{1}$. We obtain the following abstract sequence $\mathcal{B}_3$:

sv^3_{in}:		sv^3:		frm^3:		α^3 :		$\mathtt{E}^3$:	
	$\mathtt{X1} \mapsto 1$		$\mathtt{X1} \mapsto 1$		$1 \mapsto ?$		$1/\mathtt{gr}$		$0 \leq sol \leq \mathtt{sz}(\mathtt{6}) + \mathtt{1}$
	$\mathtt{X2} \mapsto 2$		$\mathtt{X2} \mapsto 7$		$2 \mapsto ?$		$2/\mathtt{any}$		`sz(7)=sz(5) + 1`
	$\mathtt{X3} \mapsto 3$		$\mathtt{X3} \mapsto 3$		$3 \mapsto [4\|6]$		$3/\mathtt{grlist}$		`sz(3)=sz(6)+1`
			$\mathtt{X4} \mapsto 4$		$4 \mapsto ?$		$4/\mathtt{gr}$		`sz(5)=sz(6) + 1`
			$\mathtt{X5} \mapsto 5$		$5 \mapsto ?$		$5/\mathtt{grlist}$		
			$\mathtt{X6} \mapsto 6$		$6 \mapsto ?$		$6/\mathtt{grlist}$		
					$7 \mapsto [4\|5]$		$7/\mathtt{grlist}$		

Reduction operation. The abstract sequence $\mathcal{B}_4$ is computed by rectricting $\mathcal{B}_3$ to the variables `X1,X2,X3`. It results in

sv^3_{in}:	sv^3:	frm^3:	α^3 :	$\mathtt{E}^3$:
$\mathtt{X1} \mapsto 1$	$\mathtt{X1} \mapsto 1$	$1 \mapsto ?$	$1/\mathtt{gr}$	$0 \leq sol \leq \mathtt{sz(6)} + 1$
$\mathtt{X2} \mapsto 2$	$\mathtt{X2} \mapsto 7$	$2 \mapsto ?$	$2/\mathtt{any}$	$\mathtt{sz(7)=sz(5)\ +\ 1}$
$\mathtt{X3} \mapsto 3$	$\mathtt{X3} \mapsto 3$	$3 \mapsto [4\|6]$	$3/\mathtt{grlist}$	$\mathtt{sz(3)=sz(6)\ +\ 1}$
		$4 \mapsto ?$	$4/\mathtt{gr}$	$\mathtt{sz(5)=sz(6)\ +\ 1}$
		$5 \mapsto ?$	$5/\mathtt{grlist}$	
		$6 \mapsto ?$	$6/\mathtt{grlist}$	
		$7 \mapsto [4\|5]$	$7/\mathtt{grlist}$	

Exit operation The last step in the clause analysis consists in verifying whether $\mathcal{B}_4$ satisfies the size relation in `E` and the output abstract substitution $\langle frm, sv_{out}, \alpha \rangle$ of the `Prepost` component $\langle \mathcal{B}^+, \mathtt{X3} \rangle$ of the behaviour applied so far. In our example, this is trivially true.

3.2 The Procedure Analyzer

Using the clause analyzer, we define a procedure analyzer which given a Prolog program `P` defining a predicate `p/n` and a behaviour for each subprocedure in `P`, checks the following: for each abstract substitution respecting at least one precondition in the behaviour for `p/n` with multiplicity (in)equation set `E`,

1. for each clause `CL` of `P` the clause analyzer does not fail;
2. the sequence of answer substitutions for the whole procedure respects the multiplicity (in)equation set;
3. if sol/∞ is a solution for the restriction of `E` to sol, then at most one clause of `P` can produce an infinite number of solutions and none of the preceeding literals in the clause produces more than one solution. 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 meaning that the correctness of `P` has not been proved.

4 Complexity Analysis

The analyzer provides a suitable basis for the complexity analysis of Prolog programs in the spirit of [8]. The complexity analysis is useful to choose the most efficient version of a procedure.

Indeed, using the information relative to the size relations and the number of solutions computed by the analyzer at each program point, the time complexity of a procedure can be easily estimated. 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.

Callee 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[5].

Let `CL` be a clause of the form $\mathtt{H} \leftarrow \mathtt{L}_1, \ldots, \mathtt{L}_\mathtt{f}$ $(\mathtt{f} \geq 0)$, $\bar{\mathtt{A}}$ represent the input size for `CL` and $\bar{\mathtt{A}}_\mathtt{i}$ represent the input size for $\mathtt{L}_\mathtt{i}$. The time complexity of `CL` can be expressed as

$$\mathtt{t}_{\mathtt{CL}}(\bar{\mathtt{A}}) \quad = \quad \tau + \sum_{\mathtt{i}=1}^{\mathtt{f}} \mathtt{Max}_\mathtt{i}(\bar{\mathtt{A}}_\mathtt{i})\ \mathtt{t}_\mathtt{i}(\bar{\mathtt{A}}_\mathtt{i})$$

[5] For the automatic resolution of general difference equations the reader is referred to [3, 12].

where τ is the time needed to unify with the head H of CL, $\mathtt{Max_i}(\bar{\mathtt{A}}_\mathtt{i})$ is an upper bound to the number of solutions generated by the literals preceeding $\mathtt{L_i}$ and $\mathtt{t_i}(\bar{\mathtt{A}}_\mathtt{i})$ is the time complexity of $\mathtt{L_i}$.

There are a number of different metrics that can be used as the unit of time complexity, e.g., the number of resolutions, the number of unifications, or the number of instructions executed. For simplicity, in what follows, we assume that the time complexity metric used is the number of resolutions giving an upper bound on the number of vertices in the search tree. In this case, both τ and the time needed to solve a built-in is 1.

Example 4.1 Consider once more the program `select/3` and the second directionality in the specification depicted in Figure 1. The time complexity for `select/3` in terms of the size of the input ground argument $\mathtt{L_s}$, noted $\mathtt{t_{select}}$, can be estimated as follows. First, we compute the time complexity $\mathtt{t^i_{select}}(0)$ for each clause called with $\mathtt{L_s}$ being the empty list.

$$\mathtt{t^1_{select}}(0) \quad = \quad 5 \quad \text{(in the first clause, both head unification and the body literals succeed)}$$

$$\mathtt{t^2_{select}}(0) \quad = \quad 2 \quad \text{(in the second clause, only head unification and the first body literal succeed).}$$

Then, the time complexity $\mathtt{t^i_{select}}(\mathtt{L_s})$ for each clause called with a non empty list $\mathtt{L_s}$ is estimated. In this case both the size relation and the multiplicity information computed by the analyzer are used.

$$\mathtt{t^1_{select}}(\mathtt{L_s}) \quad = \quad 5 \quad \text{(in the first clause, both head unification and the body literals succeed)}$$

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

The time complexities $\mathtt{t_{select}}(0)$ and $\mathtt{t_{select}}(\mathtt{L_s})$ for the calls of `select/3` with $\mathtt{L_s}$ being the empty list and a non empty list, respectively, are obtained by summing the time complexity for the first two clauses.

$$\begin{aligned} \mathtt{t_{select}}(0) &= 7 \\ \mathtt{t_{select}}(\mathtt{L_s}) &= 8 + \mathtt{t_{select}}(\mathtt{L_s} - 1). \end{aligned}$$

This system can be solved to obtain the time complexity

$$\mathtt{t_{select}} \equiv \lambda \mathtt{x}.\ 8\mathtt{x} + 7 \quad (\mathtt{x} \text{ standing for the size of } \mathtt{L_s}).$$

5 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 [15], 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. Proving Termination of Logic Programs by Exploiting Term Properties. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. TAPSOFT'91*, volume 494 of *Lecture Notes in Computer Science*, pages 153–180. Springer-Verlag, Berlin, 1991.

[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] J. Cohen and J. Katcoff. Symbolic solution of finite-difference equations. *ACM Transactions on Mathematical Software*, 3(3):261–271, 1977.

[4] 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.

[5] A. Cortesi, B. Le Charlier and P. van Hentenryck, Combinations of Abstract Domains for Logic Programming, *Proc. 21th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, ACM-Press, New York, pp.227-239, 1994.

[6] 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)*, volume 456 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1990.

[7] 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.

[8] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.

[9] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A Framework for analysing the termination of definite logic programs with respect to call patterns. In H. Tanaka, editor, *FGCS'92*, 1992.

[10] Y. Deville. *Logic Programming: Systematic Program Development.* Addison-Wesley, 1990.

[11] 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)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1992.

[12] J. Ivie. Some MACSYMA programs for solving recurrence relations. *ACM Transactions on Mathematical Software*, 4(1):24–33, 1978.

[13] B. Le Charlier, and S. Rossi. Automatic Derivation of Totally Correct Prolog Procedures from Logic Descriptions. Research Report RP-95-009, University of Namur.

[14] 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'l Logic Programming Symposium, (ILPS'94), Ithaca, NY.* The MIT Press, Cambridge, Mass., 1994.

[15] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for prolog. *ACM Transactions on Programming Languages and Systems*, 1993.

[16] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second edition.