AUTOMATED VERIFICATION OF PROLOG PROGRAMS

BAUDOUIN LE CHARLIER, CHRISTOPHE LECLÈRE, SABINA ROSSI AND AGOSTINO CORTESI

▷ Although Prolog is (still) the most widely used logic language, it suffers from a number of drawbacks which prevent it from being truely declarative. The non declarative features such as the depth-first search rule are nevertheless necessary to make Prolog reasonably efficient. Several authors have proposed methodologies to reconcile declarative programming with the algorithmic features of Prolog. The idea is to analyse the logic program with respect to a set of properties such as modes, types, sharing, termination, and the like in order to ensure that the operational behaviour of the Prolog program complies with its logic meaning. Such analyses are tedious to perform by hand and can be automated to some extent.
This paper presents a state-of-the-art analyser which allows one to integrate many individual analyses previously proposed in the literature as well as new ones. Conceptually, the analyser is based on the notion of abstract sequence which makes it possible to collect all kinds of desirable information, including relations between the input and output sizes of terms, multiplicity, and termination. ◁

1. INTRODUCTION

Declarative and logic languages allow the programmer to concentrate on the description of the problem to be solved and to ignore low level implementation details. Nevertheless, their implementation remains a delicate issue: since efficiency

Address correspondence to Baudouin Le Charlier, Institut d'Informatique, University of Namur, 21 rue Grandgagnage, B-5000 Namur, Belgium. E-mail: `ble@info.fundp.ac.be`

THE JOURNAL OF LOGIC PROGRAMMING

©Elsevier Science Publishing Co., Inc., 1993
655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

is a major concern for most applications, "real" declarative languages often deviate from the declarative paradigm and include additional "impure" features, which are intended to improve the efficiency of the language but often ruin its declarative nature. This is what happens in logic programming with Prolog, which is characterized by an incomplete (depth-first) search rule, a non logical negation (by failure), and a number of non logical operations such as the test predicates (e.g., `var`) and the cut. In order to improve on this situation different approaches have been investigated in the recent years. In particular,

- static analyses, mainly based on abstract interpretation, have been developed aiming at optimizing Prolog programs, relieving the programmer from using impure control features [47, 56];
- new languages have been defined, like Mercury [61], that improve on declarativeness; efficiency is kept by asking the programmer to specialize its code with mode and type declarations.

In this paper, we look at the problem from a different perspective. Instead of been targeted towards optimizations, we follow the approach depicted in [54] and we show how Prolog program verification (a very demanding software engineering task) may benefit from techniques of static analysis [19], as recently pointed out in a more abstract setting also by [36, 50].

The aim of this work is to introduce a tool to verify that a non declarative implementation of a program (a Prolog code) in fact behaves according to its declarative meaning (a declarative specification given by the user). This verification process can be used also to transform a first (declaratively but not operationally correct) version of a program into a both declaratively and operationally correct version.

In order to define such a verifier, we greatly benefit from works on static analysis of Prolog programs. The analyser presented here is general enough to integrate most automatable analyses previously described in the literature. The design of the analyser is based on the methodology of abstract interpretation, where information provided by the user is used instead of performing a fixpoint computation. It could be integrated in a programming environment to check the correctness of Prolog programs and/or to derive efficient Prolog programs from purely logic descriptions [37]. Moreover, since the information provided by the user is certified by the system, it can be also used by a compiler to optimize the object code. Even though the same ideas may be applied to any other declarative language, it is clear that the current proposal specifically applies to Prolog, which is "de facto" the standard language of the logic programming paradigm. This makes somehow incomparable our contribution with respect to works that follow a completely different philosophy, like the ones on Mercury [61].

In order to put our contribution in perspective, we first discuss the main requirements for a unified (abstract) semantic framework.

1.1. A Complex Analysis, Based on a Number of Abstract Domains

The nature of the information useful for the various applications of logic and Prolog program analyses is nowadays well identified. Nevertheless, no previous framework was able to incorporate all kinds of information in a single analysis. Although some authors prefer to decompose a complex analysis into a series of simpler and independent ones [3], we follow the spirit of [16] where the benefits of combining domains are properly discussed. Let us summarize the information the most relevant for logic programs that is integrated in our analyser.

- *Determinacy and cardinality* information models the number of solutions to a procedure and is useful for optimizations, like dead code elimination, and automatic complexity analysis [29].
- *Mode* information describes the instantiation level of program variables at some program point. Groundness ("is a variable bound to a ground term?") and freeness ("is a variable either uninstantiated or an alias of other variables?") are the most interesting situations to detect since they allow for various forms of unification specialization. Groundness is also essential for ensuring a safe use of negation by failure and is instrumental for determinacy analysis. Freeness is useful to detect sure success of unification, which is required by some optimizing transformations and improves the precision of a cardinality analysis.
- *Sharing* information expresses that the terms bound to different program variables may (or may not) contain occurrences of the same (free) variable. This kind of information is needed to ensure that unification is occur-check free, and to improve the precision of mode analysis.
- *Term size information* states relationships between the size of the terms bound to different program variables. It is useful for termination analysis.
- *Type* information defines an approximation to the set of terms that can be bound to a program variable. It allows one to refine most analyses and optimizations based on modes. In a verification context, type information is inferred to ensure that procedures are correctly called and/or produce well-typed results. Type information is instrumental for term size analysis.

Mode, sharing, term size, and type information can easily be expressed within classical abstract interpretation frameworks based on the abstract substitution notion such as [10, 47, 48]. Other kinds of information cannot be expressed within classical abstract interpretation frameworks because the latter ignore important operational aspects of Prolog such as the depth-first search rule and the difference between failure and non termination. Thus, for instance, information about determinacy and termination is in general derived within specific frameworks more directly based on the operational semantics of Prolog. Nevertheless, such analyses may benefit from mode, term size, and type information and thus often assume that a preliminary analysis based on abstract interpretation has been performed.

¿From the previous discussion it should be clear that a complete analyser of Prolog programs should be based on an integrated framework. This is precisely what we propose in this paper.

1.2. Contribution of this Paper.

The main contributions of this paper can be summarized as follows.

1. We introduce a novel notion of abstract sequence which models sets of pairs of the form $\langle \theta, S \rangle$, where θ and S denote an (input) substitution and the

sequence of answer substitutions resulting from executing a clause, a goal, or a procedure with this input. Abstract sequences make it possible to relate the number of solutions and the size of output terms to the size of input terms in full generality. For instance, we can relate the input and ouput sizes of the *same* term (i.e., bound to the same program variable) without requiring any invariance under instantiation. To the best of our knowledge, such generality was not available in previous frameworks for term size analysis.

2. We provide a complete description of an analyser of Prolog procedures which integrates all previously mentioned analyses in a single, more powerful, one. The analyser does not perform a fixpoint computation but instead it verifies the correctness of the program with respect to a set of abstract descriptions provided by the user. Such descriptions are called *behaviours* and consist of abstract sequences and size expressions which must strictly decrease through recursive calls (the analyser only accepts terminating procedures). For the sake of simplifying the presentation, we only consider a single built-in operation, namely unification, and we do not treat the cut nor the negation. We explain in the conclusion of the paper how to overcame these simplifications.

3. We describe a generic domain of abstract sequences whose elements have the form $\langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$, where β_{in} describes a set of input substitutions, β_{ref} is a refinement of β_{in} modelling the input substitutions leading to a successful execution, β_{out} describes the set of output substitutions, E_{ref_out} is a set of constraints between the size of the terms in β_{ref} and the size of the terms in β_{out}, E_{sol} is a set of constraints between the size of terms in β_{ref} and the number of solution. The introduction of β_{ref} allows us to improve the accuracy of E_{ref_out} and E_{sol}, as constraints have only to deal with successful executions. Note that abstract sequences can be seen as a way to abstract a trace-based semantics into relations, in the spirit of [22].

4. We instantiate the generic domain by fixing a particular domain of abstract substitutions (for the β's) and a particular domain of constraints (for the E's). The domain of abstract substitution is an improvement of the domain `Pattern` [47, 55] extended with a type component. The domain of constraints consists of sets of integer linear equalities and inequalities. Notice that we only consider integer linear expressions for size expressions. Based on this instantiation, we describe the (high level) implementation of the main abstract operations.

1.3. Plan of the Paper.

The rest of the paper is organized as follows: Section 2 provides an overview of the functionalities of the analyser on a simple example. Section 3 contains a complete description of our domain of abstract sequences. The analyser is described in Section 4. Section 5 details the implementation of two main operations of the analyser in the context of the chosen abstract domain. Section 6 discusses related works. Section 7 concludes.

2. INFORMAL OVERVIEW: A SAMPLE ANALYSIS

2.1. Specification of Operational Properties of a Procedure

Consider the procedure `select/3` of which both the usual Prolog code and its normalized version are depicted in Figure 2.1; the latter is the one to which the analyser actually applies and is annotated by its program points for the sake of the presentation. Declaratively, the procedure `select/3` defines a relation select(X, L,

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

$\texttt{select(X, L, LS):-}_1\ \texttt{L=[H|T],}_2\ \texttt{H=X,}_3\ \texttt{LS=T,}_4\ \texttt{list(T)}_5\ \texttt{.}_6$
$\texttt{select(X, L, LS):-}_7\ \texttt{L=[H|T],}_8\ \texttt{LS=[H|TS],}_9\ \texttt{select(X, T, TS)}_{10}\ \texttt{.}_{11}$

Figure 2.1. The procedure `select/3` and its (annotated) normalized version

LS), between three terms, that holds if and only if the terms L and LS are lists and LS is obtained by removing one occurrence of X from L. Note that, declaratively, the type checking literal `list(T)` is needed to express that the relation does not hold if L and LS are not lists. Our analyser checks a number of operational properties which ensure that Prolog actually computes the specified relation, assuming that the procedure is "declaratively" correct. In fact, it is not the case that the procedure is correct for all possible calls. So, we restrict our attention to one particular and reasonable class of calls, i.e., calls such that X and LS are *distinct* variables and L is any ground term (not necessarily a list).[1] For this class of calls, the user has to provide a description of the expected behaviour of the procedure by means of the formal specification depicted in Figure 2.2. In order to explain the

```
select(in(X:var, L:ground, LS:var),
       ref(_, [_|list],_),
       out(ground,_,ground list),
       srel(L_ref = LS_out + 1, sol = L_ref),
       sexpr(L))
list(in(L: ground), ref(list), srel(sol = 1), sexpr(L))
```

Figure 2.2. Specifications for `select/3` and `list/1`

meaning of such a specification, we view the (concrete) semantics of the procedure `select/3` as a (total) function that maps every (input) substitution θ such that $dom(\theta) = \{\texttt{X}, \texttt{L}, \texttt{LS}\}$ to a *sequence* S of (output) substitutions over the same domain. According to this viewpoint, the formal specification describes 1) the set of all input substitutions θ considered acceptable (i.e., the class of calls to be analysed) and 2) (an over-approximation of) the set of all pairs $\langle\theta, S\rangle$ such that θ is an acceptable input substitution and S is the corresponding sequence of output substitutions.

[1] Operationally, the literal `list(T)` could be removed if we further restrict the class of calls by requiring that L is a ground *list*. This fact can be deduced automatically by our analyser. However, for the sake of demonstrating the functionalities of the analyser, it is interesting to consider a more general class of calls where L is *any* ground *term*.

The `in` part of the formal specification of select/3 states that the acceptable input substitutions θ are exactly those such that $\mathtt{X}\theta$ and $\mathtt{LS}\theta$[2] are distinct variables and $\mathtt{L}\theta$ is any ground term. The fact that X and LS are distinct is expressed by the absence of any possible sharing information in the `in` part.

The `ref` part of the specification is a *refinement* of the `in` part; it gives properties shared by all acceptable input substitutions θ that lead to at least one result, i.e., such that S has at least one element. In this case, the `ref` part indicates that the execution succeeds at least once only if L is a non empty list. The information provided by this part is essential both to simplify the analysis of a procedure and to improve its precision: we can treat separately executions that fails and thus give more precise information between the input and output substitutions for executions that succeed.[3] Occurrences of the symbol "_" in this part of the specification means that the information about the corresponding argument cannot be refined with respect to the `in` part. More generally, the user is allowed to omit from the specification all pieces of information which can be inferred from another part.

The `out` part of the specification provides information about output substitutions (i.e., the elements of S). In this case, it indicates that X will become a ground term and that LS will become a ground list.

The `srel` and `sexpr` parts of the specification are useful to prove termination and to predict the number of solutions to a call. The meaning of these parts presupposes the notion of size of a term. In this paper, we assume that it is given by the *list-length* norm, which is defined by $\|[t_1|t_2]\| = 1 + \|t_2\|$ and $\|t\| = 0$ if t is not of the form $[t_1|t_2]$.[4]

Based on this norm, the `sexpr` part of the specification describes a positive integer linear function of the input terms sizes, which must decrease through recursive calls. In this case, it is just the size of L. This information is used to prove that the execution terminates for all calls described by the `in` part. Moreover, the `srel` part of the specification describes a relation between the sizes of input terms and the sizes of output terms and a relation between the sizes of input terms and the number of solutions to the call. In this case, it says that the input size of L is always equal to the output size of LS plus 1 and that the number of solutions (i.e., the length of S) is equal to the input size of L. Two points are worth to be clarified here. First, we can see that the `ref` part allows us to state precise information about the number of solutions. (If L is a ground term but not a list, the number of solutions is 0. Thus without the refined information about successful inputs, we could only state `0<=sol<=L_in`, since we only consider linear (in)equations between the sizes of terms.) Second, let us stress that the `srel` part does not describe a so-called interargument relation (as, e.g., in [30]) but a relation between the sizes of input and output terms. In this example, both approaches are equivalent since L is initially ground. In general however, our approach is more powerful because we do not need to restrict to *rigid* terms (i.e., whose size is invariant under instantiation)

[2]To simplify the notations, we abusively denote $\mathtt{X}\theta$, $\mathtt{L}\theta$, and $\mathtt{LS}\theta$ by X, L and LS.

[3]In fact, the `ref` part does not always describe exactly the set of inputs that succeed, since this would require to solve an undecidable problem. But it does when the `srel` part ensures that there is at least one solution, which is the case for `select/3`.

[4]This choice is rather restrictive and related to the fact that we concentrate on lists manipulating programs. Nevertheless, the presentation of the analyser in the rest of the paper is largely generic. Thus, it will become clear later on that more general size notions can be used.

as we differentiate the input and output sizes of the terms. However our approach is also computationally more expensive since it potentially doubles the number of variables in the (in)equations.

2.2. Abstract Sequences

Technically, the first five parts of a specification define a mathematical object called an *abstract sequence*. The semantics of abstract sequences is defined in Section 3.3.2. In order to give an informal overview of our analyser, we present the abstract sequences corresponding to the specifications of `select/3` and `list/1`, as they are printed out by the analyser, in Figure 2.3. Abstract sequences contain the same

```
============================= B_select =============================
beta_in: sv = {X->1,L->2,LS->3}; frm = {}
         mo = {1->var,2->ground,3->var}
         ty = {1->anylist,2->any,3->anylist}
         ps = {(1,1),(3,3)}
beta_ref: sv = {X->1,L->2,LS->3}; frm = {2->[4|5]}
          mo = {1->var,2->ground,3->var,4->ground,5->ground}
          ty = {1->anylist,2->list,3->anylist,4->any,5->list}
          ps = {(1,1),(3,3)}
beta_out: sv = {X->1,L->2,LS->3}; frm = {2->[4|5]}
          mo = {1->ground,2->ground,3->ground,4->ground,5->ground}
          ty = {1->any,2->list,3->list,4->any,5->list}
          ps = {}
in_ref = {1->1,2->2,3->3,4->4,5->5}
in_out = {1->6,2->7,3->8,4->9,5->10}
E_ref_out = {sz(8)=sz(5)}
E_sol = {sol=sz(5)+1}
============================== B_list ==============================
beta_in:                  beta_ref:                 beta_out:
 sv = {L->1}               sv = {L->1}               sv = {L->1}
 frm = {}                  frm = {}                  frm = {}
 mo = {1->ground}          mo = {1->ground}          mo = {1->ground}
 ty = {1->any}             ty = {1->list}            ty = {1->list}
 ps = {}                   ps = {}                   ps = {}
E_ref_out = {}
E_sol = {sol=1}
```

Figure 2.3. Abstract sequences for `select/3` and `list/1`

information as the corresponding specifications but the information is expressed in a form better suited for defining and implementing abstract operations. We use abstract substitutions from the generic domain $\mathtt{Pat}(\Re)$ [16, 47] instantiated to mode, type, and possible sharing information. In this abstract domain, the information is expressed on *indices*, not directly on the procedure variables. Indices represent terms bound to the program variables or subterms of those terms. For instance, the abstract substitution `beta_ref` of the abstract sequence `B_select` characterizes a set of substitutions θ as follows: the `sv` component binds the program variables `X`, `L`, and `LS` to the indices `1`,`2`, and `3`, which represent the terms t_1, t_2, and t_3 respectively bound to `X`, `L`, and `LS` in θ. The `frm` component states that the term t_2 is of the form $[t_4|t_5]$. The `mo(de)` component states that t_1 and t_3 are variables and that t_2, t_4, and t_5 are ground terms. The `ty(pe)` component provides information about the types of terms. In this paper, we treat types in a rather simplified way as we consider only three types, namely `list` (lists), `anylist` (all terms which can be instantiated to a list), and `any` (all terms). This restricted "type system" is sufficient to deal with simple lists manipulating programs.The terms t_1 and t_3 have type `anylist` because they are variables. The `ps` component consists of the pairs of indices of terms that may share a variable. Since the pair `(1,3)` does not belong to `ps`, t_1 and t_3 are distinct variables. Let us now turn to the components `E_ref_out` and `E_sol` of the abstract sequence. The first one relates the sizes of terms in an input substitution to the sizes of terms in the corresponding output substitutions. Terms are represented by indices, but as the same indices can be used in `beta_ref` and `beta_out`, we express the relation on the "disjoint union" of the two sets. The functions `in_ref` and `in_out` maps the original indices to their image in the disjoint union. Many equality constraints between terms can be derived from the components `mo` and `frm` of `beta_ref` and `beta_out` and from the correspondence between the indices. Those constraints are not represented in `E_ref_out`. Only "essential" constraints are represented. In this case, it is sufficient to state that `sz(8)=sz(5)` which means that the output size of LS is equal to the input size of the tail of L. Finally, the component `E_sol` defines the constraints on the number of solutions. As for the previous component we choose to express the constraints in terms of the "elementary" indices, whose principal functor is unknown.

2.3. Description of a Successful Analysis

The analysis of the procedure `select/3` according to the above specification works on the normalized version of the procedure given in the second part of Figure 2.1. We have annotated the procedure with natural numbers identifying its "program points." The first work of the analyser precisely consists of attaching an abstract sequence `B_`i to every program point $_i$. We now provide a trace of the execution.[5] To understand the trace, it is worth pointing out that every abstract sequence `B_`i describes (possibly an over-approximation of) the set of pairs $\langle\theta, S\rangle$ such that θ is described by `beta_in` and S is the set of output substitutions produced by the literals of the clause *before* the program point $_i$. The analysis ignores the next literals in the clause and is thus compositional (contrary to SLD-resolution). The first abstract sequence is the following:

```
============================== B_1 =================================
beta_ref: sv = {X->1,L->2,LS->3}; frm = {}
          mo = {1->var,2->ground,3->var}
          ty = {1->anylist,2->any,3->anylist}
```

[5]For space reasons, only essential changes are depicted. The hidden parts of every abstract sequence are thus identical to those of the abstract sequence relative to the preceding program point. For instance, no abstract substitution `beta_in` is depicted, since it is the same in all abstract sequences.

```
          ps = {(1,1),(3,3)}
beta_out: sv= {X->1,L->2,LS->3,H->4,T->5}; frm = {}
          mo = {1->var,2->ground,3->var,4->var,5->var}
          ty = {1->anylist,2->any,3->anylist,4->anylist,5->anylist}
          ps = {(1,1),(3,3),(4,4),(5,5)}
E_ref_out = {}
E_sol = {sol=1}
```

The abstract substitution `beta_ref` is identical to `beta_in` because the head of the clause is unifiable with any call since it contains distinct variables. Similarly, `beta_out` is obtained by extending `beta_in` with information about the local variables `H` and `T`. Since they are brand-new, their mode, type, and sharing information is obviously obtained. All size constraints between terms can be inferred by establishing a correspondence between the indices of `beta_ref` and those of `beta_out`, thus the component `E_ref_out` is empty since we only depict "essential" constraints. Finally, the component `E_sol` expresses that the unification of the head of the clause succeeds exactly once.

The first three literals in the clause are unifications. They result in the following abstract sequences:

```
=============================== B_2 =================================
beta_ref: frm = {2->[4|5]}
          mo = {...,4->ground,5->ground}
          ty = {...,4->any,5->any}
beta_out: frm = {2->[4|5]}
          mo = {...,4->ground,5->ground}
          ty = {...,4->any,5->any}
=============================== B_3 =================================
beta_out: sv= {X->1,...,H->1,T->4}; frm = {2->[1|4]}
          mo = {1->ground,...,4->ground}
          ty = {1->any,...,4->any}
          ps = {(3,3)}
=============================== B_4 =================================
beta_out: sv= {...,LS->3,...,T->3}; frm = {2->[1|3]}
          mo = {...,3->ground}
          ty = {...,3->any}
          ps = {}
```

The first unification `L=[H|T]` succeeds if and only if L is a ground term of the form $[t|t']$ because H and T are distinct variables as specified by the components `mo` and `ps` of `B_1`. Thus, in `B_2`, the analyser updates the `frm` component of `beta_ref` and `beta_out` with the structural information about L. Importantly, the component `E_sol` is *not* modified because unification succeeds for *all* terms L of the form $[t|t']$. The next two unifications `H=X` and `LS=T` both surely succeed because X and LS are distinct variables as indicated by the components `mo` and `ps` of `B_2`. The result of the unification is recorded by mapping corresponding variables to the same index, in the `sv` component.

The literal `list(T)` is then analysed by means of the abstract sequence `B_list` (see Figure 2.3). Since T is now a ground term, the call is compatible with the component `beta_in` of the abstract sequence. Thus, the analyser infers that the call succeeds if and only if T is a list, and that it succeeds exactly once. This information is recorded in B_5 as follows:

```
============================== B_5 =================================
beta_ref: ty = {...,2->list,...,5->list}
beta_out: ty = {...,2->list,3->list}
```

The final abstract sequence for the first clause is obtained by removing the local variables from the sv component. The abstract sequence B_6 is thus:

```
============================== B_6 =================================
beta_ref: sv = {X->1,L->2,LS->3}; frm = {2->[4|5]}
          mo = {1->var,2->ground,3->var,4->ground,5->ground}
          ty = {1->anylist,2->list,3->anylist,4->any,5->list}
          ps = {(1,1),(3,3)}
beta_out: sv= {X->1,L->2,LS->3}; frm = {2->[1|3]}
          mo = {1->ground,2->ground,3->ground}
          ty = {1->any,2->list,3->list}
          ps = {}
E_ref_out = {}
E_sol = {sol=1}
```

We now consider the second clause. Since the treatment of the first two unifications is similar to the treatment of unifications in the first clause, we directly provide the abstract sequence B_9 corresponding to the program point just before the recursive call:

```
============================== B_9 =================================
beta_ref: sv = {X->1,L->2,LS->3}; frm = {2->[4|5]}
          mo = {1->var,2->ground,3->var,4->ground,5->ground}
          ty = {1->anylist,2->any,3->anylist,4->any,5->any}
          ps = {(1,1),(3,3)}
beta_out: sv= {X->1,L->2,LS->3,H->4,T->5,TS->6}
          frm = {2->[4|5],3->[4|6]}
          mo = {1->var,2->ground,3->ngv,4->ground,5->ground,6->var}
          ty = {1->anylist,2->any,3->anylist,4->any,5->any,
                6->anylist}
          ps = {(1,1),(3,3),(6,6)}
E_ref_out = {}
E_sol = {sol=1}
```

Since we want to prove termination of the procedure, the analyser first checks that the size expression (provided by the sexpr part of the specification) is smaller for the recursive call than for the initial call, i.e., that the size of T is smaller than the initial size of L. This can be deduced from the "implicit" constraints of E_ref_out obtained by mapping the indices of beta_ref to those of beta_out and by reasoning on the structural information and the modes. Next, the analyser checks that the information given by beta_out about the actual parameters X, T, and TS is compatible with the component beta_in of B_select, which is the case, since X and TS are distinct variables and T is a ground term. Thus, the analyser may use the information from B_select to update B_9. The following abstract sequence is obtained:

```
=============================== B_10 ===============================
beta_ref: frm = {...,5->[6|7]}
          mo = {...,6->ground,7->ground}
          ty = {...,2->list,...,5->list,6->any,7->list}
beta_out: frm = {...,5->[7|8]}
          mo = {1->ground,...,3->ground,...,6->ground,7->ground,
                8->ground}
          ty = {1->any,2->list,3->list,...,5->list,6->list,7->any,
                8->list}
          ps = {}
in_ref = {1->1,2->2,3->3,4->4,5->5,6->6,7->7}
in_out = {1->8,2->9,3->10,4->11,5->12,6->13,7->14,8->15}
E_ref_out = {sz(13)=sz(7)}
E_sol = {sol=sz(7)+1}
```

It is intuitively clear that all the information contained in `B_10` can be deduced by mapping the indices of the components of `B_9` to those of `B_select` and reexpressing the information in `B_select` on the indices of `B_9`. Technically, this is done by means of operations called *constraint mappings*, which are described in Section 5.2.1. The final abstract sequence for the second clause is obtained by removing the local variables from the component `sv`:

```
=============================== B_11 ===============================
beta_ref: sv = {X->1,L->2,LS->3}; frm = {2->[4|5],5->[6|7]}
          mo = {1->var,2->ground,3->var,4->ground,5->ground,
                6->ground,7->ground}
          ty = {1->anylist,2->list,3->anylist,4->any,5->list,6->any,
                7->list}
          ps = {(1,1),(3,3)}
beta_out: sv= {X->1,L->2,LS->3}; frm = {2->[4|5],3->[4|6],5->[7|8]}
          mo = {1->ground,2->ground,3->ground,4->ground,5->ground,
                6->ground,7->ground,8->ground}
          ty = {1->any,2->list,3->list,4->any,5->list,6->list,7->any,
                8->list}
          ps = {}
in_ref = {1->1,2->2,3->3,4->4,5->5,6->6,7->7}
in_out = {1->8,2->9,3->10,4->11,5->12,6->13,7->14,8->15}
E_ref_out = {sz(13)=sz(7)}
E_sol = {sol=sz(7)+1}
```

The next task of the analyser is to combine the abstract sequences `B_6` and `B_11` to compute an abstract sequence `B_final` describing the global behaviour of the procedure.[6] Its components `beta_ref` and `beta_out` are computed from those of `B_6` and `B_11` by a least upper bound operation (which is classical for this kind of abstract substitutions, see [16, 47]).

The final component `E_ref_out` is computed in two steps: first, the (in)equations of the components `E_ref_out` of `B_6` and `B_11` are reexpressed in terms of the indices

[6] It happens that the abstract sequence `B_final` is identical to `B_select` in this case. Thus the reader should look at Figure 2.3 to understand the next steps of the discussion.

of `B_final`; second, the least upper bound (i.e., geometrically, the convex hull) of the two sets of (in)equations is computed. In the first step, both "implicit" and "essential" (in)equations of `B_6` and `B_11` must be taken into account because part of the structural information contained in `B_6` and `B_11` is removed from `B_final`. As a consequence, previously "implicit" equations can become "essential" in `B_final`. For instance, we obtain two "essential" equations from `B_6`: `sz(4) = sz(6)` and `sz(5) = sz(8)`. These equations express that the final value of X (i.e., t_6) and LS (i.e., t_8) have the same size as the first element (i.e., t_4) and the tail (i.e., t_5) of L. The information that we actually have $t_4 = t_6$ and $t_5 = t_8$ is lost due to the weaker structural information of `B_final`. From the abstract sequence `B_11`, we obtain the "essential" (in)equations `sz(5) = sz(8)` and `sz(5) >= 1`. In the second step, it is more efficient to compute the least upper bound on the "essential" (in)equations only, since the convex hull is a computationally expensive operation. In this case, we obtain a single equation: `sz(5) = sz(8)`.

The final component `E_sol` is also an upper bound of two systems of linear (in)equations reexpressed on the indices of the component `beta_ref` of `B_final`. The first system corresponds to the case where both clauses succeed, i.e., the case where L contains at least two elements. Introducing two new symbols to denote the number of solutions of the two clauses, we obtain the system:

```
sol = sol1+sol2,   sol1 = 1,   sol2 = sz(5),   sz(5) >= 1.
```

Eliminating `sol1` and `sol2`, the system reduces to

```
sol = sz(5)+1,   sz(5) >= 1.
```

The second system corresponds to the case where only the first clause succeeds. It can be deduced by comparing the components `beta_ref` of `B_6` and `B_11` that this is possible only if L consists of a single element. The corresponding system is:

```
sol = 1,   sz(5) = 0.
```

Obviously, the convex hull of the two systems is given by the single equation `sol = sz(5)+1` (implicitly, all sizes are greater or equal to 0).

The very last step of the analysis consists of verifying that the information contained in the abstract sequence `B_final` implies (is at least as precise as) the information contained in the formal specification (i.e., in `B_final`). In this case, the verification is immediate since the two are equal.

2.4. An Unsuccessful Analysis

It should be clear from the previous explanations that all the information given in the specifications and recorded in the abstract sequences is essential and must be fully exploited to obtain a successful analysis. For instance, let us remove the condition that X and LS initially are distinct variables. This should be expressed by adding the information `ps:(X,LS)` in the `in` part of the specification. The component `ps` of `beta_in` thus becomes {`(1,1),(1,3),(3,3)`}. In the first clause, the unification `H=X` still surely succeeds but since the indices `1` and `3` "may share", it gives the mode `gv` (ground or variable) to the index `3`. Now, since LS is possibly ground, the system is unable to prove that the unification `LS=T` surely succeeds. The component `E_sol` of `B_4` is thus {`sol <= 1`} instead of {`sol = 1`}. As a consequence, the analyser is globally unable to prove that the number of solutions is equal to the size of L. It is however possible to obtain a successful analysis by relaxing the `srel` part of the specification to

```
srel(L_ref = LS_out + 1, sol <= L_ref).
```

3. ABSTRACT DOMAINS

In this section, we present a simplified description of the abstract domains used by our analyser (a more complete presentation can be found in [43]). Section 3.2 describes our domain of abstract substitutions. This part is classical. Section 3.3 is novel: it presents our domain of abstract sequences. Finally, Section 3.4 defines the notion of behaviour, which formalizes the notion of formal specification introduced in Section 2, i.e., the full package of information provided (for verification) by the user to the system.

3.1. Preliminaries

The reader is assumed to be familiar with the basic concepts of logic programming and abstract interpretation (see [21, 51]).

Terms, Indices and Norms. We denote by $\mathcal{T}$ the set of all terms, and by I (possibly subscripted or superscripted) a set of indices; in particular, we assume that I is a finite subset of $\mathbf{N}$. $\mathcal{T}^I$ is the set of all tuples of terms $\langle t_i \rangle_{i \in I}$ and $\mathcal{T}^*_I$ is the set of all "frames" of the form $f(i_1, \ldots, i_n)$ where f is a functor of arity n and $i_1, \ldots, i_n \in I$. A size measure, or norm, is a function $\| \cdot \| : \mathcal{T} \to \mathbf{N}$, see [8, 26, 65]. In this paper, we always refer to the list-length measure presented in Section 2.

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

Substitution Sequences. A *program substitution sequence* S is a *finite* sequence $< \theta_1, \ldots, \theta_n >$ $(n \geq 0)$ where the θ_i are program substitutions with the same domain D. D is also the domain of S, denoted by $dom(S)$. We denote by $< >$ the empty sequence. $Subst(S)$ is the set of all substitutions which are elements of S. $SSeq$ is the set of all program substitution sequences. The *restriction* of S to $D \subseteq dom(S)$, denoted by $S_{/D}$, is the sequence obtained by restricting each $\theta \in Subst(S)$ to D. The symbol :: denotes sequence concatenation.

3.2. Abstract Substitutions

The domain of abstract substitutions we consider is a simple extension (with type information) of the domain `Pattern` presented in [47]. It can be viewed as an instantiation to modes, types and possible sharing of the generic abstract domain `Pat`($\Re$) described in [15, 16].

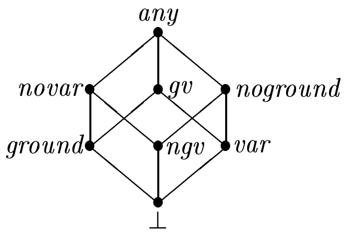


Figure 3.1. Modes

3.2.1. Modes. We consider the set of modes $Modes = \{\perp, ground, var, ngv, novar, gv, noground, any\}$, satisfying the ordering relationship implied by the diagram depicted in Figure 3.1, where an arc between M_1 and M_2 with M_1 above M_2 means that $M_1 > M_2$. The semantics of modes can be given by the following concretization function:

$$\begin{array}{lcl} Cc(\perp) & = & \emptyset; \\ Cc(ground) & = & \{t \mid t \text{ is a ground term}\}; \\ Cc(var) & = & \{t \mid t \text{ is a variable}\}; \\ Cc(ngv) & = & \{t \mid t \text{ is neither a variable nor a ground term}\}; \\ Cc(lub(M_1, M_2)) & = & Cc(M_1) \cup Cc(M_2). \end{array}$$

For any set of indices I, we denote by $Modes_I$ the set of all functions from I to $Modes$ augmented with $\perp$. The semantics of an element $mo \in Modes_I$ is given by the following concretization function Cc. If $mo = \perp$ then $Cc(mo) = \emptyset$, otherwise $Cc(mo)$ is the set $\{\langle t_i \rangle_{i \in I} \in \mathcal{T}^I \mid \forall i \in I : t_i \in Cc(mo(i))\}$.

3.2.2. Types. A simple type domain for lists is considered: $Types = \{\perp, list, anylist, any\}$, ordered by: $\perp \leq list \leq anylist \leq any$. The semantics of types is as follows:

$$\begin{array}{lcl} Cc(\perp) & = & \emptyset; \\ Cc(list) & = & \{t \mid t \text{ is a list}\}; \\ Cc(anylist) & = & \{t \mid t \text{ is a term that can be instantiated to a list}\}; \\ Cc(any) & = & \{t \mid t \text{ is any term}\}. \end{array}$$

For any set of indices I, we denote by $Types_I$ the set of all functions from I to $Types$ augmented with $\perp$. The semantics of an element $ty \in Types_I$ is given by the following concretization function Cc. If $ty = \perp$ then $Cc(ty) = \emptyset$, otherwise $Cc(ty)$ is the set $\{\langle t_i \rangle_{i \in I} \in \mathcal{T}^I \mid \forall i \in I : t_i \in Cc(ty(i))\}$.

3.2.3. PSharing. This domain [62] specifies possible variable sharing between terms. For any set of indices I, we denote by $PSharing_I$ the set of all binary and symmetrical relations $ps \subseteq I \times I$ augmented with $\perp$. The semantics of an element $ps \in PSharing_I$ is given by the following concretization function. If $ps = \perp$ then $Cc(ps) = \emptyset$, otherwise $Cc(ps)$ is the set $\{\langle t_i \rangle_{i \in I} \in \mathcal{T}^I \mid \forall i, j \in I : \; Var(t_i) \cap Var(t_j) \neq \emptyset \Rightarrow (i, j) \in ps\}$.

3.2.4. Abstract Tuples. The component of abstract substitutions that gives information about the modes, types and possible sharing of the terms is called the *abstract tuple.*

Definition 3.1. [Abstract Tuple] An *abstract tuple* α over a set of indices I is either $\perp$ or a triplet of the form $\langle mo, ty, ps\rangle$ where $mo \in Modes_I$, $ty \in Types_I$ and $ps \in PSharing_I$, with $mo, ty, ps \neq \perp$ and for all $i \in I$, $mo(i), ty(i) \neq \perp$.

Definition 3.2. [Semantics of an Abstract Tuple] The *semantics of an abstract tuple* α over I is given by the following concretization function. If $\alpha = \perp$ then $Cc(\alpha) = \emptyset$, otherwise $Cc(\alpha) = Cc(mo) \cap Cc(ty) \cap Cc(ps)$.

3.2.5. Abstract Substitutions. We are now in position to introduce the notion of abstract substitution in a formal way. We first introduce a *pseudo*-version of this abstract object which is simpler and easier to manipulate. The corresponding (*strict*) version is endowed with further conditions to prevent from incorrect and redundant representation. The distinction between pseudo-objects and strict-objects is useful because in many cases it is more convenient to work with "imperfect" descriptions which are easier to compute. A normalization operation (preserving the semantics) allows us to compute a strict object from a pseudo-object. Strict objects can be seen as approximate implementations of the reduced product [20] of their components.

An abstract substitution β over variables $X_1, \ldots, X_n$ is a triplet $\langle sv, frm, \alpha\rangle$ where sv is a function from $\{X_1, \ldots, X_n\}$ to a set of indices I, frm is a partial function from I to $\mathcal{T}_I^*$, and α describes properties concerning modes, types and possible sharing of some terms. It represents a set of program substitutions of the form $\{X_1/t_1, \ldots, X_n/t_n\}$. The main idea behind this abstract domain is that an abstract substitution β can provide information not only about terms $t_1, \ldots, t_n$ but also about subterms of them. If t_i is a term of the form $f(t_{i_1}, \ldots, t_{i_m})$, then β is expected to represent information relative to $t_{i_1}, \ldots, t_{i_m}$. Each term described in β is denoted by the corresponding index.

Let us describe the three components of $\beta = \langle sv, frm, \alpha\rangle$. The *same-value* component sv is responsible for mapping each variable X_j to the index i corresponding to the term t_i. In particular, it may express equality constraints between two variables X_i and X_j, when $sv(X_i) = sv(X_j)$. The *frame* (or *pattern*) component frm is a partial function that provides information relative to the structure of terms. The value of $frm(i)$, when it is defined, is equal to a term of the form $f(i_1, \ldots, i_n)$, meaning that t_i is of the form $f(t_{i_1}, \ldots, t_{i_n})$. Finally, the *abstract tuple* α provides information about modes, types and possible sharing of the terms t_i's. It is defined in terms of the elementary domains *Modes*, *Types* and *PSharing* described above.

Definition 3.3. [Pseudo-Abstract Substitution] A *pseudo-abstract substitution* β over a set of indices I is either $\perp$ or a triplet of the form $\langle sv, frm, \alpha\rangle$ where the *same-value* component sv is a function, $sv : \{X_1, \ldots, X_n\} \to I$; the *frame* component frm is a partial function, $frm : I \not\rightarrow \mathcal{T}_I^*$ (we denote the fact that no frame is associated with i by $frm(i) = undef$); and α is an *abstract tuple* over I.

The set of variables $\{X_1, \ldots, X_n\}$ is called the domain of β and is denoted by $dom(\beta)$.

Definition 3.4. [Semantics of a Pseudo-Abstract Substitution] The *semantics of a pseudo-abstract substitution* β over I is given by the following concretization function Cc. If $\beta = \perp$ then $Cc(\beta) = \emptyset$, otherwise

$$\begin{array}{rcl} Cc(\beta) & = & \{\theta \mid dom(\theta) = dom(\beta) \text{ and } \exists\ \langle t_i \rangle_{i \in I} \in \mathcal{T}^I : \\ & & \quad \forall X \in dom(\beta),\ X\theta = t_{sv(X)}; \\ & & \quad \forall i \in I,\ frm(i) = f(i_1, \ldots, i_n) \Rightarrow t_i = f(t_{i_1}, \ldots, t_{i_n}); \\ & & \quad \langle t_i \rangle_{i \in I} \in Cc(\alpha)\}. \end{array}$$

Some auxiliary notation is necessary for defining (strict-) abstract substitutions.

Definition 3.5. Let I be a set of indices, $sv : \{X_1, \ldots, X_n\} \to I$ be a function and $frm : I \not\rightarrow \mathcal{T}_I^*$ be a partial function. Consider the following relation between the indices of I: $i \prec_{frm} j$ holds iff $frm(i) = f(i_1, \ldots, i_m)$ and $i_k = j$ for some $k \in \{1, \ldots, m\}$. We denote by $\prec\!\!\prec_{frm}$ the transitive closure of $\prec_{frm}$ and by $\preceq\!\!\preceq_{frm}$ the reflexive and transitive closure of $\prec_{frm}$. We say that frm is *circuit-free* iff there exists no index $i \in I$ such that $i \prec\!\!\prec_{frm} i$. An index $i \in I$ is *reachable by sv* and frm iff there exists a variable X_k $(1 \leq k \leq n)$ such that $sv(X_k) \preceq\!\!\preceq_{frm} i$.

Definition 3.6. [(Strict-) Abstract Substitution] A *(strict-) abstract substitution* β over I is a pseudo-abstract substitution $\langle sv, frm, \alpha \rangle$ over I such that $\alpha \neq \perp$; frm is circuit-free; all $i \in I$ are reachable by sv and frm; and for all $i, j \in I$ such that $frm(i) = f(i_1, \ldots, i_n)$ and $(j, i_k) \in ps$ for some $k \in \{1, \ldots, n\}$, $(j, i) \in ps$.

Example 3.1. The abstract substitution β_{ref}, which is part of the formal specification of `select/3`, given in Figure 2.3, is represented by $\beta_{ref} = \langle sv_{ref}, frm_{ref}, \alpha_{ref} \rangle$, where $\alpha_{ref} = \langle mo_{ref}, ty_{ref}, ps_{ref} \rangle$ with

$$\begin{array}{llllllll} sv_{ref}: & \mathtt{X} \mapsto 1 & frm_{ref}: & 1 \mapsto ? & mo_{ref}: & 1 \mapsto var & ty_{ref}: & 1 \mapsto anylist \\ & \mathtt{L} \mapsto 2 & & 2 \mapsto [4|5] & & 2 \mapsto ground & & 2 \mapsto list \\ & \mathtt{LS} \mapsto 3 & & 3 \mapsto ? & & 3 \mapsto var & & 3 \mapsto anylist \\ & & & 4 \mapsto ? & & 4 \mapsto ground & & 4 \mapsto any \\ & & & 5 \mapsto ? & & 5 \mapsto ground & & 5 \mapsto list \end{array}$$

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

This substitution requires L to be a non-empty (ground) list. Therefore, the structure of the term associated with the index 2 (representing L) is known: the main functor of this term is `[.|.]`. Moreover, its first subterm (associated with 4), should be ground and its second subterm (associated with 5), should be a ground list.

Given one particular substitution θ with domain $\{X_1, \ldots, X_n\}$ and represented by an abstract substitution β over I, the correspondence between indices in I and (sub)terms in $X_1\theta, \ldots, X_n\theta$ is made explicit by the function `DECOMP` defined below. This operation computes a set $\mathcal{S}$ of term tuples. Each of them is a *decomposition of θ with respect to the (pseudo-) abstract substitution β.*

Operation 3.1. $\mathtt{DECOMP}(\theta, \beta) = \mathcal{S}$

SPECIFICATION Let θ be a substitution and $\beta = \langle sv : \{X_1, \ldots, X_n\} \to I, \mathit{frm}, \alpha\rangle$ be a (pseudo) abstract substitution over I such that $\theta \in Cc(\beta)$. $\mathtt{DECOMP}(\theta, \beta)$ returns the set $\mathcal{S} \subseteq \mathcal{T}^I$ of term tuples such that for all $\langle t_i\rangle_{i\in I} \in \mathcal{S}$ the following properties hold:

- $\theta = \{X_1/t_{sv(X_1)}, \ldots, X_n/t_{sv(X_n)}\}$;
- $\forall i \in I,\ \mathit{frm}(i) = f(i_1, \ldots, i_n) \Rightarrow t_i = f(t_{i_1}, \ldots, t_{i_n})$;
- $\langle t_i\rangle_{i\in I} \in Cc(\alpha)$.

Notice that if β is a strict abstract substitution, then the set $\mathtt{DECOMP}(\theta, \beta)$ is a singleton, i.e., it contains exactly one term tuple.

3.3. Abstract Sequences

We now formalize the notion of abstract sequence introduced in Section 2.

3.3.1. Sizes. For any set of indices I, we denote by Sizes_I any set of elements endowed with a concretization function $Cc : \mathit{Sizes}_I \to \wp(\mathbf{N}^I)$. In this paper,[7] we assume Sizes_I to be the set of all systems of linear equations and inequations over $\mathbf{Exp}_I$ (the set of all linear expressions with integer coefficients on the indices of I), extended with the special symbol $\perp$. An element $se \in \mathbf{Exp}_{\{X_1,\ldots,X_m\}}$ can also be seen as a function from $\mathbf{N}^m$ to $\mathbf{N}$, as size expressions are positive. The value of $se(\langle n_1, \ldots, n_m\rangle)$ is obtained by evaluating the expression se where each X_i is replaced by n_i. Notice that any system of linear equations and inequations over $\mathbf{Exp}_I$ defines a polyhedron in a space whose dimension is the cardinality of I.

In order to distinguish indices of I, considered as variables, from integer coefficient and constants when writing elements of $\mathbf{Exp}_I$, we wrap up each element i of I into the symbol $\mathtt{sz}(i)$.

The concretization function Cc is as follows. For all $E \in \mathit{Sizes}_I$, if $E = \perp$ then $Cc(E) = \emptyset$, otherwise, $Cc(E) = \{\langle n_i\rangle_{i\in I} \in \mathbf{N}^I | \ \langle n_i\rangle_{i\in I}$ is a solution of $E\}$.

In the following, (in)equations will be written between double brackets $[\![\cdots]\!]$, meaning that they are syntactic objects, not semantic relations. If f is a function from one set of indices to another one, such that $f(i) = i'$ and $f(j) = j'$, the expression $[\![\mathtt{sz}(f(i)) = \mathtt{sz}(f(j)) + 1]\!]$ has to be read as the syntactical equation $\mathtt{sz}(i') = \mathtt{sz}(j') + 1$. As indices from different abstract substitutions can occur in these (in)equations (e.g., we use indices from β_{ref} and β_{out} to compare the size of the terms before and after the execution of a procedure), we have to introduce a notion allowing us to "merge" two sets of indices into one set, in such a way

[7] By the generality of the definition of Sizes_I, other domains representing tuples of natural numbers may also fit in the current framework (e.g., arbitrary arithmetic constraints).

that elements from both sets remain distinct (the indices that are present in both abstract substitutions should remain distinct, as they refer to different terms). Let A and B be two (possibly non disjoint) sets. The *disjoint union* of A and B is an arbitrarily chosen set, denoted by $A+B$, equipped with two injections functions in_A and in_B satisfying the following property: for any set C and for any pair of functions $f_A : A \to C$ and $f_B : B \to C$, there exists a unique function $f : A{+}B \to C$ such that $f_A = f \circ in_A$ and $f_B = f \circ in_B$ (where the symbol $\circ$ is the usual function composition). Since the function f is uniquely defined, we can express it in terms of f_A and f_B. In the following, it is denoted by $f_A + f_B$.

3.3.2. Abstract Sequences. We are now in a position to define abstract sequences in a formal way. As usual, we introduce the notion of pseudo-abstract sequence first. The symbol *sol* is used to denote a special index representing the number of substitutions belonging to the approximated sequences.

Definition 3.7. [Pseudo-Abstract Sequence] A *pseudo-abstract sequence* B is either $\perp$ or a tuple of the form $\langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol}\rangle$ where β_{in} is a pseudo-abstract substitution over I_{in}; β_{ref} is a pseudo-abstract substitution over I_{ref} with $dom(\beta_{ref}) = dom(\beta_{in})$; β_{out} is a pseudo-abstract substitution over I_{out} with $dom(\beta_{out}) \supseteq dom(\beta_{in})$; $E_{ref\_out} \in Sizes_{(I_{ref}+I_{out})}$; and $E_{sol} \in Sizes_{(I_{ref}+\{sol\})}$.

We will refer to β_{in} and β_{out} also as $input(B)$ and $output(B)$, respectively. Moreover, we define $dom_{in}(B) = dom(\beta_{in})$ and $dom_{out}(B) = dom(\beta_{out})$.

Definition 3.8. [Semantics of a Pseudo-Abstract Sequence] The *semantics of a pseudo-abstract sequence* B is given by the following concretization function: if $B = \perp$ then $Cc(B) = \emptyset$, otherwise[8]

$$\begin{aligned}
Cc(B) = \{\langle\theta, S\rangle \mid\ & \theta \in Cc(\beta_{in}), S \in SSeq, Subst(S) \subseteq Cc(\beta_{out}),\\
& (S \neq <\ > \Rightarrow \theta \in Cc(\beta_{ref})),\\
& (\theta' \in Subst(S), \langle t_i\rangle_{i\in I_{ref}} \in \mathtt{DECOMP}(\theta,\beta_{ref}), \langle s_i\rangle_{i\in I_{out}} \in \mathtt{DECOMP}(\theta',\beta_{out})\\
& \qquad \Rightarrow \langle \|t_i\|\rangle_{i\in I_{ref}} + \langle \|s_i\|\rangle_{i\in I_{out}} \in Cc(E_{ref_out})),\\
& (\langle t_i\rangle_{i\in I_{ref}} \in \mathtt{DECOMP}(\theta,\beta_{ref})\\
& \qquad \Rightarrow \langle \|t_i\|\rangle_{i\in I_{ref}} + \{sol \mapsto |S|\} \in Cc(E_{sol}))\}.
\end{aligned}$$

The first condition on $\langle\theta, S\rangle$ expresses that all the substitutions θ that are not described by β_{ref} lead to unsuccessful calls; the second and third ones ensures that the relations expressed by E_{ref_out} (between the terms of the input substitution and those of the output substitution) and by E_{sol} (between the terms of the input substitution and the number of solutions, i.e., the number of substitutions in S) are respected.

Additional conditions are introduced to avoid (at least partially) multiple representations of the same set of substitution sequences. A (strict-) abstract sequence is defined as follows.

Definition 3.9. [(Strict) Abstract Sequence] A *(strict-) abstract sequence* B is a

[8] Notice that the $+$ operator used below is the one that applies to functions, as defined in Section 3.3.1, since tuples $\langle\|t_i\|\rangle_{i\in I}$ actually are functions.

pseudo-abstract sequence $\langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$ such that $\beta_{in}, \beta_{ref}, \beta_{out}$ are abstract substitutions; $\beta_{in} \neq \bot$; $\beta_{ref} \leq \beta_{in}$[9]; for all $\theta' \in Cc(\beta_{out})$, $\exists \theta \in Cc(\beta_{ref})$ such that $\theta'_{/dom(\beta_{ref})} \leq \theta$; and, if either β_{ref} or β_{out} or E_{ref_out} or E_{sol} is equal to $\bot$, then they are all equal to $\bot$;

Example 3.2. Consider once again the abstract sequence B for `select/3` depicted in Figure 2.3, where $I_{ref} = I_{out} = \{1, 2, 3, 4, 5\}$. The component E_{ref_out} is expressed on the disjoint union $I_{ref} + I_{out} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, where the injection functions are $in_{ref} : I_{ref} \to I_{ref} + I_{out}$ and $in_{out} : I_{out} \to I_{ref} + I_{out}$, also depicted in Figure 2.3. According to the notations introduced above, it could be rewritten into $E_{ref\_out} = [\![\mathtt{sz}(5) = \mathtt{sz}(8)]\!]$ and E_{sol} could be rewritten into $E_{sol} = [\![sol = \mathtt{sz}(5){+}1]\!]$.

3.4. Behaviours

A behaviour for a procedure is a formalization of the specification of behavioural properties provided by the user.

Definition 3.10. [Behaviour] A *behaviour* Beh_p for a procedure name $p \in \mathcal{P}$ of arity n is a finite set of pairs $\{\langle B_1, se_1 \rangle, \ldots, \langle B_m, se_m \rangle\}$ where $B_1, \ldots, B_m$ are abstract sequences such that $dom_{in}(B_k) = dom_{out}(B_k) = \{X_1, \ldots, X_n\}$ $(1 \leq k \leq m)$; and $se_1, \ldots, se_m$ are positive linear expressions[10] from $\mathbf{Exp}_{\{X_1,\ldots,X_n\}}$.

Each pair of the form $\langle B_k, se_k \rangle$ will be called a behavioural pair (or, if no confusion is possible, a behaviour). The positive linear expression se is required to strictly decrease in recursive calls of the described procedure to ensure termination.

Example 3.3. Let B be the abstract sequence of Example 3.2. The behaviour for `select/3` described in Section 2 can be represented by $\{\langle B, L \rangle\}$.

4. DESCRIPTION OF THE ANALYSER

In this section, we describe the analyser, and we discuss how it executes a program at the abstract level. If the analyser succeeds, the given behaviours correctly describe the execution of the analysed program. In particular, every procedure call (allowed by these behaviours) terminates. If the analyser does not succeed, then, either the program does not terminate or is not consistent with the behaviours given by the user, or the information given in the behaviours is not sufficent for the analyser to deduce that the program is consistent and terminates.

[9] For the sake of brevity, we omit the definition of this ordering. A formal definition may be found in [47].

[10] In fact, it is possible to use more general linear expressions, possibly involving negative coefficient, and to prove that such expressions actually are positive at each procedure call. However, for simplicity, we only consider positive linear expressions in the rest of the paper.

To simplify the presentation, we assume that the program we want to analyse contains no mutually recursive procedures. Moreover, we assume that each recursive subcall occurring in the execution of a call described by some behaviour $\langle B_q, se_q\rangle$ can also be described by this behaviour. We explain how these simplifications can be removed in Section 4.3. For space reasons, we omit the correctness proof of the analyser; it can be found in [43].

4.1. Concrete Semantics

The reasoning underlying the design of our analyser is based on the intuition that a Prolog procedure is a function mapping every input substitution to a sequence of (answer) substitutions. Proving the correctness of our analyser thus requires a (concrete) semantics which formalizes this intuition (and yet is equivalent to Prolog operational semantics). In practice, we use the concrete semantics presented in [46]. It has been proven equivalent to Prolog operational semantics in [44]. Actually, the correctness proof of the analyser uses a simplified semantic characterization for terminating executions, also given in [43]. This characterization is simpler because it has only to deal with finite sequences of substitutions while the semantics in [46] has also to consider infinite and (so-called) incomplete sequences. Observe that there is no vicious circle created by assuming that the program terminates because the correctness proof of our analyser uses an induction on a well-founded relation over procedure calls; so we can always assume that the sub-calls terminate, i.e., that our simplified characterization applies.

Programs are assumed to be normalized as follows. A *normalized program* P is a non empty set of procedures pr. A procedure is a non empty sequence of clauses c. Each clause has the form $h\colon -g$ where the head h is of the form $p(X_1,\ldots,X_n)$ and p is a predicate symbol of arity n, whereas the body g is a possibly empty sequence of literals. A literal l is either a built-in of the form $X_{i_1} = X_{i_2}$, or a built-in of the form $X_{i_1} = f(X_{i_2},\ldots,X_{i_n})$ where f is a functor of arity $n-1$, or a procedure call $p(X_{i_1},\ldots,X_{i_n})$.[11] The variables occurring in a literal are all distinct; all clauses of a procedure have exactly the same head; if a clause uses m different variables, these variables are $X_1, \ldots, X_m$. We denote by $\mathcal{P}$ the set of all predicate symbols occurring in the program P. Variables used in the clauses are called *program variables* and are denoted by $X_1,\ldots,X_i,\ldots$. Observe that all programs can be rewritten into equivalent normalized programs.

The concrete semantics associates with every program P a total function from the set of pairs $\langle\theta,p\rangle$, where p is a predicate symbol occurring in P and $dom(\theta) = \{X_1,\ldots,X_n\}$, where n is the arity of p, to the set of substitution sequences. In the rest of this section, we only consider input pairs $\langle\theta,p\rangle$ such that the execution of the call $p(X_1,\ldots,X_n)\theta$ terminates and produces the (finite) sequence of answer substitutions S. This fact is denoted by $\langle\theta,p\rangle \longmapsto S$ in our concrete semantics. We use similar notations for describing the execution of a procedure pr, a clause c and a prefix of the body of a clause, denoted by $\langle g,c\rangle$.

[11] For the sake of simplicity, once again, we do not explicitly consider other built-ins such as `var` or `is`, nor negated literals, nor the cut. It is relatively straightforward to incorporate such operations to our analyser (see the conclusion).

4.2. Abstract Execution of a Prolog Program

Our analyser is based on a standard verification technique: for a given program, it analyses each procedure; for a given procedure, it analyses each clause; for a given clause, it analyses each atom. If an atom in the body of a clause is a procedure call, the analyser looks at the given behaviours to infer information about its execution. The analyser succeeds if, for each procedure and each behaviour describing this procedure, the analysis of the procedure yields results that are covered by the considered behaviour.

In this section, we describe how our analyser executes at the abstract level the clauses and the procedures of a given Prolog program. In the following, *SBeh* is a family of behaviours $SBeh = \langle Beh_p \rangle_{p \in \mathcal{P}}$ containing exactly one behaviour Beh_p for each procedure name $p \in \mathcal{P}$ (where $\mathcal{P}$ is the set of all procedure names occurring in the analysed program).

4.2.1. Specification of the Abstract Operations. This section contains the specifications of the operations used for the abstract execution of a procedure. We suggest the reader to skip it at a first reading, and to refer to it whenever one of these operations occurs in the next (sub)sections. In Section 5, the interested reader may find a detailed description of two main abstract operations in the context of the abstract domain of Section 3, namely `UNIF_VAR` and `CONC`.

- `EXTC`$(c, \beta) = B$ is an operation that extends the domain of β to the set of all variables occurring in the clause c. The result is an abstract sequence B such that $\forall \theta \in Cc(\beta) : \langle \theta, S \rangle \in Cc(B)$, where S is the sequence whose only element is the extension of the substitution θ to the set of all variables of c.

- `RESTRC`$(c, B) = B'$ is an operation that restricts the output domain of B (which is assumed to be the set of all variables occurring in the clause c) to the variables occurring in the head of c. The abstract sequence B' must satisfy $\forall \langle \theta, S \rangle \in Cc(B) : \langle \theta, S' \rangle \in Cc(B')$, where S' is the sequence obtained by restricting the substitutions of S to the variables of the head of c.

- `RESTRG`$(l, B) = \beta$ is an operation that restricts the output domain of B to (a renaming of) the variables occurring in the literal l. The result is an abstract substitution β satisfying $\forall \langle \theta, S \rangle \in Cc(B), \forall \theta' \in Subst(S) : \theta'' \in Cc(\beta)$, where θ'' is a substitution obtained from θ' in two steps: by first restricting θ' to the variables $X_{i_1}, ..., X_{i_n}$ of the litteral l and then by renaming those variables to the standard ones $(X_1, ..., X_n)$ in order to allow the execution of the procedure the litteral is a call of.

- `EXTG`$(l, B_1, B_2) = B$ is an operation computing the effect of the execution of the literal l (which is given by the abstract sequence B_2) on the abstract sequence B_1. Intuitively, the effect of the execution of the litteral l on B_1 can be computed as an instantiation by some substitution, which yields B_2 (when applied on `RESTRG`(l, B_1)). The operation `EXTG` extends the effect of the instanciation on the whole sequence B_1 (taking into account necessary renaming to avoid name clashes).

- `LOOKUP`$(\beta, p, SBeh) = (success, B_{out})$ is an operation searching Beh_p for an abstract sequence $B \in Beh_p$ whose input substitution is at least as general as

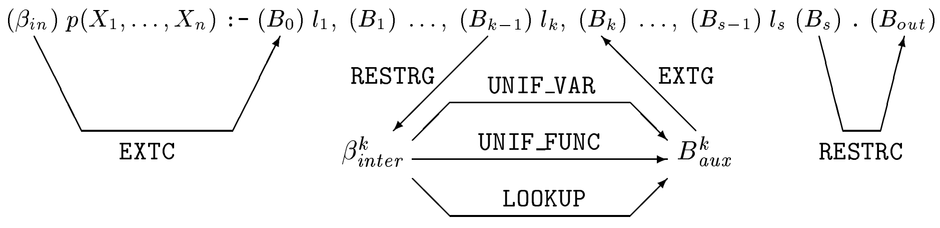
β. If such an abstract sequence exists, this operation returns $success = true$ and this abstract sequence. Otherwise, it returns $success = false$, and the value of B_{out} is undefined. The specification of `LOOKUP` can be written as $success \Rightarrow \exists se \mid \langle B, se\rangle \in Beh_p \wedge \beta \leq input(B)$.

- $\texttt{CHECK\_TERM}(l, B, se) = term$ is an operation checking if the size (according to se) of the arguments of a recursive call given by the output substitution of B is smaller than the size of the arguments of the head call. If the value $term$ is true and the literal l is $p(X_{i_1}, \ldots, X_{i_n})$, then $\forall\langle\theta, S\rangle \in Cc(B), \forall\theta' \in Subst(S)$, $se(\langle\|X_{i_1}\theta'\|, \ldots, \|X_{i_n}\theta'\|\rangle) < se(\langle\|X_1\theta\|, \ldots, \|X_n\theta\|\rangle)$.
- $\texttt{UNIF\_VAR}(\beta) = B$ executes the unification $X_1 = X_2$ on the abstract substitution β. The abstract sequence B is such that, for all $\theta \in Cc(\beta)$, and for all $\sigma \in mgu(X_1\theta, X_2\theta)$, the tuple $\langle\theta, <\theta\sigma>\rangle$ belongs to $Cc(B)$; moreover, the tuple $\langle\theta, <>\rangle$ belongs to $Cc(B)$ whenever $X_1\theta$ and $X_2\theta$ are not unifiable. An implementation of this operation will be described in Section 5.
- $\texttt{UNIF\_FUNC}(\beta, f) = B$ executes the unification $X_1 = f(X_2, \ldots, X_n)$ on the abstract substitution β, where $n - 1$ is the arity of f. Its specification is similar to the previous one.
- $\texttt{CONC}(B_1, B_2) = B$ concatenates the abstract sequences B_1 and B_2 which must have the same input abstract substitution and the same output domain. The abstract sequence B must satisfy $\forall\langle\theta, S_1\rangle \in Cc(B_1), \forall\langle\theta, S_2\rangle \in Cc(B_2)$, $\langle\theta, S_1 :: S_2\rangle \in Cc(B)$. An implementation of this operation is given in Section 5.

4.2.2. Abstract Execution of a Clause. Let

$$c \equiv p(X_1, \ldots, X_n) \texttt{ :- } l_1, \ldots, l_s \,\texttt{.}$$

be a clause of the program P and $\langle B, se\rangle$ be an element of Beh_p. Let also $\beta_{in} = input(B)$ be the input abstract substitution of B. The execution of the clause c for the input abstract substitution β_{in} may be computed as depicted below.



$$R1 : \frac{\begin{array}{c} g ::= <> \\ B' = \texttt{EXTC}(c, \beta_{in}) \end{array}}{\langle\beta_{in}, g, c\rangle \longmapsto B'} \qquad R2 : \frac{\begin{array}{c} c ::= h \colon -g \\ \langle\beta_{in}, g, c\rangle \longmapsto B' \\ B'' = \texttt{RESTRC}(c, B') \end{array}}{\langle\beta_{in}, c\rangle \longmapsto B''}$$

$$R3: \frac{\begin{array}{c} g ::= g', l \\ l ::= X_{i_1} = X_{i_2} \\ \langle \beta_{in}, g', c \rangle \longmapsto B' \\ \beta_{inter} = \mathtt{RESTRG}(l, B') \\ B_{aux} = \mathtt{UNIF\_VAR}(\beta_{inter}) \\ B'' = \mathtt{EXTG}(l, B', B_{aux}) \end{array}}{\langle \beta_{in}, g, c \rangle \longmapsto B''} \qquad R4: \frac{\begin{array}{c} g ::= g', l \\ l ::= X_{i_1} = f(X_{i_2}, \ldots, X_{i_n}) \\ \langle \beta_{in}, g', c \rangle \longmapsto B' \\ \beta_{inter} = \mathtt{RESTRG}(l, B') \\ B_{aux} = \mathtt{UNIF\_FUNC}(\beta_{inter}, f) \\ B'' = \mathtt{EXTG}(l, B', B_{aux}) \end{array}}{\langle \beta_{in}, g, c \rangle \longmapsto B''}$$

$$R5: \frac{\begin{array}{c} g ::= g', l \\ l ::= q(X_{i_1}, \ldots, X_{i_n}) \\ q \neq p, \text{ where } p \text{ is the predicate of } c \\ \langle \beta_{in}, g', c \rangle \longmapsto B' \\ \beta_{inter} = \mathtt{RESTRG}(l, B') \\ (true, B_{aux}) = \mathtt{LOOKUP}(\beta_{inter}, q, SBeh) \\ B'' = \mathtt{EXTG}(l, B', B_{aux}) \end{array}}{\langle \beta_{in}, g, c \rangle \longmapsto B''} \qquad R5': \frac{\begin{array}{c} g ::= g', l \\ l ::= p(X_{i_1}, \ldots, X_{i_n}) \\ p \text{ is the predicate of } c \\ \langle \beta_{in}, g', c \rangle \longmapsto B' \\ \beta_{inter} = \mathtt{RESTRG}(l, B') \\ \beta_{inter} \leq \beta_{in} \\ \mathtt{CHECK\_TERM}(l, B', se) = true \\ B'' = \mathtt{EXTG}(l, B', B) \end{array}}{\langle \beta_{in}, g, c \rangle \longmapsto B''}$$

Let us now briefly describe the rules depicted above.

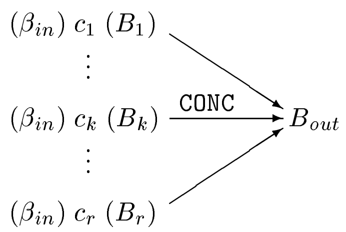
Rule $R1$ initiates the abstract execution of the clause by extending the input substitution β_{in} to the set of all variables in c. Rules $R3$, $R4$, $R5$ and $R5'$ are used for executing the litterals of the clause. Observe that, for each litteral, only one rule amongst those may apply.

First, Rule $R3$ takes care of the unifications of the type "$X_{i_1} = X_{i_2}$". In order to obtain the abstract sequence B'', associated to the program point just after the unification, from B', associated to the program point just before it, we use three abstract operations: RESTRG to obtain an abstract substitution β_{inter} whose domain is $\{X_1, X_2\}$ (computed from the abstract sequence B'); UNIF_VAR to compute the unification on β_{inter}; and EXTG to extend the effect of the unification on the whole abstract sequence B'. This last step guarantees that all the variables (in the substitution of B') whose instanciation shares a variable with the instanciation of X_{i_1} or X_{i_2} will be correctly treated. Rule $R4$ follows a very similar process to execute function unification.

Rule $R5$ and $R5'$ execute procedure calls (either non-recursive or recursive). In the case of $R5$ (non-recursive call), the effect of the procedure call is obtained by searching $SBeh$ for a description of the procedure q. In the case of recursive calls, we impose that two conditions are satisfied: first, we only allow recursive calls that can be described by the behaviour currently analysed ($\beta_{inter} \leq \beta_{in}$) and second, we require the recursive call to be strictly "smaller" (according to the size expression given in the behaviour) than the initial call (this condition is verified by CHECK_TERM). If those two assumptions hold, we simulate the execution of the recursive call by the information given in the behaviour currently analysed. If any of those tests fails, we give up the analysis as we do not possess enough information to go on safely.

Finally, Rule $R2$ completes the execution of the clause c by restricting the output substitutions described by B' to the variables occurring in the head of c.

4.2.3. Abstract Execution of a Procedure. Let $pr \equiv c_1, \dots, c_r$ be a procedure whose name is p. Its abstract execution can be summarized by the following graph and rules.



$$R6 : \frac{\begin{array}{c} pr ::= c \\ \langle \beta_{in}, c \rangle \longmapsto B' \end{array}}{\langle \beta_{in}, pr \rangle \longmapsto B'}$$

$$R7 : \frac{\begin{array}{c} pr ::= c, pr' \\ \langle \beta_{in}, c \rangle \longmapsto B' \\ \langle \beta_{in}, pr' \rangle \longmapsto B'' \\ \mathtt{CONC}(B', B'') = B''' \end{array}}{\langle \beta_{in}, pr \rangle \longmapsto B'''}$$

Rules $R6$ and $R7$ simply assert that, in order to compute the abstract execution of a whole procedure, it suffices to compute the abstract sequences given by each of its clauses and to (abstractly) concatenate those results.

In order to check that the given set of behaviours $SBeh$ correctly describes the execution of a program P, the analyser simply verify that, for each behavioural pair $\langle B, se \rangle$ attached to a procedure p, it is possible to deduce from Rules $R1$ to $R7$ that $\langle \beta_{in}, pr \rangle \longmapsto B'$, where β_{in} is the input substitution of B and pr is the text consisting of all the clauses describing the procedure p, and that the abstract sequence B' is more precise than B.

4.3. Removing the Restrictions of the Analyser

We conclude this section by explaining how the simplifying hypotheses about the form of the program can be removed. We do not discuss the treatment of additional built-ins, such as test predicates and the cut, nor the treatment of negation, since these issues are addressed in the conclusion. Here, we concentrate on how to deal with mutual recursion and with recursive calls using other behaviours than the one that is currently analysed.

Procedures with recursive subcalls that may not be described by the abstract sequence used for the input call are in fact very similar (at the abstract level) to mutually recursive procedures. Indeed, when such procedures p are decomposed into several procedures $p_1, \dots, p_s$ (with different names but - nearly - the same definition as p), each of them associated with one of the abstract sequences of Beh_p, these procedures $p_1, \dots, p_s$ are mutually recursive.

Therefore, we first explain how to treat mutual recursion and, afterwards, we explicit how to replace procedures with subcalls that cannot be described by the abstract sequence of the input call by mutually recursive procedures.

Mutual Recursion. If mutual recursion is allowed, we have to add a termination test based on the size expressions of all procedures concerned by mutual recursion (above, we only used such a test for recursive procedures). So, if p and q are mutually recursive procedures, if $\langle B_p, se_p \rangle \in Beh_p$ and if the execution of $\langle \theta, p \rangle$, where $\theta \in Cc(input(B_p))$, uses a subcall $\langle \theta', q \rangle$, where θ' can be described by $\langle B_q, se_q \rangle \in Beh_q$, we have to check (at the abstract level) that $se_q(\langle \|\theta' X_1\|, \dots, \|\theta' X_m\| \rangle) < se_p(\langle \|\theta X_1\|, \dots, \|\theta X_n\| \rangle)$, where n and m are respectively the arities of p and q. This test ensures that the mutually recursive procedures will not loop infinitely.

In order to use this method, we must analyse the program to find out all mutually recursive procedures or, more precisely, all pairs of triplets $\langle\langle p, B_p, se_p\rangle, \langle q, B_q, se_q\rangle\rangle$ (with $\langle B_p, se_p\rangle \in Beh_p$ and $\langle B_q, se_q\rangle \in Beh_q$) describing procedure calls that may use subcalls described by the other one. The termination test should be realized only when the triplets associated with the subcall and the head call are "mutually recursive".

Procedures with Subcalls that Cannot Be Described by the Abstract Sequence of the Input Call. Once the restriction about mutual recursivity has been removed, it is quite easy to allow recursive calls that cannot be described by the abstract sequence used for the head call by creating several copies of the procedure with different names (one copy for each abstract sequence given in *SBeh*) and replacing the recursive calls by calls to one of these new procedures.

More precisely, let p be the name of a procedure and $\langle B_1, se_1\rangle, \ldots, \langle B_s, se_s\rangle$ be the elements of Beh_p. In order to simplify the presentation, we assume that the definition of p contains only one recursive call. We first compute (using the abstract execution process described previously), for each (input) abstract sequence B_k, which abstract sequence B_{j_k} can be used to solve the recursive call. Afterwards, we create s procedures named $p_1, \ldots, p_s$ (we assume that these names are not used), one for each abstract sequence in Beh_p. Each procedure p_k is defined by the same text as p but the recursive call $p(X_{i_1}, \ldots, X_{i_n})$, found in the definition of p, is replaced by $p_{j_k}(X_{i_1}, \ldots, X_{i_n})$ in the definition of p_k. Then, we remove Beh_p from *SBeh* and add $Beh_{p_1}, \ldots, Beh_{p_s}$, where $Beh_{p_k} = \langle B_k, se_k\rangle$.

So, instead of analysing a single procedure where recursive calls are described by abstract sequences different from the one used as input, we analyse several (possibly mutually recursive) procedures. Once all "mutually recursive" triplets have been listed, we may be able to remove some termination tests for the (simply) recursive procedure that has been replaced and, thereby, extend the applicability of the analyser. For example, if the execution of all calls described by the triplet $t = \langle p, B, se\rangle$ leads to subcalls that may be described by $t' = \langle p, B', se'\rangle$ and if the execution of calls described by t' never uses subcalls of t, we may remove the termination test for t.

5. ABSTRACT OPERATIONS

The last step to achieve in order to obtain an implementable analyser is to provide a practical definition of all abstract operations used by the analyser. In this section we explain how we deal with a couple of operations. The same methodology can be applied to construct the whole operation set systematically. More specifically, we describe in details two main abstract operations, namely `UNIF_VAR` and `CONC`. Correctness of their implementation has been proved in [43]. Note that these implementations reuse (old) abstract operations from *GAIA* (see mainly [46, 47]). We recall the specifications of these operations but we omit their implementation.

5.1. Unification of Two Variables

The operation `UNIF_VAR` executes the built-ins $X_i = X_j$ at the abstract level. The implementation is as follows: first, we (re)use the old version of the operation, here called $\texttt{UNIF\_VAR}_{old}$, to compute an abstract substitution β'_{out} describing the result of $X_i = X_j$ called with an abstract input substitution β. Then, in order to refine β to the set of $\theta \in Cc(\beta)$ for which the unification succeeds, we establish a mapping (called structural mapping since it respects the structure of the frame component) between the indices of β and the indices of β'_{out} representing the corresponding terms. This allows us to refine the information on modes, types, and patterns provided by β, producing β'_{ref}. This is realized by operation $\mathtt{REF}_{ref}$. Finally, we derive constraints between the size of terms in β'_{ref} and β'_{out} as well as constraints on the number of solutions.

5.1.1. Structural Mapping. A structural mapping between two abstract substitutions is a mapping on the corresponding indices preserving same-value and frame.

Definition 5.1. [Structural Mapping] Let $\beta = \langle sv, frm, \alpha\rangle$ and $\beta' = \langle sv', frm', \alpha'\rangle$ be two abstract substitutions over I and I', respectively. A ***structural mapping*** between β and β' (if it exists) is a function $tr : I \to I'$ such that

- $\forall X \in dom(\beta),\ tr(sv(X)) = sv'(X)$;
- $\forall i \in I,\ frm(i) = f(i_1, \ldots, i_n) \Rightarrow frm'(tr(i)) = f(tr(i_1), \ldots, tr(i_n))$.

5.1.2. Old Operations. The operation UNIF_VAR is defined in terms of the operation $\mathtt{UNIF\_VAR}_{old}$ which is a slight generalization of the operation UNIF_VAR defined in [47]. Hereafter, we recall the specification of $\mathtt{UNIF\_VAR}_{old}$.

Operation 5.2. $\mathtt{UNIF\_VAR}_{old}(\beta) = \langle \beta', ss, sf, tr, U\rangle$

This operation unifies $X_1\theta$ and $X_2\theta$ for all $\theta \in Cc(\beta)$. We do not provide an implementation for it since it is similar to [47] whose extension is discussed in [46]. The only novelty is that we explicitly return the structural mapping tr and the set of indices U. More precisely this operation returns an abstract substitution β', two boolean values ss and sf specifying whether sure success or sure failure can be inferred at the abstract level, a structural mapping tr between β and β', and a set of indices U representing the set of terms in θ whose norm is not affected by the instantiation. The latter will allow us to establish precise constraints between the size of terms in β'_{ref} and β'_{out}.

Specification Let β be an abstract substitution over I with $dom(\beta) = \{X_1, X_2\}$. $\mathtt{UNIF\_VAR}_{old}(\beta)$ returns a pseudo abstract substitution β' over I', two boolean values ss and sf, a structural mapping $tr : I \to I'$ and $U \subseteq I$ such that:

$$\left.\begin{array}{l} \theta \in Cc(\beta) \\ \sigma \in mgu(X_1\theta, X_2\theta) \\ \langle t_i\rangle_{i\in I} \in \mathtt{DECOMP}(\theta, \beta) \\ \langle s_i\rangle_{i\in I'} \in \mathtt{DECOMP}(\theta\sigma, \beta') \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \theta\sigma \in Cc(\beta') \\ \forall i \in U,\ \|t_i\| = \|t_i\sigma\| \\ \forall i \in I,\ t_i\sigma = s_{tr(i)}; \end{array}\right.$$

$$\begin{array}{lcl} ss = true & \Rightarrow & (\forall \theta \in Cc(\beta) : X_1\theta \text{ and } X_2\theta \text{ are unifiable}); \\ sf = true & \Rightarrow & (\forall \theta \in Cc(\beta) : X_1\theta \text{ and } X_2\theta \text{ are not unifiable}). \end{array}$$

5.1.3. Refinement Operations. The operation $\mathtt{REF}_{ref}$ refines the input abstract substitution β into β'_{ref}. It is defined in terms of operations $\mathtt{REF}_{frm}$ (which focuses on the frame component) and $\mathtt{REF}_{\alpha}$ (which refines the α component). The three operations respect the same specification given below for $\mathtt{REF}_{ref}$.

Operation 5.3. $\mathtt{REF}_{ref}(\beta_1, \beta_2, tr_{1,2}) = \langle \beta', tr' \rangle$
This operation refines the abstract substitution β_1 by keeping substitutions in $Cc(\beta_1)$ that have at least an instance in $Cc(\beta_2)$.

Specification Let β_1 and β_2 be two abstract substitutions over I_1 and I_2, respectively, with $dom(\beta_1) = dom(\beta_2)$ and $tr_{1,2} : I_1 \to I_2$ be a structural mapping between β_1 and β_2. $\mathtt{REF}_{ref}(\beta_1, \beta_2, tr_{1,2})$ produces an abstract substitution β' over I' and a structural mapping $tr' : I' \to I_2$ between β' and β_2 such that $dom(\beta') = dom(\beta_k)$ $(k = 1, 2)$, $\beta' \leq \beta_1$ and

$$\left.\begin{array}{l} \theta_k \in Cc(\beta_k) \ (k = 1, 2) \\ \theta_2 \leq \theta_1 \end{array}\right\} \Rightarrow \theta_1 \in Cc(\beta').$$

The implementation of the three REF operations uses four simpler operations on modes and types that we present first. The implementation of the first one has been described in [47].

Operation 5.4. $\mathtt{EXTRM}(f, M) = \langle M_1, \ldots, M_n \rangle$
This operation computes the most precise modes of terms $t_1, \ldots, t_n$ when we know that the mode of $f(t_1, \ldots, t_n)$ is M.

Specification Let f be a function symbol of arity n and $M \in Modes$.

$$f(t_1, \ldots, t_n) \in Cc(M) \quad \Rightarrow \quad t_i \in Cc(M_i) \ (1 \leq i \leq n).$$

Operation 5.5. $\mathtt{EXTRT}(f, T) = \langle T_1, \ldots, T_n \rangle$
It is analogous to the previous one; it computes types instead of modes.

Operation 5.6. $\mathtt{UNIST}_{mo}(M) = M'$
It approximates the set of terms that can be instantiated to a term $t \in Cc(M)$.

Specification Let $M, M' \in Modes$. The following relation holds:

$$\left.\begin{array}{l} t \in Cc(M) \\ t = t'\sigma \end{array}\right\} \Rightarrow t' \in Cc(M').$$

Implementation

$$M' = \begin{array}{ll} var & \text{if } M = var \\ noground & \text{if } M \in \{ngv, noground\} \\ \perp & \text{if } M = \perp \\ any & \text{otherwise.} \end{array}$$

Operation 5.7. $\mathtt{UNIST}_{ty}(T) = T'$
It as the same specification as the previous operation where $T, T' \in \mathit{Types}$.

IMPLEMENTATION

$$\begin{array}{lcll} T' & = & \mathit{anylist} & \text{if } T \in \{\mathit{list}, \mathit{anylist}\} \\ & & \perp & \text{if } T = \perp \\ & & \mathit{any} & \text{otherwise.} \end{array}$$

Operation 5.8. $\mathtt{REF}_{\mathit{frm}}(\beta_1, \beta_2, tr_{1,2}) = \langle \beta', tr' \rangle$
It refines the abstract substitution β_1 only using the frame component of β_2.

IMPLEMENTATION Construct the sequence of intermediate abstract substitutions $\beta^0, \ldots \beta^i \ldots$ and structural mappings $tr^0, \ldots, tr^i, \ldots$ as follows.

1. $\beta^0 = \beta_1$ and $tr^0 = tr_{1,2}$.
2. Assume given β^i and the structural mapping $tr^i : I^i \to I_2$.
   Suppose that there exists $j \in I^i$ such that $mo^i(j) \leq \mathit{novar}$, $\mathit{frm}^i(j) = \mathit{undef}$ and $\mathit{frm}_2(tr^i(j)) = f(k_1, \ldots, k_n)$. Then β^{i+1} and tr^{i+1} are defined by:
   - $I^{i+1} = I^i \cup \{j_1, \ldots, j_n\}$ where $j_1, \ldots, j_n$ are distinct new indices;
   - $sv^{i+1} = sv^i$;
   - $\mathit{frm}^{i+1} = \mathit{frm}^i \cup \{j \mapsto f(j_1, \ldots, j_n)\}$;
   - $tr^{i+1} = tr^i \cup \{j_1 \mapsto k_1, \ldots, j_n \mapsto k_n\}$;
   - $mo^{i+1}(j) = mo^i(j)$ for all $j \in I^i$ and $\langle mo^{i+1}(j_1), \ldots, mo^{i+1}(j_n) \rangle = \mathtt{EXTRM}(f, mo^i(j))$;
   - $ty^{i+1}(j) = ty^i(j)$ for all $j \in I^i$ and $\langle ty^{i+1}(j_1), \ldots, ty^{i+1}(j_n) \rangle = \mathtt{EXTRT}(f, ty^i(j))$;
   - $ps^{i+1} = ps^i \cup \{(j_l, k) |\ l \in \{1, \ldots, n\}, mo^{i+1}(j_l) \neq \mathit{ground}, (j,k) \in ps^i\}$.
3. Otherwise, $\beta' = \beta^i$ and $tr' = tr^i$.

Operation 5.9. $\mathtt{REF}_{\alpha}(\beta_1, \beta_2, tr_{1,2}) = \langle \beta', tr' \rangle$
It refines β_1 only considering the α component of β_2.

IMPLEMENTATION The implementation is as follows:

$$\begin{array}{lcll} I' & = & I_1 & \\ sv' & = & sv_1 & \\ \mathit{frm}' & = & \mathit{frm}_1 & \\ mo'(i) & = & mo_1(i) \sqcap \mathtt{UNIST}(mo_2(tr_{1,2}(i))) & \text{for all } i \in I' \\ ty'(i) & = & ty_1(i) \sqcap \mathtt{UNIST}(ty_2(tr_{1,2}(i))) & \text{for all } i \in I' \\ ps' & = & ps_1 & \\ tr' & = & tr_{1,2}. & \end{array}$$

Operation 5.10. $\mathtt{REF}_{ref}(\beta_1, \beta_2, tr_{1,2}) = \langle \beta', tr' \rangle$
It combines the two refinement operations defined above.

Implementation

$$
\begin{array}{lcl}
\langle \beta_3, tr_{3,2} \rangle & = & \mathtt{REF}_{frm}(\beta_1, \beta_2, tr_{1,2}) \\
\langle \beta', tr' \rangle & = & \mathtt{REF}_{\alpha}(\beta_3, \beta_2, tr_{3,2}).
\end{array}
$$

5.1.4. Unification of Two Variables. We are now in position to define $\mathtt{UNIF\_VAR}$.

Operation 5.11. $\mathtt{UNIF\_VAR}(\beta) = B'$

Specification Let β be an abstract substitution such that $dom(\beta) = \{X_1, X_2\}$. $\mathtt{UNIF\_VAR}(\beta)$ computes a pseudo abstract sequence B' such that:

$$
\left.\begin{array}{l}
\theta \in Cc(\beta) \\
\sigma \in mgu(X_1\theta, X_2\theta)
\end{array}\right\} \Rightarrow \langle \theta, < \theta\sigma > \rangle \in Cc(B')
$$

$$
\left.\begin{array}{l}
\theta \in Cc(\beta) \\
mgu(X_1\theta, X_2\theta) = \emptyset
\end{array}\right\} \Rightarrow \langle \theta, < > \rangle \in Cc(B').
$$

Implementation Let $\langle \beta_{out}, tr, ss, sf, U \rangle = \mathtt{UNIF\_VAR}_{old}(\beta)$. The pseudo abstract sequence $B' = \langle \beta'_{in}, \beta'_{ref}, \beta'_{out}, E'_{ref\_out}, E'_{sol} \rangle$ is defined by

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

where the structural mapping tr_{in_ref} is a canonical inclusion. The following commutative diagram is satisfied by tr_{in_ref}, tr_{ref_out} and the injections in_{ref} and in_{out}.

$$
\begin{tikzcd}
U \subseteq I = I'_{in} \arrow[r, "tr_{in\_ref}"] & I'_{ref} \arrow[r, "tr_{ref\_out}"] \arrow[dr, "in_{ref}"'] & I'_{out} \arrow[d, "in_{out}"] \\
 & & I'_{ref} + I'_{out}
\end{tikzcd}
$$

Remark 5.1. The precision of operation UNIF_VAR can be improved with a reexecution strategy (see, e.g., [48]): in the case where *ss* and *sf* are both false, we can reapply the unification operation to the abstract substitution β'_{ref} computed by $\mathtt{REF}_{ref}$. It may happen that the new abstract unification surely succeeds, allowing us to derive better information on the number of solutions. This improvement is needed to obtain optimal precision on the example of Section 2.

5.2. Concatenation of two Abstract Sequences

The second operation we present is the concatenation operation CONC. It is the counterpart for abstract sequences of the operation UNION, used in [47], which simply collects the information provided by two abstract substitutions into a single one. In fact, the operation CONC is similar to UNION for all but one component, namely E_{sol}; this is because the number of solutions of a procedure is the sum of the numbers of solutions of its clauses, not an "upper bound" of them. To obtain a good precision in the computation of E_{sol}, it is important to detect mutual exclusion of clauses [9, 46]. In our implementation, we generalize this idea. First, we compute the greatest lower bound of the β_{ref} component of the two abstract sequences. Then, we compute the sum of the numbers of solutions for this greatest lower bound only. In particular, when the greatest lower bound is equal to $\perp$, the clauses are exclusive, and no sum is computed: we only collect the numbers of solutions of the two clauses.

The implementation of CONC is complex but can be explained in a concise way through the use of special operations called constrained mappings that we present first. Some auxiliary operations are also described.

5.2.1. Constrained Mappings. Constrained mappings have been introduced in [49] as a formalism to manipulate indices. We give below a general definition of constrained mappings. This is a relaxation of the notion proposed in [49]. The reader can find the implementation for the size domain in [43].

Definition 5.2. [Constrained Mappings] Let I and I' be two finite sets of indices and $tr : I \to I'$ be a function. The *concrete constrained mapping* of tr is the pair of dual functions, $tr^{>}_{*} : \wp(\mathcal{T}^{I}) \to \wp(\mathcal{T}^{I'})$ and $tr^{<}_{*} : \wp(\mathcal{T}^{I'}) \to \wp(\mathcal{T}^{I})$ defined below. For all $\Sigma_I \in \wp(\mathcal{T}^{I})$ and $\Sigma_{I'} \in \wp(\mathcal{T}^{I'})$,

$$tr^{>}_{*}(\Sigma_I) = \{\langle s_i\rangle_{i\in I'} \in \mathcal{T}^{I'} \mid \exists\langle t_i\rangle_{i\in I} \in \Sigma_I : \forall i \in I, s_{tr(i)} = t_i\}$$

$$tr^{<}_{*}(\Sigma_{I'}) = \{\langle t_i\rangle_{i\in I} \in \mathcal{T}^{I} \mid \exists\langle s_i\rangle_{i\in I'} \in \Sigma_{I'} : \forall i \in I, t_i = s_{tr(i)}\}.$$

Let A_I and $A_{I'}$ be two abstract domains approximating $\wp(\mathcal{T}^{I})$ and $\wp(\mathcal{T}^{I'})$, respectively, with concretization functions Cc. An *(abstract) constrained mapping* is any sound approximation $tr^{>} : A_I \to A_{I'}$ and $tr^{<} : A_{I'} \to A_I$ of a concrete one, i.e.,

$$\forall \alpha_I \in A_I,\ tr^{>}_{*}(Cc(\alpha_I)) \subseteq Cc(tr^{>}(\alpha_I))$$

$$\forall \alpha_{I'} \in A_{I'},\ tr^{<}_{*}(Cc(\alpha_{I'}) \subseteq Cc(tr^{<}(\alpha_{I'})).$$

5.2.2. Auxiliary operations. Let us introduce some auxiliary operations.

Operation 5.12. $\mathtt{LUB}(\beta_1, \beta_2) = \langle \beta', tr_1, tr_2 \rangle$
This operations returns a pseudo-abstract substitution $\beta' = \beta_1 \sqcup \beta_2$ and two structural mappings tr_k between β' and β_k, i.e., $tr_k : I' \to I_k$ $(k = 1, 2)$.

Operation 5.13. $\mathtt{EXT\_LUB}(\beta_1, \beta_2) = \langle \beta', tr_1, tr_2, st \rangle$
It returns β', tr_1, tr_2 as above and a boolean value, st, such that $st = true$ implies that β' is a strict union, i.e., $Cc(\beta') = Cc(\beta_1) \cup Cc(\beta_2)$.

Operation 5.14. $\mathtt{GLB}(\beta_1, \beta_2) = \langle \beta', tr_1, tr_2 \rangle$
This operations returns a pseudo-abstract substitution $\beta' = \beta_1 \sqcap \beta_2$ and two structural functions tr_k between β_k and β', i.e., $tr_k : I_k \to I'$ $(k = 1, 2)$.

Operation 5.15. $\mathtt{SUM}_{sol}(E_1, E_2) = E'$
This operation is used to express the length of a sequence obtained by concatenating two other sequences.

Specification Let I be a set of indices and $E_k \in Sizes_{I+\{sol\}}$ $(k = 1, 2)$. $\mathtt{SUM}_{sol}(E_1, E_2)$ returns $E' \in Sizes_{I+\{sol\}}$ such that

$$\left.\begin{array}{l} (n_i^k)_{i \in I+\{sol\}} \in Cc(E_k)\ (k = 1, 2) \\ n_i^1 = n_i^2 = n_i\ (i \in I) \\ n_{sol} = n_{sol}^1 + n_{sol}^2 \end{array}\right\} \Rightarrow \quad (n_i)_{i \in I+\{sol\}} \in Cc(E').$$

Implementation Let sol_1 and sol_2 be two new variables.

$$E' \quad = \quad tr_{sol}^{<}(E_1[sol \mapsto sol_1] \cup E_2[sol \mapsto sol_2] \cup \{[\![sol = sol_1 + sol_2]\!]\})$$

where $tr_{sol} : I + \{sol\} \to I + \{sol, sol_1, sol_2\}$ is the canonical injection, and $E_i[sol \mapsto sol_i]$ is the set of (in)equations obtained by syntactically replacing every occurrence of sol by sol_i in E_i.

5.2.3. Concatenation of two abstract sequences. We are now in position to describe in details the concatenation operation CONC.

Operation 5.16. $\mathtt{CONC}(B_1, B_2) = B'$
This operation is used to concatenate the (abstract) results obtained from the execution of a procedure and a clause.

Specification Let $B_k = \langle \beta_{in}, \beta_{ref}^k, \beta_{out}^k, E_{ref\_out}^k, E_{sol}^k \rangle$ $(k = 1, 2)$ be two abstract sequences with $dom_{out}(B_1) = dom_{out}(B_2)$. $\mathtt{CONC}(B_1, B_2)$ returns a pseudo abstract sequence $B' = \langle \beta'_{in}, \beta'_{ref}, \beta'_{out}, E'_{ref\_out}, E'_{sol} \rangle$ such that $dom_{in}(B') =$

$dom_{in}(B_k)$, $dom_{out}(B') = dom_{out}(B_k)$ $(k = 1, 2)$ and

$$\left.\begin{array}{l}\langle\theta, S_1\rangle \in Cc(B_1)\\ \langle\theta, S_2\rangle \in Cc(B_2)\end{array}\right\} \;\Rightarrow\; \langle\theta, S_1 :: S_2\rangle \in Cc(B').$$

Implementation The implementation is defined as follows:

$$\begin{array}{lcl}
\beta'_{in} & = & \beta_{in}\\
\langle\beta'_{ref}, tr^1_{ref}, tr^2_{ref}, st\rangle & = & \mathtt{EXT_LUB}(\beta^1_{ref}, \beta^2_{ref})\\
\langle\beta'_{out}, tr^1_{out}, tr^2_{out}\rangle & = & \mathtt{LUB}(\beta^1_{out}, \beta^2_{out})
\end{array}$$

$$E'_{ref\_out} \;=\; (tr^1_{ref} + tr^1_{out})^<(E^1_{ref\_out}) \sqcup (tr^2_{ref} + tr^2_{out})^<(E^2_{ref\_out})$$

$$E'_{sol} \;=\; \begin{cases}
\begin{array}{l}(tr^1_{ref} + \{sol \mapsto sol\})^<(E^1_{sol}) \sqcup\\ (tr^2_{ref} + \{sol \mapsto sol\})^<(E^2_{sol}) \sqcup\\ (tr_{int} + \{sol \mapsto sol\})^<(\mathtt{SUM}_{sol}(\overline{E}^1_{sol}, \overline{E}^2_{sol}))\end{array} & \text{if } st\\[2ex]
\begin{array}{l}(tr^1_{ref} + \{sol \mapsto sol\})^<(E^1_{sol}) \sqcup\\ (tr^2_{ref} + \{sol \mapsto sol\})^<(E^2_{sol}) \sqcup\\ (tr_{int} + \{sol \mapsto sol\})^<(\mathtt{SUM}_{sol}(\overline{E}^1_{sol}, \overline{E}^2_{sol})) \sqcup\\ tr^>_{sol}([\![sol = 0]\!])\end{array} & \text{if } \neg st.
\end{cases}$$

where

$$\langle\beta_{int}, tr^1_{int}, tr^2_{int}\rangle \;=\; \mathtt{GLB}(\beta^1_{ref}, \beta^2_{ref})$$

$$\begin{array}{lcl}
\overline{E}^1_{sol} & = & (tr^1_{int} + \{sol \mapsto sol\})^>(E^1_{sol})\\
\overline{E}^2_{sol} & = & (tr^2_{int} + \{sol \mapsto sol\})^>(E^2_{sol})
\end{array}$$

and $tr_{sol} : \{sol\} \to I'_{ref} + \{sol\}$ is the canonical injection. The structural mappings tr^k_{ref}, tr^k_{int} $(k = 1, 2)$ and tr_{int} satisfy the following commutative diagram:

$$\begin{tikzcd}
& I^1_{ref} \arrow[dr, "tr^1_{int}"] & \\
I'_{ref} \arrow[ur, "tr^1_{ref}"] \arrow[rr, "tr_{int}"] \arrow[dr, "tr^2_{ref}"'] & & I_{int} \\
& I^2_{ref} \arrow[ur, "tr^2_{int}"'] &
\end{tikzcd}$$

The least upper bound operator $\sqcup$ between (in)equation systems is implemented as convex union (see [69]).

6. RELATED WORKS

Logic program analysis has attracted so many researchers in the last decade that it is not possible to give a comprehensive account of all interesting works related to ours. We focus on some of them, which can be integrated (at least partially) in an implementation of our analyser.

6.1. Logic Program Construction and Verification With Prolog

A methodology to verify the correctness of Prolog programs based on the theory of logic programming and a number of additional arguments has been proposed by K.R. Apt in [3]. The emphasis is on elegant methods which are not fully automatable but can be applied straightforwardly "by hand". Termination proofs for logic programs executing using the Prolog search rule is a prerequisite for the other aspects of the methodology but we delay the discussion of this topic to the next section entirely devoted to termination. Assuming termination, other desirable properties such as (partial) correctness, occur-check freedom, absence of run-time errors (for arithmetic predicates), and absence of floundering (for negated atoms) are established: occur-check freedom and absence of floundering can be verified by a syntactic analysis establishing that the program is *well moded* (or alternatively, but for occur-check only, *nicely moded*). Occur-check freedom and absence of floundering can be verified by our analyser thanks mainly to the mode and possible sharing components.[12] However, still better results could be obtained by enhancing the domain with a linearity component. Generally speaking, our approach is more powerful than the syntactic characterizations given by well moded and nicely moded programs because we can reason "inside" the terms bound to program variables. Absence of run-time errors is verified in Apt's approach by resorting to (a limited form of) directional types [1]. The same information can be derived by our analyser if the type component of the abstract domain is extended with information about numbers and ground arithmetic expressions.

Another methodology for Prolog program construction based on the logic programming paradigm has been proposed by Y. Deville in [32]. This methodology consists of three main steps: elaboration of a specification, construction of a logic description, and derivation of a Prolog procedure. The third step of the methodology involves a number of checks relative to the modes and the types of the arguments, the number of solutions to the procedure, and termination. Our analyser is strongly connected to Deville's proposal since our notion of abstract sequence is able to express the mode and multiplicity information of Deville's specification scheme. Our proposal even improves on Deville's by allowing us to specify structural and sharing information. Our treatment of types and termination is however not able to support the methodology in full generality because, in Deville's approach, types are arbitrary sets of terms and termination proofs may use arbitrary well-founded relations. Previous attempts to partially automate Deville's methodology have been made in the project FOLON [37]. The analyser described here can be viewed as an improvement of the FOLON analysers presented in [23, 24], which are only based on the abstract substitution notion and are unable to deal with termination, multiplicity, and term size relations. A more refined analyser, which includes multiplicity and termination analysis, has finally been presented in [14]. It can be seen as a preliminary version of the analyser proposed in this paper.

[12] Other components may improve the precision of mode and sharing information.

6.2. Termination Analysis of Logic Programs

Termination analysis of logic programs has received a lot of attention in the last few years (see [25] for a detailed survey). So, once again, we restrict our discussion to a few selected works.

The most general approach to proving termination of Prolog programs is probably the one of Y. Deville [32]. It basically consists of proving that recursive calls to a procedure are strictly decreasing with respect to some well-founded relation. A drawback of this approach is that it can be cumbersome to apply it "by hand," because it requires to explicitly reason about the execution of the procedure, according to Prolog operational semantics.

Thus, simpler methods have been investigated, the most fundamental of which are due to K.R. Apt, M. Bezem, and D. Pedreschi [2, 4, 5, 6]. They noticeably introduce the classes of *acceptable* and *semi-acceptable* programs which are guaranteed to terminate according to Prolog search rule, for a large class of queries (i.e., bounded queries). Such programs are characterized through the existence of a *level mapping*, which maps literals to natural numbers, and of a model I such that (roughly speaking) the level mapping of literals respecting the model decreases through embedded procedure calls. The simplicity of the method comes from the consideration of a model which relieves us of reasoning about Prolog operational semantics. The limitation to bounded queries (i.e., queries whose level mapping is bounded under ground instantiation) has been relaxed by A. Bossi, N. Cocco, and M. Fabris, who reason on terms that are *rigid*, i.e., whose norm is invariant under any instantiation [7, 8].

The previous methods cannot be fully automated since they involve finding a model and a level mapping for the program. Nevertheless, several (incomplete) automatic methods have been shown able to prove the termination of interesting classes of programs. The methods proposed by J.D. Ullman and A. Van Gelder [63], L. Plümer [58, 59, 60],and D. Schreye and K. Verschaetse [65, 66, 67] amounts to derive an *interargument relation* on the sizes of the arguments of a procedure and to using it to prove that the size of some argument decreases through recursive calls. In these methods, the interargument relation can be seen as a model of the procedure and can be inferred by means of bottom-up abstract interpretation. The size of arguments is however fixed by an a priori given norm. Further works by S. Decorte, D. De Schreye, and M. Fabris have addressed the issue of inferring norms automatically [31, 30].

The analyser that we presented in this paper can be seen as a partial implementation of Deville's approach because we use size relations between input and output terms without requiring term rigidity or similar conditions. For instance, our analyser can prove the termination of the following "impure" Prolog procedure, for any possible input:

```
close(X):- var(X), X=[].
close(X):- nonvar(X), X=[H|T], close(T).
```

Nevertheless, our use of norms is less general than Deville's use of arbitrary well-founded relations.

6.3. Abstract Interpretation and Logic Program Analysis

The design of our analyser is based on the methodology of abstract interpretation [10, 18, 20]. More specifically, we reuse the approach (and actually part of the code) of the system *GAIA* [47]. There are however two major differences between our analyser and *GAIA*. First, an analysis with *GAIA* (or with other similar systems (e.g., *PLAI* [56]) based on abstract interpretation frameworks such as [10, 41, 52, 53, 57]) operates on a complete program P and an (abstract) description of a top level goal. The system then explores the whole code of P and performs fixpoint computations to handle recursive calls. On the contrary, our analyser deals with each procedure of the program separately and exploits user-provided information to "solve" the literals of a clause (except unification and other built-in predicates). Second, the notion of abstract sequence that we use is more elaborated than the abstract substitution notion used in the various applications of *GAIA* (e.g., [16, 17, 47, 48, 64]). A simpler notion of abstract sequence has been introduced in *GAIA* recently [9, 45, 46] but it is less convenient than ours to express relations between input terms, output terms, and the number of solutions to a goal as well as to detect mutual exclusion of clauses.

The abstract domain for substitutions that we use in this paper is related to the abstract equation system (AES) introduced in [40] by G. Janssens, M. Bruynooghe, and A. Mulkers.[13] The structural description of the terms associated with the program variables is equivalent in both domains: in $\mathtt{Pat}(\Re)$, it is expressed by the same-value and frame components while, in the domain AES, abstract equations associating every program variable with an "abstract" term are used. A noticeable conceptual difference between the two domains lies in the interaction between the structural description and the information given by the other components: in the domain AES, such information is given only for the "leaves" (i.e., the abstract variables, representing the subterms whose structure is not known), while, in $\mathtt{Pat}(\Re)$, the particular component describes all indices (i.e., all subterms of the terms of the substitutions). Keeping information about all indices eases the construction of abstract operations [16] and, for some domains, increases the precision of the abstraction.[14] Moreover, if all indices are described, the "Generalise" operation of the domain AES reduces to an inverse constraint mapping (as it is no longer necessary to propagate this information to all indices). The generic domain $\mathtt{Pat}(\Re)$ is thus based on a representation that is closer to the implementation (of the abstract domain): operations on this domain can be easily translated to algorithms, thereby simplifying the correctness proof of an implemented system. Finally, if the cost is too high to keep information about all indices, or if it does not improve the precision of the information, it is possible to work only with leaves and to compute descriptions for all indices only when it is needed (e.g., before applying constraint mappings).

A similarity can also be seen between our work and the type, mode, and determinism system encapsulated in the programming language Mercury [61]. In fact, as already mentioned in the introduction, information like modes and types is crucial in every logic program analysis and a language aiming at incorporating optimiza-

[13] Note however that the first definition of `Pattern` [55] is anterior to the definition of AES.

[14] e.g., it is not possible to deduce information about the linearity of terms of the form $f(t_1, t_2)$ from the sole assertion that t_1 and t_2 are linear terms.

tion needs to deal with them. In practice, our pattern and type components are less expressive than Mercury type system but, conversely, determinism in Mercury does not benefit from size relations which results in an a priori less precise multiplicity analysis. Thus an analyser similar to ours could be integrated to Mercury conditional to a (substantial) improvement of the type component. (Techniques similar to [17] could be applied.) Such an analyser should then outperform the current Mercury analyser both for determinacy analysis and termination. Furthermore, our analyser could alternatively be used to transform pure untyped logic programs into Mercury programs (not into Prolog).

Another interesting relation can be seen with papers on declarative debugging [13] and even more with recent proposals on integrating verification and abstract interpretation techniques in a uniform, more general setting [36, 50]. All these proposals are mainly based on the assertion (precondition-postcondition) approach by Drabent and Maluszinsky [33]. The novelty of our approach is that the notion of abstract sequences allows us to characterize "success" input substitutions (by means of β_{ref}) and to deal with global information relating input and output substitutions (e.g., size relations) explicitly.

6.4. Automatic Complexity Analysis of Logic Programs

Automatic complexity analysis [27] is useful for automatically tuning the task granularity in parallel executions of logic programs [28]. It can be used also to select the most efficient Prolog version of a logic procedure [14]. Our analyser is able to verify precise relations between the sizes of the arguments and the number of solutions to a Prolog procedure. Thus it can be used as a basis for an automatic complexity analysis similar to [27]. The work in [27] is not based on abstract interpretation but instead it exploits general knowledge about logic programs; different size notions are used corresponding to different types (e.g., lists, integers) and the relation between the number of solutions and the size of terms is expressed by means of difference equations; finally, this work assumes a number of preliminary analyses. In our approach, all analyses are performed at the same time and may interact, which theoretically allows more precise analyses. However, in order to compete with [27], our abstract domain needs to be improved further to deal with multiple norms and difference equations.

7. CONCLUSION AND FUTURE WORKS

In this paper, we have presented a generic analyser for pure Prolog, designed according to a verification approach. The analyser is based on a notion of abstract sequence, which is expressive enough to model most semantic properties of terminating Prolog programs. Additionally to the description of the analyser, a complete domain of abstract sequences has been presented. This domain allows us to derive all kinds of information that are useful for Prolog program verification, in a single analysis: modes, types, sharing, sizes, determinacy, and multiplicity. The domain has nevertheless some limitations, mainly with respect to types. We also have described the implementation of two main operations over the domain in order to demonstrate how such operations can be designed. Finally, we have compared our approach to a number of other works relative to Prolog program verification and construction, termination analysis of logic programs, abstract interpretation and abstract domains, automatic complexity analysis, as well as to the analyser of the new logic language Mercury. Thus, in our opinion, this paper contains sufficient material to allow an implementor building a practical system in which state-of-the-art techniques of Prolog program verification can be integrated.

Although our analyser has been presented for pure Prolog, it can be readily extended to deal with most non pure features of Prolog. We have incidentally mentioned how this can be done in the previous sections. Now, we summarize this issue. Arithmetic built-ins, such as `is` and `<`, and test predicates, such as `var` and `ground`, can be handled without additional coding by providing behaviours capturing their operational semantics. (Unification could also be handled by means of a set of behaviours but, due to the ubiquitous character of this operation, such a treatment would be inaccurate.) The treatment of the cut requires to enhance the concrete and abstract domains with so-called "cut information" in the style of [9, 45, 46]; such a treatment can be integrated in our analyser, since it is based on the same concrete semantics. Furthermore, as negation by failure is easy modeled through the cut, it can also be handled simply. Some Prolog systems include a "non floundering" test to ensure that negated atoms are executed safely. Such a test can be performed statically in our analyser thanks to the mode and possible sharing components. The occur-check can be treated by the same means. Nevertheless, other aspects of some Prolog systems such as the "dynamic predicates" `assert` and `retract` cannot be handled by our analyser; neither can other treatments of negation such as delaying non ground negated atoms. We are aware of no rigorous methods to verify programs using these features, however.

We are currently completing an implementation of the analyser based on the domain presented in this paper. In fact, we have been able to reuse most of the code of *GAIA* [47] but we still have to implement the operations on the size components based on the polyhedron library of D.K. Wilde [69]. Our next task will be to apply the analyser to the verification of a significant number of Prolog programs. A further step will be to extend the analyser with more powerful abstract domains for types [17, 39], sharing [38], and linearity [62].

In addition to the implementation of a complete analyser, various applications of it will be investigated. First, we will go back to the problem of deriving correct Prolog implementations of purely declarative descriptions. More specifically, we will investigate various logic description (or program) classes which can be obtained by inductive [34] or deductive [11, 35, 42, 68] synthesis. Following the general idea of [32], we will investigate how our analyser can be used to prove that some Prolog translation of such logic descriptions correctly implements the intended meaning of the descriptions according to the correctness criteria proposed by the authors of [34, 35], respectively. This will require to integrate the correctness criteria and our behaviour notion into a convenient specification schema similar to [32]. Second, we will extend our analyser to perform an automatic complexity analysis in the spirit of [27]. Such an analysis can be seen as a relatively straightforward by-product of our analysis of the number of solution to a procedure. Best-case and worst-case analyses are both obtainable since our component E_{sol} provides lower and upper bounds to the number of solutions. Finally, our ultimate goal will be to derive the most efficient version of a Prolog procedure automatically thanks to the results of the complexity analysis.

ACKNOWLEDGEMENTS

We wish to thank the referees for their much useful comments and suggestions for improvements. Sabina Rossi and Agostino Cortesi are partially supported by the Italian MURST project N. 9701248444-044: *Tecniche formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software*. Baudouin Le Charlier and Christophe Leclère are partially supported by the HCM project ABILE N. ERBCHRXCT940624: *Abstract Interpretation of Declarative Languages*.

REFERENCES

1. A. Aiken and T.K. Lakshman. Directional Type Checking of Logic Programs. In B. Le Charlier, editor, *First International Static Analysis Symposium*, number 864 in Lecture Notes in Computer Science, pages 43–60, Namur, Belgium, September 1994. Springer-Verlag.

2. K. R. Apt and D. Pedreschi. Studies in Pure Prolog: Termination. In J.W. Lloyd, editor, *Proc. Symp. on Computational Logic*, volume 1 of *Basic Research Series*, pages 150–176. Springer-Verlag, Berlin, 1990.

3. K.R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.

4. K.R. Apt and D. Pedreschi. Proving Termination of General Prolog Programs. In *Proc. International Conference on Theoretical Aspects of Computer Science*, Sendai, Japan, 1991.

5. K.R. Apt and D. Pedreschi. Modular Termination Proofs for Logic and Pure Prolog Programs. Technical Report 6/93, Dipartimento di Informatica, Università di Pisa, 1993.

6. M. Bezem. Characterizing Termination of Logic Programs with Level Mappings. *Journal of Logic Programming*, 15(1 & 2):79–98, 1992.

7. A. Bossi, N. Cocco, and M. Fabris. Typed norms. In B. Krieg-Brueckner, editor, *Proc. ESOP'92*, pages 73–92. Springer-Verlag, LNCS 582, 1992.

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

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

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

11. A. Bundy, A. Smaill, and G. Wiggins. The synthesis of logic programs from inductive proofs. In J.W. Lloyd, editor, *Computational Logic*, Esprit Basic Research Series, 1990.

12. M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 194–205. ACM Press, 1993.

13. M. Comini, G. Levi, M.C. Meo, and G. Vitiello. Proving Properties of Logic Programs by Abstract Diagnosis. In Mads Dam, editor, *Proc. of the Fifth Workshop on Analysis and Verification of Multiple-Agent Languages (LOMAPS'96)*, volume 1192 of *LNCS*. Springer Verlag, June 1996.

14. A. Cortesi, B. Le Charlier, and S. Rossi. Specification-Based Automatic Verification of Logic Programs. In *Logic Program Synthesis and Transformation. Proceedings of the 6th International Workshop, LOPSTR'96*, volume 1207 of *LNCS*. Springer Verlag, August 1996.

15. 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, Institute of Computer Science, University of Namur, Belgium, (also Brown University), Namur, Belgium, 1993.

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

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

18. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, January 1977.

19. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundation. In *Proc. ACM Symposium on Artificial Intelligence and Programming Languages*, pages 1–12. SIGPLAN Notices, 1977.

20. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2–3), 1992.

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

22. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In ACM Press, editor, *Conference Record of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages*, pages 83–94, Aug 1992.

23. P. De Boeck and B. Le Charlier. Static Type Analysis of Prolog Procedures for Ensuring Correctness. In *Proc. of Programming Language Implementation and Logic Programming PLILP'90*, volume 456 of *Lecture Notes in Computer Science*, pages 222–237, Linköping, Sweden, August 1990. Springer-Velag.

24. P. De Boeck and B. Le Charlier. Mechanical Transformation of Logic Definitions Augmented with Type Information into Prolog Procedures: Some Experiments. In *Proceedings of LOPSTR'93*, Workshops in Computer Science. Springer Verlag, July 1993.

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

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

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

28. S.K. Debray, N.W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proceedings ACM SIGPLAN'90 conference on programming language design and implementation*, pages 174–188, June 1990.

29. S.K. Debray and D.S. Warren. Functional computations in logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(3):451–481, July 1989.

30. S. Decorte, D. De Schreye, and M. Fabris. Exploiting the Power of Typed Norms in Automatic Inference of Interargument Relations. Technical report, Department of Computer Science, K.U.Leuven, Belgium, 1994.

31. S. Decorte, D. De Schreye, and M. Fabris. Automatic inference of norms : a missing link in automatic termination analysis. In D. Miller, editor, *Proceedings ILPS'93*, pages 420–436, Vancouver, Canada, 1993.

32. Y. Deville. *Logic Programming: Systematic Program Development.* MIT Press, 1990.

33. W. Drabent and J. Maluszyński. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59:133–155, 1988.

34. P. Flener and Y. Deville. Logic Program Synthesis from Incomplete Specifications. *Journal of Symbolic Computation: Special Issue on Automatic Programming*, 1993.

35. P. Flener and K.-K. Lau. Program Schemas as Steadfast Programs. Technical Report BU-CEIS-97, Bilkent University, Department of Computer Science, 1997.

36. M. Gallardo, P. Merino, and J.M. Troya. Relating Abstract Interpretation with Logic Program Verification. In A. Bossi, editor, *ILPS'97 Post-Conference Workshop on Verification, Model-Checking and Abstract Interpretation*, Port Jefferson, USA, 1997.

37. J. Henrard and B. Le Charlier. FOLON: An Environment for Declarative Construction of Logic Programs (Extended Abstract). In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Workshop on Programming Language Implementation and Logic Programming (PLILP'92)*, Lecture Notes in Computer Science, Leuven, August 1992. Springer-Verlag.

38. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In E.L. Lusk and R.A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming (NACLP'89)*, pages 154–165, Cleveland, Ohio, October 1989. MIT Press.

39. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. *Journal of Logic Programming*, 13(2-3):205–258, 1992.

40. G. Janssens, M. Bruynooghe, and A. Mulkers. Abstract equation systems: Description and insights. Report CW217, Department of Computing Science, K.U. Leuven, November 1995.

41. N.D. Jones and H. Søndergaard. A Semantic-Based Framework for the Abstract Interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 6, pages 123–142. Ellis Horwood Limited, 1987.

42. K.K. Lau and S.D. Prestwich. Top-down Synthesis of Recursive Logic Procedures from First-order Logic Specifications. In D.H.D. Warren and P. Szeredi, editors, *Proc. Seventh Int'l Conf. on Logic Programming*, pages 667–684. The MIT Press, Cambridge, Mass., 1990.

43. B. Le Charlier, C. Leclère, S. Rossi, and A. Cortesi. Automated verification of prolog programs. Research Paper RP-98-002, Institute of Computer Science, University of Namur, Belgium, 1998.

44. B. Le Charlier and S. Rossi. Sequence-Based Abstract Semantics of Prolog. Technical Report RR-96-001, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, February 1996.

45. B. Le Charlier, S. Rossi, and P. Van Hentenryck. An Abstract Interpretation Framework Which Accurately Handles Prolog Search-Rule and the Cut. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium (ILPS'94)*, Ithaca NY, USA, November 1994. MIT Press.

46. B. Le Charlier, S. Rossi, and P. Van Hentenryck. Sequence-Based Abstract Interpretation of Prolog. Technical Report RR-97-001, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, January 1997.

47. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):35–101, January 1994.

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

49. 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 RP-96-028, University of Namur, Belgium, September 1996.

50. G Levi and Volpe P. A Reconstruction of Verification Techniques by Abstract Interpretation. In A. Bossi, editor, *ILPS'97 Post-Conference Workshop on Verification, Model-Checking and Abstract Interpretation*, Port Jefferson, USA, 1997.

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

52. K. Marriott and H. Søndergaard. Semantics-based Dataflow Analysis of Logic Programs. In G. Ritter, editor, *Information Processing'89*, pages 601–606, San Fransisco, California, 1989.

53. C. S. Mellish. Abstract Interpretation of Prolog Programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 8, pages 181–198. Ellis Horwood Limited, 1987.

54. Daniel Le Métayer. Program analysis for software engineering: new applications, new requirements, new tools. *ACM Computing Surveys*, 28(4es):167–167, December 1996.

55. K. Musumbu. *Interprétation Abstraite de Programmes Prolog.* PhD thesis, Institute of Computer Science, University of Namur, Belgium, September 1990. In French.

56. K. Muthukumar and M. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2-3):315–347, August 1992.

57. U. Nilsson. Systematic Semantic Approximations of Logic Programs. In P. Deransart and J. Małuszyński, editors, *Proc. of the International Workshop on Programming Language Implementation and Logic Programming (PLILP'90)*, volume 456 of *Lecture Notes in Computer Science*, pages 293–306, Linköping, Sweden, August 1990. Springer-Verlag.

58. L. Plümer. Termination Proofs for Logic Programs based on Predicate Inequalities. In *Proceedings ICLP'90*, pages 634–648, Jerusalem, June 1990. MIT Press.

59. L. Plümer. Automatic Termination Proofs for Prolog Programs Operating on Nonground Terms. In *Proc. ILPS'91*, pages 503–517, San Diego, October 1991. MIT Press.

60. Plümer, L. Automatic Verification of GHC-Programs: Termination. In *Proceedings FGCS'92*, Tokyo, 1992.

61. Somogyi, Z. and Henderson, F. and Conway, T. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

62. Søndergaard, H. An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction. In B. Robinet and R. Wilhelm, editors, *Proceedings of the European Symposium on Programming (ESOP'86)*, volume 213 of *Lecture Notes in Computer Science*, pages 327–338, Sarrbruecken, Germany, March 1986. Springer-Verlag.

63. J.D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *Journal ACM*, 35(2):345–373, April 1988.

64. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of *Prop*. *Journal of Logic Programming*, 23(3):237–278, June 1995.

65. K. Verschaetse. *Static Termination Analysis for Definite Horn Clause Programs.* PhD thesis, Dept. Computer Science, K.U.Leuven, 1992.

66. K. Verschaetse and D. De Schreye. Deriving Termination Proofs for Logic Programs, using Abstract Procedures. In *Proc. ICLP'91*, pages 301–315, Paris, June 1991. MIT Press.

67. K. Verschaetse, S. Decorte, and D. De Schreye. Automatic Termination Analysis. In *Proc. LOPSTR'92, LNCS*. Springer-Verlag, 1993.

68. G.A. Wiggins. Synthesis and Transformation of Logic Programs in the Whelk Proof Development System. In *Proc. of the 1992 Joint International Conference and Symposium on Logic Programming.* The MIT Press, Cambridge, Mass., 1992.

69. D. K. Wilde. A Library for Doing Polyhedral Operations. Technical Report No. 785, IRISA-Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes Cedex-France, 1993.