MASTER IN COMPUTER SCIENCE

Automated verification of Prolog programs: an implementation

Gobert, François

Award date:
2003

Facultés Universitaires Notre Dame de la Paix
Institut d'informatique
Rue Grandgagnage, 21
B-5000 Namur

Automated verification of Prolog programs: an implementation

François Gobert

Promoteur :

Baudouin Le Charlier

Co-promoteur :

Christophe Leclère

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique
Année académique 2002-2003

Abstract

The aim of this work is to implement a static analyser of Prolog programs. According to specifications provided by the user including some information about types, modes, sharing, relations between the input and output sizes of terms, multiplicity, and termination, the analyser will check if the input Prolog program respect or no these specifications. It relies on an Abstract Interpretation framework. Conceptually, the analyser is based on the notion of abstract sequence which makes it possible to collect all the desirable information, and which allows a step by step analysis of a clause, able to model the result of the execution of a goal.

We cover all the theoretical aspects of this analyser (relying on the paper of B. Le Charlier et al.: *Automated verification of Prolog programs* [14]) and we give in detail the algorithms of all the abstract operations. Finally we discuss the practical implementation in Java.

Keywords:
Abstract interpretation; Automated verification; Logic programs; Prolog

Résumé

Ce travail consiste en l'implémentation d'un analyseur statique de programmes Prolog. Etant données des spécifications relatives aux types, modes, partages de variables, relations entre les tailles des termes en entrée et en sortie, la connectivité, et la terminaison, l'analyseur vérifie si le programme Prolog en entrée respecte ou non ces specifications. Il est construit grâce à la méthode d'interprétation abstraite. Conceptuellement, l'analyseur se base sur la notion de séquence abstraite, qui collecte toutes les informations désirables, et qui permet une analyse étape par étape d'une clause, capable de modéliser le résultat de l'exécution d'un but.

Nous couvrons tous les aspects théoriques de l'analyseur (se référant au rapport de B. Le Charlier et al. : *Automated verification of Prolog programs* [14]) et nous fournissons les algorithmes détaillés de toutes les opérations abstraites. Finallement, nous discutons l'implémentation effective en Java.

Mots-clés:
Interprétation abstraite; Vérification automatique; Programmes logiques; Prolog

Contents

List of Tables

List of Figures

List of Operations[1]

[1]Note: pages in *Automated Verification* (*AV*) [14], in *GAIA* [8] and in *Pat(R)* [7] references are annotated so that you can relate in those papers the presented algorithms. Note that in *GAIA*, $\alpha = \langle mo, frm, ps \rangle$ but in *AV*, $\alpha = \langle mo, ty, ps \rangle$. In this report, we readapt all the definitions taken from *GAIA* to the definitions of *AV*.

Acknowledgments

I would like to thank all the people who have contributed to this report.

I especially want to thank Sabina Rossi and Agostino Cortesi, who invited me at Universita' Ca' Foscari (Venezia-Mestre, Italy), for their support as well as for their useful discussions and valuable criticisms. They spent a lot of their time for me. They had a major impact on my work and on this paper. I also thank Baudouin Le Charlier and Christophe Leclère who supervised the whole project. Their careful and detailed comments were very judicious. They gave me the opportunity and the "taste" to work on this exciting domain.

Also this work would not been feasible without the contribution of Massimiliano Rospini. We worked together during four months on the subject (mainly on the understanding and the updating of the operations on abstract substitutions). I would like then to take this opportunity to thank him again for this fruitful cooperation. I have benefit from his advices and comments in different stages of the paper writing.

I am grateful to the reviewers whose suggestions were very helpful in improving the presentation of this report.

Finally I would like to thank my parents without whom these five years at University of Namur would not have been possible.

Chapter 1

Introduction

Declarative programming and logic programming in particular have received a lot of attention in the last years. Such languages allow the programmer to concentrate on the description of the problem to be solved and to ignore low level implementation details. Nevertheless, the implementation of declarative languages remains a delicate issue: since efficiency 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 on 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 and a number of non logical operations such as the test predicates (e.g., `var`, `novar`), the negation by failure (`not`), the cut (`!`), and so on.

Various forms of program analyses have been investigated by numerous researchers in order to improve on this situation. A methodology for Prolog program construction has been proposed by Y. Deville in [2]. 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, in order to **1)** find a correct permutation, and then to **2)** optimize this permutation by applying some transformations.

A static analyser often concentrates on one of these two aspects, but in general, the same information serves both of them. The analyser we will describe in this report is useful mainly in **1)** but its results can be used also to validate the transformations done in **2)**.

We have discerned two broad classes among static analysers, according to they are oriented towards verification or towards optimization. Let us illustrate them.

1.1 Analyses oriented towards Verification

Some analyses attempt to verify that a non declarative implementation of a program in fact behaves accordingly to its declarative meaning. Basically, the analyses reject programs whose operational behaviour is not guaranteed to meet their logical meaning but it is often possible to do better by reordering the literals in the clauses: the analysis then serves a basis for a transformation technique. Alternatively, such analyses can be used to perform static debugging.

A motivating sample

Let us illustrate how a "dataflow" analysis can help the programmer to transform a first (declaratively but not operationally correct) version of a program into a both declaratively and operationally correct version.

Procedure
$delete(X, L, Ldel)$

Type
X: any term
L, $Ldel$: lists

Relation
X is an element of L and $Ldel$ is L without the first occurrence of X.

Directionality
in(any,ground,any):out(ground,ground,ground)

Logic (declarative) construction

$$\begin{aligned} &delete(X, L, Ldel) \Leftrightarrow \\ &\quad L = [H|T] \wedge \\ &\qquad (H = X \wedge Ldel = T \wedge list(T)) \\ &\qquad \vee \\ &\qquad (H \neq X \wedge Ldel = [H|Tdel] \wedge delete(X, T, Tdel)) \end{aligned}$$

Syntactic transformation (in Prolog)

$$\begin{aligned} delete(X, L, Ldel) \quad &\leftarrow \quad L = [H|T], H = X, \\ &\qquad\quad Ldel = T, list(T). \\ delete(X, L, Ldel) \quad &\leftarrow \quad L = [H|T], not(H = X), \\ &\qquad\quad Ldel = [H|Tdel], delete(X, T, Tdel). \end{aligned}$$

Now if we query $delete(X, [1, 2, 1, 3], Ldel)$, only one solution – derived from the first clause – is calculated:

```
X = 1
Ldel = [2,1,3] ;

No
```

As the following dataflow analysis emphases, the reason comes from the fact that the negation $not(H = X)$ in the second clause fails. Indeed, because X is a variable, the unification $H = X$ never fails, such that $not(H = X)$ never succeeds. In order to solve this problem, it suffices to allow a call to the negation built-in only when all of its arguments are ground. In this example, this can be easily reached by putting it at the end of the clause.

Dataflow analysis

N.B. a = any
g = ground

- **Clause 1**

{X/a, L/g, Ldel/a}
$delete(X, L, Ldel) \leftarrow$
{X/a, L/g, Ldel/a, H/var, T/var}
$L = [H|T]$,
{X/a, L/g, Ldel/a, H/g, T/g}
$H = X$,
{X/g, L/g, Ldel/a, H/g, T/g}
$Ldel = T$,
{X/g, L/g, Ldel/g, H/g, T/g}
$list(T)$.
{X/g, L/g, Ldel/g, H/g, T/g}
{X/g, L/g, Ldel/g}

- **Clause 2**

{X/a, L/g, Ldel/a}
$delete(X, L, Ldel) \leftarrow$
{X/a, L/g, Ldel/a, H/var, T/var, Tdel/var}
$L = [H|T]$,
{X/a, L/g, Ldel/a, H/g, T/g, Tdel/var}
$not(H = X)$,
Not ok because X not g
To place at the end
$Ldel = [H|Tdel]$,
{X/a, L/g, Ldel/[H|Tdel], H/g, T/g, Tdel/a}
$delete(X, T, Tdel)$,
Possible because X/a, T/g, Tdel/a
{X/g, L/g, Ldel/g, H/g, T/g, Tdel/g}
$not(H = X)$.
OK because H and X g
{X/g, L/g, Ldel/g, H/g, T/g, Tdel/g}
{X/g, L/g, Ldel/g}

Final code (after the reordering)

$$\begin{array}{lll} delete(X, L, Ldel) & \leftarrow & L = [H|T], H = X, \\ & & Ldel = T, list(T). \\ delete(X, L, Ldel) & \leftarrow & L = [H|T], Ldel = [H|Tdel], \\ & & delete(X, T, Tdel), not(H = X). \end{array}$$

By now, if we query $delete(X, [1, 2, 1, 3], Ldel)$ again, all the possible solutions are returned:

```
X = 1
Ldel = [2,1,3] ;

X = 2
Ldel = [1,1,3] ;

X = 3
Ldel = [1,2,1] ;

No
```

1.2 Analyses oriented towards Optimization

Other analyses aim at optimizing programs automatically, relieving the programmer from using impure control features. Some optimizations can be expressed by source-to-source transformations such as introduction of cuts, replacement of negated literals by cuts, and partial evaluation.

A motivating sample

Consider the following three correct versions of the procedure `delete/3`:

Version 1

```
delete_1(X,L,Ldel) :- L=[H|T],X=H,Ldel=T.
delete_1(X,L,Ldel) :- L=[H|T],Ldel=[H|Tdel],
                      delete_1(X,T,Tdel),not(X=H).
```

Version 2

```
delete_2(H,[H|T],T).
delete_2(X,[H|T],[H|Tdel]) :- delete_2(X,T,Tdel),not(X=H).
```

Version 3

```
delete_3(H,[H|T],Tdel) :- !,Tdel=T.
delete_3(X,[H|T],[H|Tdel]) :- delete_3(X,T,Tdel).
```

In Table 1.1 we compare the related execution times of those three versions and we see that the introduction of a cut (in version 3) in place of an explicit negation (in versions 1 and 2) makes it well more performing.

Data			Related execution times		
X	L	Ldel	delete_1	delete_2	delete_3
2	[1,2,3,2,4]	[1,3,2,4]	8	2	1
2	[1,2,3,2,4]	[1,2,3,4]	8	3	1

Table 1.1: Efficiency of `delete/3`. Source: [2].

1.3 Contribution of this paper

An automated verification of Prolog programs – based on an abstract interpretation[1] methodology – was proposed in [13] and [14]. In this report, we provide a detailed description and an implementation of that static analyser.

Our work was mainly concerned with the acquisition of the needed background and the formalization of the problem. In order to obtain a strong theoretical basis, we did a kind of "unification" of the concepts and notations of different references (mainly [7], [8], [13] and [14]). That job was not so easy: we needed clear insight and in-depth understanding of the nontrivial problems we have encountered.

We didn't want to reinvent the wheel: some elements of the analyser were already well written. So we directly took back them without paraphrasing (further more, there is not so many ways to formalize those concepts). However, some existing implementations were sometimes readjusted in order to improve on their accuracy (this is necessary if we want our analyser to give "good" results). Also, new fresh algorithms never implemented before appear in this report.

Last but not least, this "self-contained" report has served as a basis for an effective implementation in Java.

1.4 Plan of the paper

The rest of the report is organized as follows: Chapter 2 provides an overview of the functionalities of the analyser based on a simple example. In Chapter 3 we provide the concrete semantics of normalized Prolog programs and we explain rapidly the different standard (concrete) operations that will be later approximated by their non-standard (abstract) ones. Chapter 4 contains a complete description of our domain of abstract sequences. The execution of the analyser (i.e., the abstract semantics) is then described in Chapter 5. In Chapters 6, 7 and 8 you will find the specifications and the implementations of the abstract operations, written in a formal, mathematical way. Chapter 9 discusses the practical implementation of the analyser (coded in Java), where we explain quickly the data types used for the different abstract domains. Finally, Chapter 10 concludes.

The appendix contains the following materials: Appendix A shows the concrete syntax written in ABNF of the Pure Prolog procedures and of the formal specifications. The transformational semantics of such a formal specification is given in Appendix B. Next we explain in Appendix C how to transform a

[1] Abstract interpretation [4, 5] is a general methodology for systematic development of static program analysis. An abstract interpretation framework is centered around the definition of a *non-standard* (or *abstract*) *semantics* approximating a concrete semantics of the language.

Prolog procedure into its normalized counterpart. Appendix D gives an informal description of the classes belonging to the Java packages and shows some UML-like diagrams displaying the structural relations between the classes (the organization of the packages). Appendix E explains briefly the pseudo-code of the analyser (i.e., the abstract semantics). Finally, Appendix F provides some output reports (i.e., some analyses tested by our analyser implemented in Java), where the different steps of the analyses are shown.

Chapter 2

Overview of the analyser

In this Chapter, we give an overview of the analyser. First, we present the "big steps" of the analyser. We then identify what information is useful for verification. Finally, we show on a example the functionalities that we want to achieve.

2.1 Steps of our analyser

In practice, how does our analyser work? Important question indeed!

The analyser processes two files.

First the user gives us in some file the code of the program he writes in Prolog and he wants to verify. In general it contains several so-called *logic procedures*. In Appendix A.2 we show the concrete syntax written in ABNF of the Pure Prolog subset we will accept to proceed.

Then the user gives us in an other file **all** the formal specifications of **all** the used procedures. A procedure may have a lot of different specifications (or behaviours), according to its use: these specifications mainly differ from one another by the so-called *directionality* of their call (i.e., different modes are used). In Appendix A.3 we show the grammar (the concrete syntax) written in ABNF defining such formal specifications.
The user can specify logic procedures without giving their effective implementation (i.e., their Prolog code). Indeed this may be done because when analyzing a logic procedure, if we encounter a predicate literal (that is a subcall to another procedure if we are talking with the operational paradigm terminology) the analyser just relies on the *reading* of the formal specifications of the called procedure. Note that this approach is different from the one of *GAIA* [8] where the analyser does not exploit user-provided information to "solve" recursive calls but performs fix-point computations on the code.

Let us define the steps of our analyser (they are depicted in Figure 2.1):

1. Parse the Prolog procedures and construct the abstract tree that represents (i.e., that "encodes internally") them. It will be very important to think about an efficient data type so that we can access rapidly to every procedures and to every program points of a specific clause (designed for the "step by step" static analysis).

2. Transform all the clauses of all the procedures in their normalized form (in Appendix C, we show the syntax of normalized programs and we explain how such translation can be done).

3. In a parallel step, parse the formal specifications and construct the abstract tree that represents them. Of course, if the user forgot to write any specifications, the further analysis will fail.

4. A normalization process must be done here also, where the user-defined variables used as arguments in the formal specifications must be mapped onto normalized variables. Refer to Appendix B where we explain how to package the information contained in an user-provided specification into a corresponding "abstract" behaviour (that is a big structure of our abstract domain containing an abstract sequence).

5. Finally but the major one, the analyser does its proper job and checks if yes or no the program written by the user (in its normalized form) is correct according to the specifications he received (the set of Behaviours). It consists of abstract executions of clauses, relying on abstract operations acting on the abstract domains.

6. At the end of the analysis, the global result will be either a success or a failure. The result has to be interpreted carefully: a success means that the program fully satisfies the specifications represented by the behaviours; on the other side, a failure means that either the program is not correct with respect to the specifications, **or** that the analyser is not precise enough to check its correctness with respect to the specifications. The output reports contains the traces of the analyses (one report for each pair of procedure/specification).

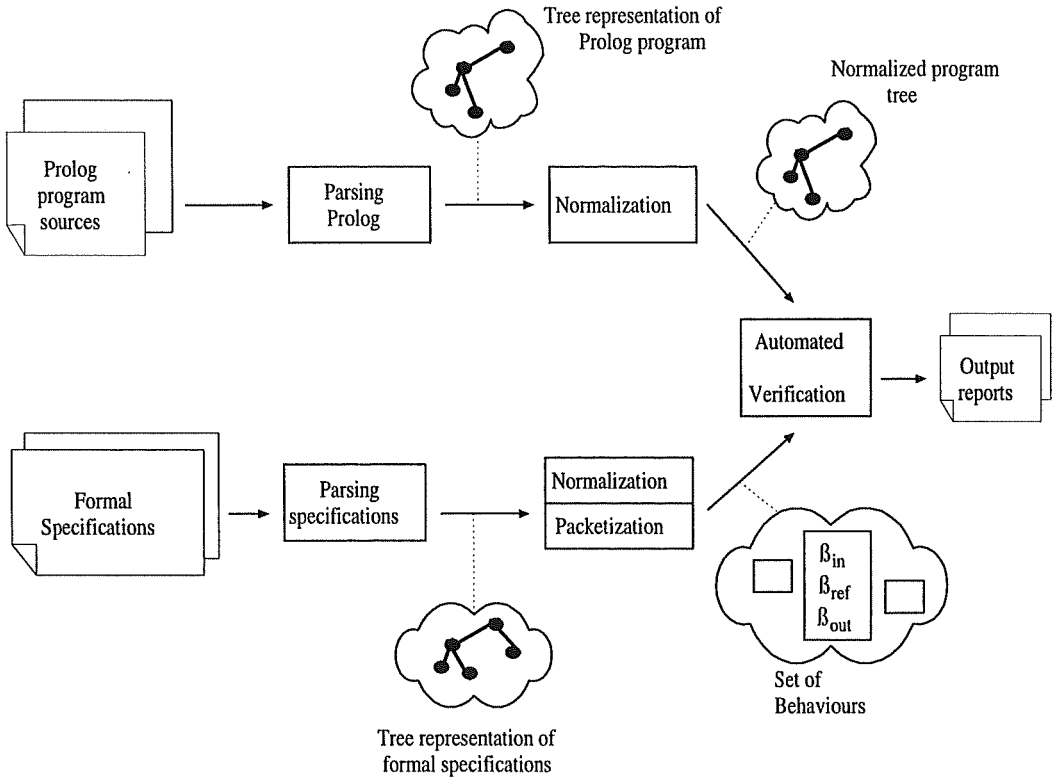


Figure 2.1: Big steps of the analyser.

2.2 What information is useful?

The nature of the information useful for the various applications of logic and Prolog program analyses is nowadays well identified. 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.
- *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.

2.3 Informal overview of the analyser

Before embarking on the technical description of our analyser, we show on an example the functionalities that we want to achieve.

Consider the Prolog procedures provided by the user depicted in Figure 2.2:

```
list([]).
list([_|LS]) :- list(LS).

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

Figure 2.2: The procedures `list/1` and `select/3`.

Declaratively, the procedure `select/3` defines a relation select(X, L, 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.

Because our analyser only processes normalized Prolog programs (see Appendix C), the procedures are transformed[1] as depicted in Figure 2.3:

```
list(L):- L=[].
list(L):- L=[H|T], list(T).

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

Figure 2.3: The normalized procedures `list/1` and `select/3`.

Our analyser is not aimed at verifying the (informal) declarative specification but instead it 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). For this class of calls, the user has to provide a description of the expected behaviour of the procedure by means of the formal specifications depicted in Figure 2.4.

[1] For the sake of clarity, the normalized variables have not been named `X1`, `X2`, `X3`, ...

```
% Specification of "list/1"
list(in(L:ground),
     ref(list),
     out(_),
     srel(),
     sol(sol = 1),
     sexpr(L))

% Specification of "select/3"
select(in(X:var , L:ground , LS:var ; noshare={(X,LS)}),
       ref(_ , [_|list] , _),
       out(ground , _ , ground list),
       srel(L_ref = LS_out + 1),
       sol(sol = L_ref),
       sexpr(L))
```

Figure 2.4: Formal specifications for `list/1` and `select/3`.

In order to explain the meaning of such a specification, we view the (concrete) semantics[2] of the procedure `select/3` as a (total) function that maps every (input) substitution θ such that $dom(\theta) = \{\mathtt{X}, \mathtt{L}, \mathtt{LS}\}$ 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 analyzed) 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.

We give an informal overview of the different parts of the formal specification of `select/3`:

- The `in` part states that the acceptable input substitutions θ are exactly those such that $\mathtt{X}\theta$ and $\mathtt{LS}\theta$[3] are distinct variables and $\mathtt{L}\theta$ is any ground term. The fact that X and LS are distinct is expressed by the no-sharing information in the `in` part.

- The `ref` part 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. 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

[2]See Chapter 3 where we explain more precisely the concrete domain and the concrete semantics of (normalized) Prolog programs.

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

specification all pieces of information which can be inferred from another part.

- The `out` part 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` part describes a relation between the sizes of input terms and the sizes of output terms. In this case, it says that the input size of L is always equal to the output size of LS plus 1.
- The `sol` part describes a relation between the sizes of input terms and the number of solutions to the call. In this case, it says that the number of solutions (i.e., the length of S) is equal to the input size of L.
- The `sexpr` part is useful to prove termination. Based on a norm[4], the `sexpr` part of the specification describes a positive integer linear function of the input terms sizes, which must strictly 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.

Technically, the first five parts of a specification define a mathematical object called *abstract sequence*:

$$B = \langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle.$$

The semantics of abstract sequences is defined in Section 4.3.2.

From the procedure and the above information, the analyser computes a number of abstract sequences: one for every prefix of the body of every clause of the procedure, one for every clause, and, finally, one for the complete procedure.

For instance, in our example, the analyser computes an abstract sequence B_1 expressing that (for the specified class of input calls) the first clause succeeds if and only if L is a non-empty list and that it succeeds exactly once in this case. The derivation of this information is possible because the analyser is able to detect that the unification `L=[X|LS]` succeeds if and only if L is of the form `[`t_1`|`t_2`]` (*not necessarily a list*) and because X and LS are free *and do not share*. Moreover, the analyser needs an abstract sequence describing the behaviour of the procedure `list/1`. The abstract sequence states that, for ground calls, the literal succeeds only for list and exactly once. From this behaviour and the previous information, the abstract sequence B_1 is inferred.

The second clause is treated similarly but contains a recursive call, which deserves a special treatment. First the analyser is able to infer that the recursive call will be executed at most once and, in fact, exactly once when L is of the

[4]A size measure, or norm, is a function $\|\cdot\| : \mathcal{T} \to \mathbb{N}$, where $\mathcal{T}$ is the set of all terms. We refer here to the list-length measure. See Section 4.1 for the general norm definition.

form $[t_1 \mid t_2]$. It also infers that X and LS are distinct variables and that T is ground and strictly smaller than L. Thus, we can assume by induction that the recursive call satisfies the conditions provided by the user through the abstract sequence B. So the analyser deduces that the recursive call succeeds only if T is a *non-empty* list and that it returns a number of solution equal to the length of T; it also infers that X is ground and that TS is a ground list whose size is the same as the size of T minus 1. Putting all pieces of information together, the analyser computes the abstract sequence B_2, which states that the second clause succeeds only for a list L of at least two elements (and actually succeeds for all of them) and that the output size of LS is equal to the size of L minus 1, i.e., $\|\mathtt{L}\| - 1$; moreover, the number of solution is also equal to $\|\mathtt{L}\| - 1$.

The next step of the analyser is to combine the abstract sequences B_1 and B_2 to get a new abstract sequence B' describing the behaviour of the whole procedure. A careful analysis is once again necessary to get the most precise result: when L is a list of at least two elements, the first clause succeeds once and the second one succeeds $\|\mathtt{L}\| - 1$ times, so the procedure succeeds $\|\mathtt{L}\|$ times. However, when the length of L is equal to 1, the second clause fails and the first one succeeds once; so the procedure also succeeds $\|\mathtt{L}\|$ times (because $\|\mathtt{L}\| = 1$). Hence, putting the abstract sequences B_1 and B_2 together, the analyser is able to reconstruct exactly the information provided by the user, which is thus correct.

The previous discussion is intended to give insights into how all kinds of information interact to produce an accurate analysis. It may however suggest that the automatization of the process is straightforward; this is because we have used all notions without formalizing them. In the rest of this paper a complete formalization of the process is described.

Chapter 3

Concrete domains and concrete semantics

3.1 Concrete domains

The concrete domain is the set of values an object can take in the standard computation domain. Prolog is based on the handling of substitutions which are defined on sets of variables and correspond to a set of assignment of terms to variables.

In this Section, we recall some terminology used for the basic concepts of logic programming, that define the concrete domains of Prolog programs. Note that we assume a preliminary knowledge of logic programming; see for instance [1], [2] or [3].

Variables and Terms. We assume the existence of two disjoint and infinite sets of variables, denoted by PV and SV. Elements of PV are called *program variables*[1] and are denoted by X_1, X_2, ..., X_i, ... The set PV is totally ordered; X_i is the i-th element of PV. Elements of SV are called *standard variables* and are denoted by letters y and z (possibly subscripted). Terms are built using standard variables only.

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 $\{X_{i_1}, \ldots, X_{i_n}\}$. We denote by PS_D the set of program substitutions whose domain is D. A *standard substitution* σ is a substitution in the usual sense which only uses standard variables. The application of a

[1]Program variables are those used in the clauses.

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 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.

Programs are assumed to be in a normalized form (see Appendix C.1).

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 $dom(\theta)$ is the set $\{X_1, \ldots, X_n\}$, 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 $\langle g, c \rangle$.

The concrete semantics of terminating executions is characterized by the set of transition rules:

$$R1: \frac{pr \text{ defines } p \text{ in } P \quad \langle \theta, pr \rangle \longmapsto S}{\langle \theta, p \rangle \longmapsto S} \qquad R2: \frac{pr ::= c \quad \langle \theta, c \rangle \longmapsto S}{\langle \theta, pr \rangle \longmapsto S} \qquad R3: \frac{pr ::= c, pr' \quad \langle \theta, c \rangle \longmapsto S \quad \langle \theta, pr' \rangle \longmapsto S'}{\langle \theta, pr \rangle \longmapsto S :: S'}$$

$$R4: \frac{\begin{array}{c} c::=h: -g \\ \langle \theta, g, c\rangle \longmapsto S' \\ S = \texttt{RESTRC}(c, S') \end{array}}{\langle \theta, c\rangle \longmapsto S} \qquad R5: \frac{\begin{array}{c} g::= < > \\ S = \texttt{EXTC}(c, \theta) \end{array}}{\langle \theta, g, c\rangle \longmapsto S}$$

$$R6: \frac{\begin{array}{c} g::=g', l \\ l::=X_{i_1} = X_{i_2} \\ \langle \theta, g', c\rangle \longmapsto S \\ S =< \theta_1, \ldots, \theta_n > \\ \theta'_k = \texttt{RESTRG}(l, \theta_k) \\ S'_k = \texttt{UNIF\_VAR}(\theta'_k) \\ S_k = \texttt{EXTG}(l, \theta_k, S'_k) \end{array}}{\langle \theta, g, c\rangle \longmapsto S_1 :: \ldots :: S_n}$$

$$R7: \frac{\begin{array}{c} g::=g', l \\ l::=X_{i_1} = f(X_{i_2}, \ldots, X_{i_n}) \\ \langle \theta, g', c\rangle \longmapsto S \\ S =< \theta_1, \ldots, \theta_n > \\ \theta'_k = \texttt{RESTRG}(l, \theta_k) \\ S'_k = \texttt{UNIF\_FUNC}(\theta'_k, f) \\ S_k = \texttt{EXTG}(l, \theta_k, S'_k) \end{array}}{\langle \theta, g, c\rangle \longmapsto S_1 :: \ldots :: S_n} \qquad R8: \frac{\begin{array}{c} g::=g', l \\ l::=p(X_{i_1}, \ldots, X_{i_n}) \\ \langle \theta, g', c\rangle \longmapsto S \\ S =< \theta_1, \ldots, \theta_n > \\ \theta'_k = \texttt{RESTRG}(l, \theta_k) \\ \langle \theta'_k, p\rangle \longmapsto S'_k \\ S_k = \texttt{EXTG}(l, \theta_k, S'_k) \end{array}}{\langle \theta, g, c\rangle \longmapsto S_1 :: \ldots :: S_n}$$

We briefly discuss the meaning of the rules above.

If pr is the procedure defining p in P, then the result of executing $p(X_1, \ldots, X_n)\theta$ is obtained by executing pr with the call substitution θ. This is expressed by Rule $R1$.

Rules $R2$ and $R3$ state, respectively, that the execution of a procedure pr consisting of one clause c is simply equal to the execution of c, whereas, in general, it is obtained by concatenating the results of the clauses belonging to the procedure.

Rule $R4$ describes the execution of a clause c called with θ. In this case, the body g of c is executed with θ returning the sequence S', and the domain of S' is restricted to the set of variables $\{X_1, \ldots, X_n\}$ in the head of c. The restriction is realized by the RESTRC operation defined by: $\texttt{RESTRC}(c, S') = S$ where $S = S'_{/\{X_1, \ldots, X_n\}}$.

The execution of the empty body g of c with θ (rule $R5$) returns a one-element sequence S obtained by extending θ to all the variables $\{X_1, \ldots, X_m\}$ $(m \geq n)$ occurring in c. Formally, $S = \texttt{EXTC}(c, \theta)$ is of the form $< \theta' >$ such that $X_i\theta' = X_i\theta$ $(1 \leq i \leq n)$ and $X_{n+1}\theta', \ldots, X_m\theta'$ are new distinct standard variables.

In the general case, suppose that g is of the form g', l. We distinguish three cases depending on the form of l. They are illustrated by rules $R6$, $R7$ and $R8$,

respectively. In all the cases, the following steps are performed.

- The prefix[2] g' of c is computed returning the sequence of answers S.
- The domain of each $\theta_k \in Subst(S)$ is restricted to the variables $X_{i_1}, \ldots, X_{i_n}$ of l and renamed into $X_1, \ldots, X_n$ returning the substitution θ'_k; this is done by applying the operation RESTRG formally defined by: $\texttt{RESTRG}(l, \theta_k) = \theta'_k$ where $\theta'_k = \{X_1/X_{i_1}\theta_k, \ldots, X_n/X_{i_n}\theta_k\}$.
- The literal l is executed with all the θ'_k returning for each of them a sequence of answers S'_k.
- The results in S'_k are propagated to θ_k with the EXTG operation defined by: if S'_k is of the form $< \theta'_k\sigma_1, \ldots, \theta'_k\sigma_n >$ with $dom(\sigma_i) \subseteq codom(\theta'_k)$, then $\texttt{EXTG}(l, \theta_k, S'_k) = S_k$ where $S_k =< \theta_k\sigma_1, \ldots, \theta_k\sigma_n >$.
- The sequences S_k are concatenated.

The execution of the built-in $X_{i_1} = X_{i_2}$ (rule $R6$) is realized by the operation UNIF_VAR defined by: for all θ with $dom(\theta) = \{X_1, X_2\}$,

$$\texttt{UNIF\_VAR}(\theta) \quad = \quad \begin{cases} <> & \text{if } \emptyset = mgu(X_1\theta, X_2\theta) \\ < \theta\sigma > & \text{if } \sigma \in mgu(X_1\theta, X_2\theta) \end{cases}$$

The execution of the built-in $X_{i_1} = f(X_{i_2}, \ldots, X_{i_n})$ (rule $R7$) is realized by the operation UNIF_FUNC defined by: for all θ with $dom(\theta) = \{X_1, \ldots, X_n\}$ and all functor f of arity $n-1$,

$$\texttt{UNIF\_FUNC}(\theta, f) \quad = \quad \begin{cases} <> & \text{if } \emptyset = mgu(X_1\theta, f(X_2, \ldots, X_n)\theta) \\ < \theta\sigma > & \text{if } \sigma \in mgu(X_1\theta, f(X_2, \ldots, X_n)\theta) \end{cases}$$

The result of the execution of a procedure call $p(X_{i_1}, \ldots, X_{i_n})$ is simply obtained by applying the function $\longmapsto$ which is indeed the concrete semantics and specifies the result of the execution of any procedure call.

[2] In this context, g' is the goal obtained by removing the last literal of the goal g.

Chapter 4

Abstract domains

The domains are a fundamental notion in abstract interpretation. We have to abstract the concrete domain on which the language is based in order to obtain an abstract domain. Following different parameters, we deduce an abstract domain which fits the best what we need. All the calculations are related to this abstract domain and the results, as well as the accuracy and efficiency, depend on this choice.

In this Chapter we describe in great detail the abstract domains. It consists of the ones described in [14]. Section 4.1 introduces some objects we need to express our abstract domains (like norms, disjoint unions, linear expressions,...). Section 4.2 shows our domain of abstract substitutions. Section 4.3 presents our domain of abstract sequences. Finally, Section 4.4 defines the notion of behaviour, which formalizes the notion of formal specification (whose concrete syntax can be found in Appendix A.3), i.e., the full package of information provided (for verification) by the user to the system.

For each abstract domain, we always give its *definition* (i.e., we show the (abstract) objects it contains and its (pre)order denoted $\leq$), and also its *semantics*, i.e., we give the related concretization function

$$Cc : ABSTRACT\_DOMAIN \rightarrow \wp(CONCRETE\_DOMAIN).$$

In Chapters 6, 7 and 8 we will describe the abstract operations applying on these abstract domains.

4.1 Preliminaries

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 $\mathbf{N}$. We denote by I_p ($p \geq 0$) the set of indices $\{1, \ldots, p\}$. $\mathcal{T}^I$ 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 $\|\cdot\| : \mathcal{T} \rightarrow \mathbf{N}$. We refer to the list-length measure for terms that have a list pattern:

$$\begin{array}{rcll} \|[\,]\| & = & 0 & \\ \|[t_1|t_2]\| & = & 1 + \|t_2\| & \text{where } t_2 \text{ has a list pattern} \end{array}$$

For all terms that have another pattern, we always refer to this general measure:

$$\begin{array}{rcl} \|X\| & = & 0 \\ \|f(t_1, \ldots, t_n)\| & = & 1 + \|t_1\| + \ldots + \|t_n\| \end{array}$$

Disjoint Unions. 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 $f_A : A \rightarrow C$ and $f_B : B \rightarrow C$, there exists a unique function $f : A+B \rightarrow C$ 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 $f_A + f_B$. The functions in_A, in_B, f_A, f_B, and $f_A + f_B$ satisfy the commutative diagram depicted in Fig.4.1.

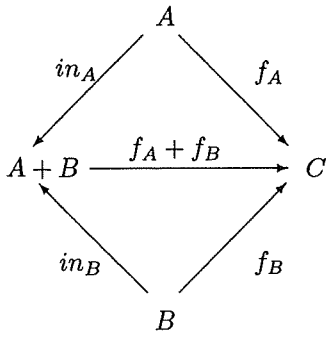


Figure 4.1: Disjoint Union $f_A + f_B$.

Linear Expressions. Let V be a set of variables. We denote by $\mathbf{Exp}_V$ the set of all linear expressions with integer coefficients on the set of variables V. An element $se \in \mathbf{Exp}_{\{X_1,\ldots,X_m\}}$ can also be seen as a function from $\mathbf{N}^m$ to $\mathbf{N}$, 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.

4.2 Abstract Substitutions

The domain of abstract substitutions is an instantiation to modes, types and possible sharing of the generic abstract domain Pat($\Re$) described in [7].

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 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 below.

In this Section we describe the various components and then turn to the ordering of the abstract substitutions.

4.2.1 The domain $\mathtt{SV}_{D,I_m}$

Definition:

[*same-value* component]. This domain assigns a subterm to each variable in the substitution. Given a set of program variables $D = \{X_1, \ldots, X_n\}$ and a set of indices I_m, we denote by SV_{D,I_m} the set of all surjective functions from D to I_m.

Semantics:

The semantics of an element $sv \in SV_{D,I_m}$ is given by the following concretization function $Cc : SV_{D,I_m} \rightarrow \wp(PS_D)$, that makes sure that two variables assigned to the same index have the same value:

$$Cc(sv) = \{\theta \mid dom(\theta) = D \; and \; \forall x_i, x_j \in D : sv(x_i) = sv(x_j) \Rightarrow x_i\theta = x_j\theta\}.$$

4.2.2 The domain $\mathtt{FRM}_{I_p}$

Definition:

[*frame* or *pattern* component]. This domain associates with some of the indices in I_p an expression $f(i_1, \ldots, i_q)$, where f is a functor symbol of arity q and $\{i_1, \ldots, i_q\} \subseteq I_p$. It is then a set of partial functions from I_p to $T_{I_p}^*$

$(frm : I_p \not\rightarrow \mathcal{T}^*_{I_p})$. We denote the fact that no frame is associated with i by $frm(i) = undef$.

Semantics:

The meaning of an element $frm \in FRM_{I_p}$ is given by the following concretization function $Cc : FRM_{I_p} \rightarrow \wp(T^{I_p})$, that specifies that the component represents all p-tuples of terms that satisfy simultaneously all pattern constraints:

$$\begin{aligned} Cc(frm) \quad = \quad & \{\langle t_i \rangle_{i \in I_p} \in T^{I_p} \mid \forall i, i_1, \ldots, i_q \in I_p : \\ & frm(i) = f(i_1, \ldots, i_q) \Rightarrow t_i = f(t_{i_1}, \ldots, t_{i_q})\}. \end{aligned}$$

4.2.3 The domain Modes

Definition:

We consider the set of modes

$$Modes = \{\bot, ground, var, ngv, novar, gv, noground, any\},$$

satisfying the ordering relationship implied by the diagram depicted in Figure 4.2, where an arc between M_1 and M_2 with M_1 above M_2 means that $M_1 > M_2$ [1].

Semantics:

The semantics of modes can be given by the following concretization function $Cc : Modes \rightarrow \wp(T)$:

$Cc(\bot)$	=	$\emptyset$;
$Cc(ground)$	=	$\{t \in T \mid t$ is a ground term$\}$;
$Cc(var)$	=	$\{t \in T \mid t$ is a variable$\}$;
$Cc(gv)$	=	$\{t \in T \mid t$ is either a ground term or a variable$\}$;
$Cc(noground)$	=	$\{t \in T \mid t$ is not a ground term$\}$;
$Cc(novar)$	=	$\{t \in T \mid t$ is not a variable$\}$;
$Cc(ngv)$	=	$\{t \in T \mid t$ is neither a ground term nor a variable$\}$;
$Cc(any)$	=	T;
$Cc(\mathtt{LUB}(M_1, M_2))$	=	$Cc(M_1) \cup Cc(M_2)$. [Least Upper Bound];
$Cc(\mathtt{GLB}(M_1, M_2))$	=	$Cc(M_1) \cap Cc(M_2)$. [Greatest Lower Bound].

We recall the definitions of the least upper bound and of the greatest lower bound:

$$\mathtt{LUB}(M_1, M_2) = M_1 \sqcup M_2 = M \mid M = min_{\leq}\{M_j \in Modes : M_1, M_2 \leq M_j\}.$$

$$\mathtt{GLB}(M_1, M_2) = M_1 \sqcap M_2 = M \mid M = max_{\leq}\{M_j \in Modes : M_1, M_2 \geq M_j\}.$$

[1] Note that $(M_1 \leq M_2) \neq \neg(M_1 > M_2)$.

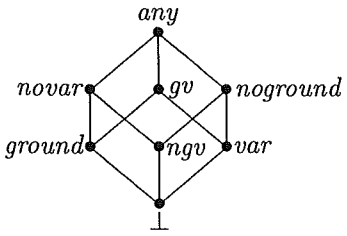


Figure 4.2: Ordering of modes as a Hasse diagram.

4.2.4 The domain $\mathtt{Modes}_{I_p}$

Definition:
For any set of indices I_p, we denote by $Modes_{I_p}$ the set of all functions from I_p to *Modes* augmented with $\bot$.

Semantics:
The semantics of an element $mo \in Modes_{I_p}$ is given by the following concretization function $Cc : Modes_{I_p} \rightarrow \wp(T^{I_p})$. If $mo = \bot$ then $Cc(mo) = \emptyset$, otherwise $Cc(mo)$ is the set $\{\langle t_i \rangle_{i \in I_p} \in T^{I_p} | \, \forall i \in I_p : t_i \in Cc(mo(i))\}$.

4.2.5 The domain Types

Definition:
The following type domain for lists is considered:

$$Types = \{\bot, list, anylist, any\},$$

ordered by: $\bot \leq list \leq anylist \leq any$ [2].

Semantics:
The semantics of types can be given by the following concretization function $Cc : Types \rightarrow \wp(T)$:

$$\begin{array}{lcl} Cc(\bot) & = & \emptyset; \\ Cc(list) & = & \{t \in T | \; t \text{ is a list}\}; \\ Cc(anylist) & = & \{t \in T | \; t \text{ is a term that can be instantiated to a list}\}; \\ Cc(any) & = & T; \\ Cc(\mathtt{LUB}(T_1, T_2)) & = & Cc(T_1) \cup Cc(T_2). \text{ [Least Upper Bound]}; \\ Cc(\mathtt{GLB}(T_1, T_2)) & = & Cc(T_1) \cap Cc(T_2). \text{ [Greatest Lower Bound]}. \end{array}$$

Be careful with the definition of *list* and *anylist*.

By type *list*, we consider this recursive definition:

[ ] has type *list*
$[t_1 \mid t_2]$ has type *list* if t_2 has type *list*

[2] Note that $(T_1 \leq T_2) = \neg(T_1 > T_2)$.

By type *anylist*, we consider this recursive definition:

t has type *anylist* if t has type *list*
$[t_1 \mid t_2]$ has type *anylist* if t_2 has type *anylist*
X has type *anylist* where X is a variable

Note that by definition of our Types semantics, a term that has mode *var* automatically has the type *anylist* (of course, it can also have the type *list* if specified), because a variable can be instantiated to a list.

Also, any *ground* term has either the type *list* or the type *any*. Indeed, a *ground* term that has the type *anylist* is in fact already instantiated to a list: it is the reason why every *ground anylist* has to be replaced by *ground list*.

We recall the definitions of the least upper bound and of the greatest lower bound:

$$\mathtt{LUB}(T_1, T_2) = T_1 \sqcup T_2 = T \mid T = min_{\leq}\{T_j \in Types : T_1, T_2 \leq T_j\}.$$

$$\mathtt{GLB}(T_1, T_2) = T_1 \sqcap T_2 = T \mid T = max_{\leq}\{T_j \in Types : T_1, T_2 \geq T_j\}.$$

4.2.6 The domain $\mathtt{Types}_{I_p}$

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

4.2.7 The domain $\mathtt{PSharing}_{I_p}$

Definition:
This domain specifies possible variable sharing between terms. For any set of indices I_p, we denote by $PSharing_{I_p}$ the set of all binary and symmetrical relations $ps \subseteq I_p \times I_p$ augmented with $\bot$.
Semantics:
The semantics of an element $ps \in PSharing_{I_p}$ is given by the following concretization function $Cc : PSharing_{I_p} \rightarrow \wp(T^{I_p})$. If $ps = \bot$ then $Cc(ps) = \emptyset$, otherwise

$$Cc(ps) = \{\langle t_i \rangle_{i \in I_p} \in T^{I_p} \mid \forall i, j \in I_p : \ Var(t_i) \cap Var(t_j) \neq \emptyset \Rightarrow (i, j) \in ps\}.$$

For the implementation, we don't want to store all this information, because a lot of this can be found in the *frame* component $frm \in FRM_{I_p}$. It is the reason why we only store in ps the pairs of terms that possibly share variables and whose patterns are undefined. So, ps should satisfy the property

$$\forall i, j : 1 \leq i, j \leq p : ps(i, j) \Rightarrow frm(i) = frm(j) = undef.$$

Together with the *frame* component (frm), ps allows us to deduce the actual sharing relation (that will be noted ps^*, in order to distinguish it with the stored information ps). It is defined as the smallest relation on I_p satisfying the following two rules for all $i, j, k \in I_p$:

(1) $ps(i, j) \Rightarrow ps^*(i, j)$;
(2) $frm(k) = f(\ldots, j, \ldots)$ & $ps^*(i, j) \Rightarrow ps^*(k, i)$.

So at the implementation level, it is more efficient if we only store ps and if we recompute ps^* when necessary.

4.2.8 The domain of `Abstract Tuples`

Definition:

The component of abstract substitutions that gives information about the modes, types and possible sharing of the terms is called the *abstract tuple*. In the `Pat`($\Re$) terminology [7], it corresponds to (is an instantiation of) the so-called $\Re$-component or $\Re$-domain.

An *abstract tuple* α over a set of indices I_p is either $\bot$ or a triplet of the form $\langle mo, ty, ps \rangle$ where $mo \in Modes_{I_p}$, $ty \in Types_{I_p}$ and $ps \in PSharing_{I_p}$, with $mo, ty, ps \neq \bot$ and for all $i \in I_p$, $mo(i), ty(i) \neq \bot$.

We can write `Abstract Tuples` $= \texttt{Modes}_{I_p} \times \texttt{Types}_{I_p} \times \texttt{PSharing}_{I_p}$.

Semantics:

The *semantics of an abstract tuple* α over I_p is given by the following concretization function $Cc : AbstractTuples \rightarrow \wp(T^{I_p})$.

If $\alpha = \bot$ then $Cc(\alpha) = \emptyset$, otherwise $Cc(\alpha) = Cc(mo) \cap Cc(ty) \cap Cc(ps)$.

4.2.9 The domain of Abstract Substitutions $\texttt{AS}_D$

The *semi-generic pattern domain* `Pat`($\Re$) automatically upgrades a domain `D` with structural information yielding a more accurate domain `Pat(D)`.

The key idea behind `Pat`($\Re$) is to provide a generic implementation of the abstract operations of `Pat(D)` in terms of a few basic operations on the domain `D`.

The main advantages of this approach are the simplicity, modularity, and accuracy it offers to abstract domain designers. *Simplicity* is achieved by abstracting away any structural information and allowing designers to focus at one domain at a time. *Modularity* comes from the fact that abstract domains can be viewed

as abstract data types simplifying both the correctness proofs and the implementation. Finally, *accuracy* results from structural information and from the idea of open operation which is so general that abstract domains can interact at will although through well-defined interfaces.

Our abstract substitution domain is an instantiation of $\mathtt{Pat}(\Re)$, namely `Pat(Abstract Tuples)`:

Definition: [*(pseudo-) abstract substitution*]
A(n) (pseudo-) abstract substitution β over a set of indices I_p (the latter contains a subset I_m, with $m \leq p$) is either $\bot$ or a triplet of the form $\langle sv, frm, \alpha \rangle$ where $sv \in SV_{D,I_m}$; $frm \in FRM_{I_p}$ and α is an *abstract tuple* over I_p.
The set of variables $D = \{X_1, \ldots, X_n\}$ is called the domain of β and is denoted by $dom(\beta)$. The set of abstract substitutions with domain D is denoted $\mathtt{AS}_D$.
We will often represent in this way an abstract substitution where the α tuple is explicit: $\langle sv, frm, \langle mo, ty, ps \rangle \rangle$.
In the following, we always assume, unless specified otherwise, and for the convenience of implementation, that the abstract substitution $\beta = \langle sv, frm, \langle mo, ty, ps \rangle \rangle$ is such that $sv \in SV_{D,I_m}$, $frm \in FRM_{I_p}$, $mo \in Modes_{I_p}$, $ty \in Types_{I_p}$, $ps \in PSharing_{I_p}$. m is the number of indices in the codomain of sv and p is the number of indices in the domains of frm, mo, ty and ps. Similarly β' (resp. β_i) is defined according to m' and p' (resp. m_i and p_i).

Semantics:
The semantics of such an abstract substitution β is given by the following concretization function $Cc : \mathtt{AS}_D \rightarrow \wp(\mathtt{PS}_D)$.

If $\beta = \bot$ then $Cc(\beta) = \emptyset$, otherwise

$$\begin{aligned} Cc(\beta) \quad = \quad & \{\theta \mid dom(\theta) = D \text{ and } \exists\, \langle t_i \rangle_{i \in I_p} \in \mathcal{T}^{I_p} : \\ & \langle t_i \rangle_{i \in I_p} \in Cc(frm) \cap Cc(\alpha); \\ & \forall X \in D,\ X\theta = t_{sv(X)} \}. \end{aligned}$$

Note that the set of indices assigned to the variables by the *same-value* component does not cover all of the indices used by the other components of the domains (e.g., if a term has a pattern then our domain will keep some information about its subterms). This is expressed by the fact that $m \leq p$ where m is the codomain of sv. On the other hand, if several variables are assigned the same value (e.g., after unification) it may be the case that the number of indices used in this component is smaller than the number of variables (this is expressed by $m \leq n$).

In fact we use here the *pseudo*-version of the definition of this abstract object which is simpler to manipulate. The corresponding *strict*-version is endowed with further conditions to prevent from incorrect and redundant representation.

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

Definition:
Let I be a set of indices, $sv : \{X_1, \ldots, X_n\} \to I$ be a function and $frm : I \not\to T_I^*$ 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 $\preceq\!\!\!\!\!\!\prec_{frm}$ 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 $sv(X_k) \preceq\!\!\!\!\!\!\prec_{frm} i$.

Definition: [*strict-abstract substitution*]
A strict-abstract substitution β over I_p is a (pseudo-) abstract substitution $\langle sv, frm, \alpha \rangle$ over I_p such that

- $\alpha \neq \bot$;
- *frm* is circuit-free;
- all $i \in I_p$ are reachable by *sv* and *frm*;
- for all $i, j \in I_p$ such that $frm(i) = f(i_1, \ldots, i_n)$ and $(j, i_k) \in ps$ for some $k \in \{1, \ldots, n\}$, $(j, i) \in ps^*$.

Note that the latest condition will be satisfied because we took the convention to only store (in *ps*) the possible sharing information between terms whose pattern are undefined. The total possible sharing (ps^*) is computed on *ps* such that the above condition holds.

At the implementation level, times to times, we can imagine to call a "normalization procedure" that takes a pseudo abstract substitution and returns its strict version. Indeed this can improve the accuracy of the algorithms which perform some abstract substitutions comparisons.

4.2.10 Decomposition (DECOMP)

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

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

- $\theta = \{X_1/t_{sv(X_1)}, \ldots, X_n/t_{sv(X_n)}\}$;

- $\forall i \in I_p,\ frm(i) = f(i_1, \ldots, i_n) \Rightarrow t_i = f(t_{i_1}, \ldots, t_{i_n})$;
- $\langle t_i \rangle_{i \in I_p} \in Cc(\alpha)$.

4.2.11 Pre-Ordering on Abstract Substitutions

Definition:
Let β_1 and β_2 be two abstract substitutions on the same domain D (i.e., $\beta_1, \beta_2 \in \mathtt{AS}_D$). $\beta_1 \leq \beta_2$ iff there exists a function $t : I_{p_2} \rightarrow I_{p_1}$ satisfying

(1) $\forall x \in D : sv_1(x) = t(sv_2(x))$;
(2) $\forall i, i_1, \ldots, i_q \in I_{p_2}$:
$\quad frm_2(i) = f(i_1, \ldots, i_q) \Rightarrow frm_1(t(i)) = f(t(i_1), \ldots, t(i_q))$;
(3) $\forall i \in I_{p_2} : mo_1(t(i)) \leq mo_2(i)$;
(4) $\forall i \in I_{p_2} : ty_1(t(i)) \leq ty_2(i)$;
(5) $\forall i, j \in I_{p_2} : frm_2(i) = frm_2(j) = undef : ps_1^*(t(i), t(j)) \Rightarrow ps_2(i, j)$.

Note in fact that this relation $\leq$ is only a preorder: two distinct substitutions can represent the same set of actual substitutions simply by permuting some indices. It is not difficult to obtain an ordering relation by considering equivalence classes. However, for all practical purposes, this is not really necessary, and we will continue to work with that domain.

Semantics:
Conceptually, $\beta_1 \leq \beta_2$ should hold iff β_1 imposes the same as or more constraints on all components than β_2 does, that is, iff $Cc(\beta_1) \subseteq Cc(\beta_2)$.

4.2.12 Structural mapping on Abstract Substitutions

A structural mapping between two abstract substitutions is a mapping on the corresponding indices preserving same-value and frame. Let $\beta = \langle sv, frm, \alpha \rangle$ and $\beta' = \langle sv', frm', \alpha' \rangle$ be two abstract substitutions over I_p and $I_{p'}$, respectively. A *structural mapping* between β and β' (if it exists) is a function $tr : I_p \rightarrow I_{p'}$ such that

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

4.3 Abstract Sequences

4.3.1 The domain $\mathtt{Sizes}_{I_p}$

Definition:
We denote by $Sizes_{I_p}$ the set of all systems of linear equations and inequations over $\mathbf{Exp}_{I_p}$, extended with the special symbol $\perp$. In order to distinguish indices of I_p, considered as variables, from integer coefficient and constants

when writing elements of $\mathbf{Exp}_{I_p}$, we wrap up each element i of I_p into the symbol $\mathtt{sz}(i)$.

Semantics:

$Sizes_{I_p}$ is endowed with a concretization function $Cc : Sizes_{I_p} \rightarrow \wp(\mathbf{N}^{I_p})$. For all $E \in Sizes_{I_p}$, if $E = \bot$ then $Cc(E) = \emptyset$, otherwise,

$$Cc(E) = \{\langle n_i \rangle_{i \in I_p} \in \mathbf{N}^{I_p} |\ \langle n_i \rangle_{i \in I_p} \text{ is a solution of } E\}.$$

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 use the notion of disjoint union[3] allowing us to "merge" two sets of indices into one set, in such a way 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).

4.3.2 The domain of Abstract Sequences $\mathtt{ASS}_D$

The analyzer is built upon the notion of *abstract sequence*. Abstract sequences describe pairs $\langle \theta, S \rangle$ where θ is a substitution and S is the sequence of answer substitutions resulting from executing a program (a procedure, a clause, etc.) with input substitution θ.

Definition: [*(pseudo-) abstract sequence*]

A (pseudo-) abstract sequence B is either $\bot$ or a tuple of the form $\langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$ where

- β_{in} is an abstract substitution over I_{in} (describing the class of accepted input calls);
- β_{ref} is an abstract substitution over I_{ref} with $dom(\beta_{ref}) = dom(\beta_{in})$ (describing an over approximation of the successful input calls, i.e., those that produce at least one solution);
- β_{out} is an abstract substitution over I_{out} with $dom(\beta_{out}) \supseteq dom(\beta_{in})$ (describing an over approximation of the set of outputs corresponding to the successful calls);
- $E_{ref\_out} \in Sizes_{(I_{ref}+I_{out})}$ (describing a relation between the size of the terms of a successful call and the size of the terms returned by the calls);
- $E_{sol} \in Sizes_{(I_{ref}+\{sol\})}$ (describing a relation between the size of the terms occurring in a successful call and the number of solutions returned by the call).

[3] See Section 4.1 where the notion of disjoint union is formally explained.

At the view of the disjoint union $I_{ref} + I_{out}$, two injections are implicit: $in_{ref} : I_{ref} \rightarrow I_{ref} + I_{out}$ and $in_{out} : I_{out} \rightarrow I_{ref} + I_{out}$.
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 $dom_{out}(B) = dom(\beta_{out})$. The set of abstract sequences with the same $dom_{in} = D$ is denoted $\mathtt{ASS}_D$.

Semantics:
The semantics of an abstract sequence B is given by the following concretization function $Cc : \mathtt{ASS}_D \rightarrow \wp(PS_D \times SSeq)$: if $B = \bot$ then $Cc(B) = \emptyset$, otherwise[4]

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

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 (indeed, the assertion is equivalent to $\theta \notin Cc(\beta_{ref}) \Rightarrow S = \langle\ \rangle$); 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: [*strict-abstract sequence*]
A strict-abstract sequence B is a (pseudo-) abstract sequence $\langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol}\rangle$ such that

- $\beta_{in} \neq \bot$;
- $\beta_{ref} \leq \beta_{in}$;
- for all $\theta' \in Cc(\beta_{out})$, $\exists \theta \in Cc(\beta_{ref})$ such that $\theta'_{/dom(\beta_{ref})} \leq \theta$;
- if either β_{ref} or β_{out} or E_{ref_out} or E_{sol} is equal to $\bot$, then they are all equal to $\bot$.

As discussed for strict abstract substitutions, we can also here imagine a kind of "normalization procedure" that transforms a pseudo version of abstract sequence into its strict one.

[4]Notice that the + operator used below is the one that applies to functions, as defined in Section 4.1, since tuples $\langle \|t_i\|\rangle_{i\in I}$ actually are functions from I to N.

4.3.3 Pre-Ordering on Abstract Sequences

Definition:
Let B_1 and B_2 be two abstract sequences on the same domain D (i.e., $B_1, B_2 \in \mathtt{ASS}_D$). $B_1 \leq B_2$ iff:

$$\begin{array}{ll} (1) & \beta^1_{in} = \beta^2_{in} \\ (2) & \beta^1_{ref} \leq \beta^2_{ref} \\ (3) & \beta^1_{out} \leq \beta^2_{out} \\ (4) & E^1_{ref\_out} \Rightarrow E^2_{ref\_out} \\ (5) & E^1_{sol} \Rightarrow E^2_{sol} \end{array}$$

Note that the implications ($\Rightarrow$) between sets of constraints in the conditions (4) and (5) have to be understood taking into account the structural mappings (if they exist) between the indices of β^1_{ref} and β^2_{ref}, and between the indices of β^1_{out} and β^2_{out}.

Semantics:
Conceptually, $B_1 \leq B_2$ should hold iff B_1 imposes the same as or more constraints on all components than B_2 does, that is, iff $Cc(B_1) \subseteq Cc(B_2)$.

4.4 Behaviours

4.4.1 Definition of a behaviour

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

A *behaviour* Beh_p for a procedure name $p \in \mathcal{P}$[5] 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[6] 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.

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 $p \in \mathcal{P}$.

[5] $\mathcal{P}$ is the set of all procedure names occurring in the analyzed program.

[6] 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.

4.4.2 A sample behaviour

Consider the formal specification for `select/3` informally described in Section 2.3 (page 11). The related behaviour $Beh_{select/3}$ is $\{\langle B, se\rangle\}$ where

$B = \langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol}\rangle$;
$se = \mathrm{L}$.

and where

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

sv_{in} :		frm_{in} :	mo_{in} :	ty_{in} :
X $\mapsto$ 1		1 $\mapsto$?	1 $\mapsto$ *var*	1 $\mapsto$ *anylist*
L $\mapsto$ 2		2 $\mapsto$?	2 $\mapsto$ *ground*	2 $\mapsto$ *any*
LS $\mapsto$ 3		3 $\mapsto$?	3 $\mapsto$ *var*	3 $\mapsto$ *anylist*

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

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

sv_{ref} :	frm_{ref} :	mo_{ref} :	ty_{ref} :
X $\mapsto$ 1	1 $\mapsto$?	1 $\mapsto$ *var*	1 $\mapsto$ *anylist*
L $\mapsto$ 2	2 $\mapsto$ [4|5]	2 $\mapsto$ *ground*	2 $\mapsto$ *list*
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*

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

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

sv_{out} :	frm_{out} :	mo_{out} :	ty_{out} :
X $\mapsto$ 1	1 $\mapsto$?	1 $\mapsto$ *ground*	1 $\mapsto$ *any*
L $\mapsto$ 2	2 $\mapsto$ [4|5]	2 $\mapsto$ *ground*	2 $\mapsto$ *list*
LS $\mapsto$ 3	3 $\mapsto$?	3 $\mapsto$ *ground*	3 $\mapsto$ *list*
	4 $\mapsto$?	4 $\mapsto$ *ground*	4 $\mapsto$ *any*
	5 $\mapsto$?	5 $\mapsto$ *ground*	5 $\mapsto$ *list*

$ps_{out} = \emptyset$.

$in_{ref} = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 5\}$
$in_{out} = \{1 \mapsto 6, 2 \mapsto 7, 3 \mapsto 8, 4 \mapsto 9, 5 \mapsto 10\}$
$E_{ref\_out} = \{sz(8) = sz(5)\}$
$E_{sol} = \{sol = sz(5) + 1\}$

Chapter 5

Description of the analyser

In this Chapter, 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 analyzed 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 sufficient for the analyser to deduce that the program is consistent and terminates.

To simplify the presentation, we assume that the program we want to analyze 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 5.5.

5.1 Abstract Execution of a Prolog program

Our analyser is based on a standard verification technique: for a given program, it analyzes each procedure; for a given procedure, it analyzes each clause; for a given clause, it analyzes 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 the following next Sections, we describe how our analyser executes at the abstract level the clauses and the procedures of a given Prolog program. You can find the algorithm of the analyser in Appendix E.

5.2 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.

- $\texttt{EXTC}(c, \beta) = B$ 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.

- $\texttt{RESTRC}(c, B) = B'$ 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.

- $\texttt{RESTRG}(l, B) = \beta$ 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.

- $\texttt{EXTG}(l, B_1, B_2) = B$ 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 $\texttt{RESTRG}(l, B_1)$). The operation EXTG extends the effect of the instanciation on the whole sequence B_1 (taking into account necessary renaming to avoid name clashes). More formally, the abstract substitution B must satisfy the following property. For all $\langle \theta, < \theta_1, \ldots, \theta_n > \rangle \in Cc(B_1)$, if $\theta'_k = \texttt{RESTRG}(l, \theta_k)$, $\langle \theta'_k, S'_k \rangle \in Cc(B_2)$, and $S_k = \texttt{EXTG}(l, \theta_k, S'_k)$ for every $k \leq n$, then $\langle \theta, S_1 :: \ldots :: S_n \rangle \in Cc(B)$.

- $\texttt{LOOKUP}(\beta, p, SBeh) = (success, B_{out})$ 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 $\forall\langle\theta, S\rangle \in Cc(B), \forall\theta' \in Subst(S)$, $se(\langle\|X_{i_1}\theta'\|, \ldots, \|X_{i_n}\theta'\|\rangle) < se(\langle\|X_1\theta\|, \ldots, \|X_n\theta\|\rangle)$.

- 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.

- 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.

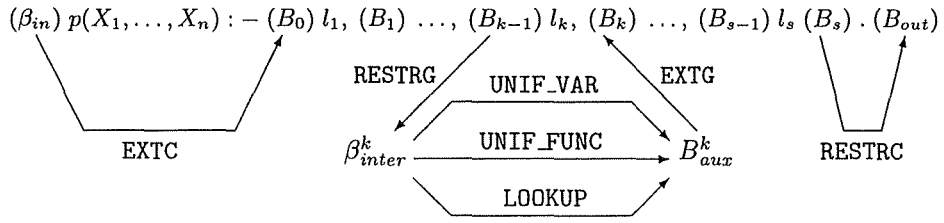
- CONC$(B_1, B_2) = B$ concatenates the abstract sequences B_1 and B_2 which must have the same input abstract substitution and the same output domain. The abstract sequence B must satisfy $\forall\langle\theta, S_1\rangle \in Cc(B_1), \forall\langle\theta, S_2\rangle \in Cc(B_2)$, $\langle\theta, S_1 :: S_2\rangle \in Cc(B)$.

5.3 Abstract Execution of a Clause

Let

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

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.



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

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

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

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 literals of the clause. Observe that, for each literal, 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 instantiation shares a variable with the instantiation of X_{i_1} or X_{i_2} will be correctly treated. Rule $R4$ follows a very similar process to execute function unification.

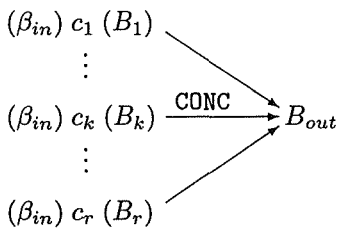
Rule $R5$ and $R5'$ execute procedure calls (either non-recursive or recursive). In the case of $R5$ (non-recursive call), the effect of the procedure call is obtained by searching $SBeh$ for a description of the procedure q. In the case of recursive calls, we impose that two conditions are satisfied: first, we only allow recursive calls that can be described by the behaviour currently analyzed ($\beta_{inter} \leq \beta_{in}$) 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 analyzed. 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.

5.4 Abstract Execution of a Procedure

Let $pr \equiv c_1, \ldots, c_r$ be a procedure whose name is p. Its abstract execution can be summarized by the following graph and rules.



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

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

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.

5.5 Removing the restrictions of the analyser

In this Section we explain 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 analyzed.

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 $p_1, \ldots, p_s$ (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 $p_1, \ldots, p_s$ 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 $se_q(\langle \|\theta' X_1\|, \ldots, \|\theta' X_m\| \rangle) < se_p(\langle \|\theta X_1\|, \ldots, \|\theta X_n\| \rangle)$, 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 analyze 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 analyzing a single procedure where recursive calls are described

by abstract sequences different from the one used as input, we analyze 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.

5.6 Design of the analyser

In the three next Chapters 6, 7 and 8, the interested reader can find a detailed description of all the abstract operations needed to build the analyser.

The Operations. Abstract operations are defined along several layers, as depicted in Figure 5.1. The Chapters corresponding to the layers are annotated.

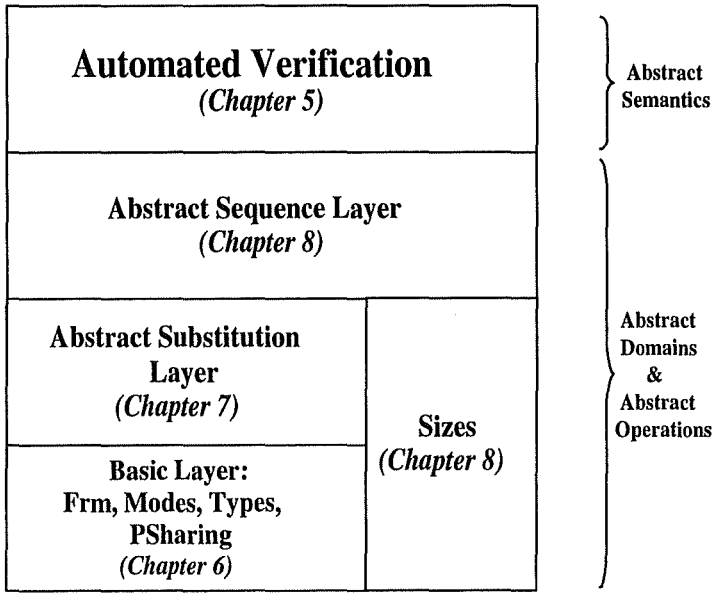


Figure 5.1: Abstract Operations Layers.

Implementation. We use the algorithms written for *GAIA* [8] to implement the operations acting on the abstract substitution domain, because this domain is quite similar to the one proposed in *Automated Verification* [14]. In order to reuse the existing algorithms with minor effort, we have "flatten" the abstract

substitution representation (abstract tuples are no more considered). Some of these algorithms were revisited to make them more accurate. Also, because the domain Types was not present in *GAIA*, we have entirely written new algorithms for that domain (limited to the lists). What about the abstract sequences operations, their implementation is based on the polyhedron library described in [15], in order to manipulate size information. We reuse the two abstract operations UNIF_VAR and CONC fully described in *Automated Verification* [14]. In this report, you will find also all the other abstract sequences operations that we have implemented. The interested reader can refer to the List of Operations (page xi) to relate those operations with the references of this report (mainly [7], [8], and [14]) .

Terminology. Each operation will be described by a *specification* and an *implementation* both written in a formal, mathematical way. The specification is the fundamental property that the implementation must satisfy and is always expressed relatively the standard (concrete) domain, using the concretization function denoted Cc.

We are now ready to go on with the implementation!

Chapter 6

Basic abstract operations

This Chapter presents the "low-level" abstract operations acting on $\mathtt{Frm}_{I_p}$, Modes, Types and $\mathtt{PSharing}_{I_p}$.

The operations on $\mathtt{FRM}_{I_p}$, Modes and $\mathtt{PSharing}_{I_p}$ were taken from *GAIA* [8]. The operations on Types are new ones.

6.1 Operations on $\mathtt{FRM}_{I_p}$

See Section 4.2.2 (page 21) where the domain $\mathtt{FRM}_{I_p}$ is defined.

6.1.1 Reachability (REACHABLE)

Operation 1. REACHABLE(i, frm)

Some operations need to compute the set of indices reachable from a given index, as the other indices are no longer relevant.
The set of indices reachable from $i \in I_p$ via frm, denoted REACHABLE(i, frm), is defined inductively by

(1) $\{i\}$ if $frm(i) = undef$
(2) $\{i\} \cup (\cup_{j=1}^{n}$ REACHABLE$(i_j, frm))$ if $frm(i) = f(i_1, \ldots, i_n)$.

We also use REACHABLE(I, frm) to denote $\cup_{i \in I}$ *reachable*(i, frm).

6.2 Operations on Modes

See Section 4.2.3 (page 22) where the domain Modes is defined.

6.2.1 Construction of Modes (CONS_MO)

Operation 2. $\texttt{CONS\_MO}(f, M_1, \ldots, M_n) = M'$
This operation computes the mode of a complex term from the modes of the subterms.

Specification:
Let $M_1, \ldots, M_n$ be modes, let $t_1, \ldots, t_n$ be terms, and let f be an n-arity functor symbol. This operation satisfies the following property:

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

Implementation:
Denote by C the condition $\exists i : 1 \leq i \leq n : M_i = \bot$. Then the CONS_MO operation can be implemented as follows:

$$M' = \begin{array}{ll} \bot & \text{if } C; \\ ground & \text{if } \forall i : 1 \leq i \leq n : M_i = ground; \\ ngv & \text{if } \neg C \text{ and } \exists i : 1 \leq i \leq n : M_i \leq noground; \\ novar & \text{otherwise.} \end{array}$$

6.2.2 Extraction of Modes (EXTR_MO)

Operation 3. $\texttt{EXTR\_MO}(f, M) = \langle M_1, \ldots, M_n \rangle$
This operation is the reverse of the CONS_MO operation. It 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. For instance, if $f(t_1, \ldots, t_n)$ is *ground*, it is clear that all of its arguments are *ground* as well.

Specification:
Let f be a functor symbol of arity n and $M \in Modes$. The operation satisfies the following property:

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

Implementation:
The operation is implemented as follows for $(1 \leq i \leq n)$:

$$M'_i = \begin{array}{ll} \bot & \text{if } M \in \{\bot, var\}; \\ ground & \text{if } M \in \{ground, gv\}; \\ noground & \text{if } M \in \{noground, ngv\} \text{ and } n = 1; \\ any & \text{otherwise.} \end{array}$$

6.2.3 Matching of Modes (MATCH_MO)

Operation 4. $\texttt{MATCH\_MO}(M, f, M_1, \ldots, M_n) = M'$
This operation recomputes the mode of a compound term when some of its subterms may have been instantiated (resulting in new subterms $t_1, \ldots, t_n$, with modes $M_1, \ldots, M_n$).

Specification:
Let $M, M_1, \ldots, M_n$ be modes, $t, t_1, \ldots, t_n$ be terms, and f be an n-ary functor symbol. The operation satisfies the following property:

$$\left.\begin{array}{r} t \in Cc(M) \\ \forall i : 1 \leq i \leq n : t_i \in Cc(M_i) \\ \exists \sigma : t\sigma = f(t_1, \ldots, t_n) \end{array}\right\} \Rightarrow \quad f(t_1, \ldots, t_n) \in Cc(M').$$

Implementation:
The implementation is $M' = \texttt{LUB}(M'_1, M'_2)$ where

M'_1	=	$\texttt{CONS\_MO}(f, M_1, \ldots, M_n)$	if $M \neq \bot$ and $M \neq ground$;
		$\bot$	otherwise.
M'_2	=	$ground$	if $M \geq ground$ and $\forall i : M_i \geq ground$;
		$\bot$	otherwise.

6.2.4 Abstract Unification of Modes (UAT_MO)

Operation 5. $\texttt{UAT\_MO}(M_1, M_2) = M'$
Given the modes of two terms, this operation returns the mode resulting of the unification of the terms.

Specification:
Let M_1, M_2 be modes, t_1, t_2 be terms, and σ be a standard substitution. The operation satisfies the following property:

$$\left.\begin{array}{r} t_1 \in Cc(M_1) \\ t_2 \in Cc(M_2) \\ \sigma \text{ is a mgu of } t_1, t_2 \end{array}\right\} \Rightarrow \quad t_1\sigma \in Cc(M').$$

Implementation:
The implementation is depicted in Table 6.1. As the operation is symmetric, M_1 and M_2 can be read vertically or horizontally.

	$\bot$	*var*	*ngv*	*ground*	*noground*	*gv*	*novar*	*any*
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
var	$\bot$	*var*	*ngv*	*ground*	*noground*	*gv*	*novar*	*any*
ngv	$\bot$	*ngv*	*novar*	*ground*	*novar*	*novar*	*novar*	*novar*
ground	$\bot$	*ground*	*ground*	*ground*	*ground*	*ground*	*ground*	*ground*
noground	$\bot$	*noground*	*novar*	*ground*	*any*	*any*	*novar*	*any*
gv	$\bot$	*gv*	*novar*	*ground*	*any*	*gv*	*novar*	*any*
novar	$\bot$	*novar*	*novar*	*ground*	*novar*	*novar*	*novar*	*novar*
any	$\bot$	*any*	*novar*	*ground*	*any*	*any*	*novar*	*any*

Table 6.1: Abstract Unification of Modes.

6.2.5 Abstract Instantiation of Modes (IAT_MO)

Operation 6. $\texttt{IAT\_MO}(M) = M'$

This operation computes the mode M' of a term whose mode was M and that has been arbitrarily instantiated.

Specification:

Let M be a mode. For any term t and standard substitution σ, the following holds:

$$t \in Cc(M) \Rightarrow t\sigma \in Cc(M').$$

Implementation:

The implementation is depicted in Table 6.2.

M	$\bot$	*var*	*ngv*	*ground*	*noground*	*gv*	*novar*	*any*
M'	$\bot$	*any*	*novar*	*ground*	*any*	*any*	*novar*	*any*

Table 6.2: Abstract Instantiation of Modes.

6.2.6 Reverse Abstract Instantiation of Modes (UNIST_MO)

Operation 7. $\texttt{UNIST\_MO}(M) = M'$

This operation is the reverse of the IAT_MO operation. 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:

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

Implementation:

The implementation is as follows:

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

6.2.7 Specialized Abstract Instantiation of Modes ($\texttt{IAT\_MO}_2$)

Operation 8. $\texttt{IAT\_MO}_2(M_1, M_2) = M'$

This operation computes the mode M' of a term whose mode was M_1 and that has since been instantiated according to a substitution of a variable by a term whose mode is M_2.

Specification:

Let M_1, M_2 be modes, t_1, t_2 be terms, and y be a variable. The operation satisfies the following property:

$$\left.\begin{array}{l} t_1 \in Cc(M_1) \\ t_2 \in Cc(M_2) \end{array}\right\} \Rightarrow t_1\{y \leftarrow t_2\} \in Cc(M').$$

Implementation:

The implementation is given in the form of Table 6.3. The values of the first argument are given vertically.

	$\bot$	*var*	*ngv*	*ground*	*noground*	*gv*	*novar*	*any*
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
var	$\bot$	*var*	*noground*	*gv*	*noground*	*gv*	*any*	*any*
ngv	$\bot$	*ngv*	*ngv*	*novar*	*ngv*	*novar*	*novar*	*novar*
ground	$\bot$	*ground*	*ground*	*ground*	*ground*	*ground*	*ground*	*ground*
noground	$\bot$	*noground*	*noground*	*any*	*noground*	*any*	*any*	*any*
gv	$\bot$	*gv*	*any*	*gv*	*any*	*gv*	*any*	*any*
novar	$\bot$	*novar*	*novar*	*novar*	*novar*	*novar*	*novar*	*novar*
any	$\bot$	*any*	*any*	*any*	*any*	*any*	*any*	*any*

Table 6.3: Specialized Abstract Instantiation of Modes: The values of the first argument are given vertically.

6.3 Operations on Types

See Section 4.2.5 (page 23) where the domain Types is defined.

The operations defined on the types are the same as the ones defined on the modes.

6.3.1 Construction of Types (CONS_TY)

Operation 9. $\texttt{CONS\_TY}(f, T_1, \ldots, T_n) = T'$

This operation computes the type of a complex term from the types of the subterms.

Specification:

Let $T_1, \ldots, T_n$ be types, let $t_1, \ldots, t_n$ be terms, and let f be an n-arity functor symbol. This operation satisfies the following property:

$$\forall i : 1 \leq i \leq n : t_i \in Cc(T_i) \Rightarrow f(t_1, \ldots, t_n) \in Cc(T').$$

Implementation:

Denote by C the condition $\exists i : 1 \leq i \leq n : T_i = \bot$. If C then the type result T' is $\bot$, otherwise the CONS_TY operation can be implemented as follows:

$$\begin{array}{lcl}
\texttt{CONS_TY}([\,], \,) & = & list \\
\texttt{CONS_TY}([\,|\,], T_1, list) & = & list \\
\texttt{CONS_TY}([\,|\,], T_1, anylist) & = & anylist \\
\texttt{CONS_TY}([\,|\,], T_1, any) & = & any \\
\texttt{CONS_TY}(f/n, T_1, \ldots, T_n) & = & any
\end{array}$$

6.3.2 Extraction of Types (EXTR_TY)

Operation 10. $\texttt{EXTR\_TY}(f, T) = \langle T_1, \ldots, T_n \rangle$

This operation is the reverse of the CONS_TY operation. It computes the most precise types of terms $t_1, \ldots, t_n$ when we know that the type of $f(t_1, \ldots, t_n)$ is T.

Specification:

Let f be a functor symbol of arity n and $T \in$ *Types*. This operation satisfies the following property:

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

Implementation:

The operation is implemented as follows:

$$\begin{array}{lcl}
\texttt{EXTR_TY}(f/n, \bot) & = & \langle \bot, \ldots, \bot \rangle \\
\texttt{EXTR_TY}([\,], list) & = & \langle\,\rangle \\
\texttt{EXTR_TY}([\,|\,], list) & = & \langle any, list \rangle \\
\texttt{EXTR_TY}([\,|\,], anylist) & = & \langle any, anylist \rangle \\
\texttt{EXTR_TY}([\,|\,], any) & = & \langle any, any \rangle \\
\texttt{EXTR_TY}(f/n, T) & = & \langle any, \ldots, any \rangle
\end{array}$$

6.3.3 Matching of Types (MATCH_TY)

Operation 11. $\texttt{MATCH\_TY}(T, f, T_1, \ldots, T_n) = T'$

This operation recomputes the type of a compound term when some of its subterms may have been instantiated (resulting in new subterms $t_1, \ldots, t_n$, with types $T_1, \ldots, T_n$).

Specification:

Let $T, T_1, \ldots, T_n$ be types, $t, t_1, \ldots, t_n$ be terms, and f be an n-ary functor symbol. The operation satisfies the following property:

$$\left.\begin{array}{r} t \in Cc(T) \\ \forall i : 1 \leq i \leq n : t_i \in Cc(T_i) \\ \exists \sigma : t\sigma = f(t_1, \ldots, t_n) \end{array}\right\} \Rightarrow \quad f(t_1, \ldots, t_n) \in Cc(T').$$

Implementation:

The operation is implemented as follows:
$\texttt{MATCH\_TY}(T, f, T_1, \ldots, T_n) = \texttt{CONS\_TY}(f, T_1, \ldots, T_n)$. We don't have to rely on T (that is the previous type of f). Indeed, the new type T' of a functor is entirely defined by its functor name f and by the new types of its subterms.

6.3.4 Abstract Unification of Types (UAT_TY)

Operation 12. $\texttt{UAT\_TY}(T_1, T_2) = T'$

Given the types of two terms, this operation returns the type resulting of the unification of the terms.

Specification:

Let T_1, T_2 be types, t_1, t_2 be terms, and σ be a standard substitution. The operation satisfies the following property:

$$\left.\begin{array}{r} t_1 \in Cc(T_1) \\ t_2 \in Cc(T_2) \\ \sigma \text{ is a mgu of } t_1, t_2 \end{array}\right\} \Rightarrow \quad t_1\sigma \in Cc(T').$$

Implementation:

The implementation is depicted in Table 6.4. As the operation is symmetric, T_1 and T_2 can be read vertically or horizontally.

Someone may wonder why the abstract unification between two *anylist* terms results in a term that has the type *any*. It is indeed the case: as an example, the unification between `[X,a|X]` and `[b,Y|Z]` (which are both of type *anylist*) gives `[b,a|b]` which is of type *any*.

6.3.5 Abstract Instantiation of Types (IAT_TY)

Operation 13. $\texttt{IAT\_TY}(T) = T'$

This operation computes the type T' of a term whose type was T and that has been arbitrarily instantiated.

	$\bot$	*list*	*anylist*	*any*
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
list	$\bot$	*list*	*list*	*list*
anylist	$\bot$	*list*	*any*	*any*
any	$\bot$	*list*	*any*	*any*

Table 6.4: Abstract Unification of Types.

Specification:
Let T be a type. For any term t and standard substitution σ, the following holds:

$$t \in Cc(T) \Rightarrow t\sigma \in Cc(T').$$

Implementation:
The implementation is depicted in Table 6.5.

T	$\bot$	*list*	*anylist*	*any*
T'	$\bot$	*list*	*any*	*any*

Table 6.5: Abstract Instantiation of Types.

6.3.6 Reverse Abstract Instantiation of Types (UNIST_TY)

Operation 14. UNIST_TY(T) = T'
This operation is the reverse of the IAT_TY operation. It approximates the set of terms that can be instantiated to a term $t \in Cc(T)$.

Specification:
Let $T, T' \in$ *Types*. The following relation holds:

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

Implementation:
The implementation is as follows:

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

6.3.7 Specialized Abstract Instantiation of Types (IAT_TY$_2$)

Operation 15. IAT_TY$_2(T_1, T_2) = T'$

This operation computes the type T' of a term whose type was T_1 and that has since been instantiated according to a substitution of a variable by a term whose type is T_2.

Specification:

Let T_1, T_2 be types, t_1, t_2 be terms, and y be a variable. The operation satisfies the following property:

$$\left.\begin{array}{l} t_1 \in Cc(T_1) \\ t_2 \in Cc(T_2) \end{array}\right\} \Rightarrow t_1\{y \leftarrow t_2\} \in Cc(T').$$

Implementation:

The implementation is given in the form of Table 6.6. The values of the first argument are given vertically.

	$\bot$	*list*	*anylist*	*any*
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
list	$\bot$	*list*	*list*	*list*
anylist	$\bot$	*anylist*	*anylist*	*any*
any	$\bot$	*any*	*any*	*any*

Table 6.6: Specialized Abstract Instantiation of Types: The values of the first argument are given vertically.

6.4 Operations on PSharing$_{I_p}$

See Section 4.2.7 (page 24) where the domain PSharing$_{I_p}$ is defined.

6.4.1 Construction of ps^* (PS_STAR)

Operation 16. PS_STAR$(frm, ps) = ps^*$

This operation constructs the complete possible sharing relation ps^*, given the *frame* component (frm) and the stored possible sharing relation ps (the latter considers only the pairs of terms that possibly share variables and whose patterns are undefined).

Implementation:

ps^* is defined as the smallest (symmetrical) relation on I_p satisfying the following two rules for all $i, j, k \in I_p$:

(1) $ps(i,j) \Rightarrow ps^*(i,j)$;
(2) $frm(k) = f(\ldots, j, \ldots)$ & $ps^*(i,j) \Rightarrow ps^*(k,i)$.

Chapter 7

Operations on Abstract Substitutions

See Section 4.2.9 (page 26) where the domain of Abstract Substitutions is defined.

Most of the following operations have been taken from *GAIA* [8]. However, some of the prototypes may differ slightly from the originate ones found in *GAIA*. Indeed, higher level operations acting on the domain of abstract sequences need more information about the structural mapping between input and output abstract substitutions. Further more, you will find in this paper some improvement about the "accuracy" of the results.

7.1 Least Upper Bound (LUB)

Operation 17. $\mathtt{LUB}(\beta_1, \beta_2) = \langle \beta, tr_1, tr_2 \rangle$

This operation[1] returns an abstract substitution (over I_p) $\beta = \beta_1 \sqcup \beta_2$ and two structural mappings tr_k between β and β_k, i.e., $tr_k : I_p \rightarrow I_{p_k}$ $(k = 1, 2)$.

Specification:

Let β_1 and β_2 be two abstract substitutions over I_{p_1} and I_{p_2} respectively, such that $dom(\beta_1) = dom(\beta_2) = D$ (i.e., $\beta_1, \beta_2 \in \mathtt{AS}_D$). $\mathtt{LUB}(\beta_1, \beta_2)$ produces an abstract substitution β such that:

$$\beta_1, \beta_2 \leq \beta \ \ \& \ \ \forall \beta' \in \mathtt{AS}_D : (\beta_1, \beta_2 \leq \beta') \Rightarrow (\beta \leq \beta').$$

Implementation:

We define the set E of pairs in correspondence induced by the same-value component:

$$E = \{(i, j) \mid \exists x \in D : i = sv_1(x) \ \& \ j = sv_2(x)\}.$$

[1]The prototype of this operation has changed relatively to the one proposed in *GAIA* ($\mathtt{LUB}(\beta_1, \beta_2) = \beta$). We now return explicitly the two structural mappings tr_1 and tr_2.

The remaining correspondences can be obtained from E and the frame components. Hence we define the set F of all correspondences as the smallest set satisfying the two following rules:

$$(1) \quad (i,j) \in E \Rightarrow (i,j) \in F$$

$$(2) \quad \left.\begin{array}{l} (i,j) \in F \\ frm_1(i) = f(i_1,\ldots,i_n) \\ frm_2(j) = f(j_1,\ldots,j_n) \end{array}\right\} \Rightarrow (i_k,j_k) \in F \ (1 \le k \le n).$$

We need a bijective function $t : F \rightarrow I_p$ to establish the relation between the old and the new indices of the corresponding subterms (where $p = \#F$). This bijective function is a function into I_m when restricted to E. $\texttt{LUB}(\beta_1,\beta_2)$ produces an abstract substitution $\beta = \langle sv, frm, \langle mo, ty, ps\rangle\rangle$ over I_p defined as follows:

$$\begin{array}{rcl} sv(x) & = & t(sv_1(x), sv_2(x)) \quad \forall x \in D; \\ frm & = & \{\langle t(i,j), f(t(i_1,j_1),\ldots,t(i_n,j_n))\rangle \mid \\ & & \qquad (i,j) \in F \ \& \\ & & \qquad frm_1(i) = f(i_1,\ldots,i_n) \ \& \\ & & \qquad frm_2(j) = f(j_1,\ldots,j_n)\}; \\ mo(t(i,j)) & = & \texttt{LUB}(mo_1(i), mo_2(j)) \quad \forall (i,j) \in F; \\ ty(t(i,j)) & = & \texttt{LUB}(ty_1(i), ty_2(j)) \quad \forall (i,j) \in F; \\ ps & = & \{(t(i,j), t(i',j')) \mid (i,j) \in F \ \& \ (i',j') \in F \ \& \\ & & \qquad (ps_1^*(i,i') \vee ps_2^*(j,j')) \ \& \\ & & \qquad frm(t(i,j)) = frm(t(i',j')) = undef\}. \end{array}$$

The operation returns also explicitly these two structural mappings:

$tr_1 : I_p \rightarrow I_{p_1}$ defined as $tr_1(k) = i$ if $\exists (i,j) \in F$ such that $t(i,j) = k$;
$tr_2 : I_p \rightarrow I_{p_2}$ defined as $tr_2(k) = j$ if $\exists (i,j) \in F$ such that $t(i,j) = k$.

7.2 Extended Least Upper Bound (EXT_LUB)

Operation 18. $\texttt{EXT\_LUB}(\beta_1,\beta_2) = \langle \beta, tr_1, tr_2, st\rangle$

Specification:
This operation[2] returns an abstract substitution $\beta = \beta_1 \sqcup \beta_2$, two structural mappings tr_k $(k = 1,2)$ between β and β_k (i.e., $tr_k : I_p \rightarrow I_{p_k}$) and a boolean value, st, such that $st = true$ implies that β is a strict union, i.e.,

$$\begin{array}{rcl} st = true & \Rightarrow & Cc(\beta) = Cc(\beta_1) \cup Cc(\beta_2). \\ st = false & \Rightarrow & Cc(\beta) \supseteq Cc(\beta_1) \cup Cc(\beta_2). \end{array}$$

[2]The EXT_LUB specification has been described in *Automated Verification* [14].

Implementation:
$\langle \beta, tr_1, tr_2 \rangle = \mathtt{LUB}(\beta_1, \beta_2)$, where LUB has been implemented just before.
If either $\beta_1 \leq \beta_2$ or $\beta_2 \leq \beta_1$ then $st = true$. Note that in this case, we have $\beta = max_{\leq}\{\beta_1, \beta_2\}$. The boolean st is set to $false$ otherwise.

7.3 Greatest Lower Bound (GLB)

Operation 19. $\mathtt{GLB}(\beta_1, \beta_2) = \langle \beta, tr_1, tr_2 \rangle$

Specification:
Let β_1 and $\beta_2 \in \mathtt{AS}_D$. This operation[3] returns the abstract substitution $\beta = \beta_1 \sqcap \beta_2$ (Greatest Lower Bound[4]) and two structural mappings tr_k between β_k and β, i.e., $tr_k : I_{p_k} \rightarrow I_p$ $(k = 1, 2)$.
In the sense of the concretization function, we have

$$Cc(\beta) = Cc(\beta_1) \cap Cc(\beta_2).$$

Implementation:
We define the set E of pairs in correspondence induced by the same-value component:

$$E = \{(i, j) \mid \exists x \in D : i = sv_1(x) \ \& \ j = sv_2(x)\}.$$

The remaining correspondences can be obtained from E and the frame components frm_1 and frm_2. Hence we define the set F of all correspondences as the set satisfying:

(1) $(i, j) \in E \Rightarrow (i, j) \in F$

(2) $\left.\begin{array}{l}(i, j) \in F \\ frm_1(i) = f(i_1, \ldots, i_n) \\ frm_2(j) = f(j_1, \ldots, j_n)\end{array}\right\} \Rightarrow (i_k, j_k) \in F \ (1 \leq k \leq n).$

(3) $\left.\begin{array}{l}(i, j) \in F \\ frm_1(i) = undef \\ frm_2(j) = f(j_1, \ldots, j_n)\end{array}\right\} \Rightarrow \begin{array}{l}(i_k, j_k) \in F \ (1 \leq k \leq n) \\ \text{where } i_k \text{ are new fresh variables.}\end{array}$

(4) $\left.\begin{array}{l}(i, j) \in F \\ frm_1(i) = f(i_1, \ldots, i_n) \\ frm_2(j) = undef\end{array}\right\} \Rightarrow \begin{array}{l}(i_k, j_k) \in F \ (1 \leq k \leq n) \\ \text{where } j_k \text{ are new fresh variables.}\end{array}$

Note that for the convenience of the algorithm, the components of the abstract substitutions must be updated to take into account the entering of new

[3]The implementation of GLB is not present in the references [7, 8, 13, 14]. We use the same approach as the one of the LUB operation.
[4]We recall the definition: $\beta_1 \sqcap \beta_2 = \beta \mid \beta \leq \beta_1, \beta_2 \ \& \ \forall \beta' \in \mathtt{AS}_D : (\beta' \leq \beta_1, \beta_2) \Rightarrow (\beta' \leq \beta)$.

fresh indices in the domain (properties (3) and (4)):

- For the property (3), we define every times:
$frm_1(i_k) \leftarrow undef \ \ (1 \leq k \leq n)$;
$mo_1(i_k) \leftarrow M_k \ \ (1 \leq k \leq n)$;
$ty_1(i_k) \leftarrow T_k \ \ (1 \leq k \leq n)$;
$frm_1(i) \leftarrow f(i_1, \ldots, i_n)$;
$ps_1 = [ps_1 \backslash \{(i,w) \in ps_1\}] \cup \{(i_k, w) | (1 \leq k \leq n) \ \wedge \ (i,w) \in ps_1\}$.
where $(M_1, \ldots, M_n) = \texttt{EXTR\_MO}(f/n, mo_1(i))$
and $(T_1, \ldots, T_n) = \texttt{EXTR\_TY}(f/n, ty_1(i))$

- For the property (4), we define every times:
$frm_2(j_k) \leftarrow undef \ \ (1 \leq k \leq n)$;
$mo_2(j_k) \leftarrow M_k \ \ (1 \leq k \leq n)$;
$ty_2(j_k) \leftarrow T_k \ \ (1 \leq k \leq n)$;
$frm_2(j) \leftarrow f(j_1, \ldots, j_n)$;
$ps_2 = [ps_2 \backslash \{(j,w) \in ps_2\}] \cup \{(j_k, w) | (1 \leq k \leq n) \ \wedge \ (j,w) \in ps_2\}$.
where $(M_1, \ldots, M_n) = \texttt{EXTR\_MO}(f/n, mo_2(j))$
and $(T_1, \ldots, T_n) = \texttt{EXTR\_TY}(f/n, ty_2(j))$

To check if the constraints (in terms of the both pattern components) defined by the two abstract substitutions are compatible (i.e., their intersection is not $\bot$), we define the boolean *incompatible* as follows:

$$(incompatible = true) \ \Rightarrow \ (Cc(\beta) = \bot).$$

This boolean is set to *true* if one of these two conditions occurs:

$$(A) \quad \left.\begin{array}{l} (i,j) \in F \\ frm_1(i) = f(i_1, \ldots, i_n) \\ frm_2(j) = g(j_1, \ldots, j_m) \\ f \neq g \end{array}\right\} \Rightarrow \ incompatible = true.$$

$$(B) \quad \left.\begin{array}{l} (i,j) \in F \\ frm_1(i) = f(i_1, \ldots, i_n) \\ frm_2(j) = f(j_1, \ldots, j_m) \\ n \neq m \end{array}\right\} \Rightarrow \ incompatible = true.$$

At this point of the implementation, if *incompatible* is *true*, then we return immediately $\beta = \bot$.
Otherwise, we need a function $t : F \rightarrow I_p$ to establish the relation between the old and the new indices of the corresponding subterms. This function is a function into I_m when restricted to E. $\texttt{GLB}(\beta_1, \beta_2)$ produces then an abstract

substitution $\beta = \langle sv, frm, \langle mo, ty, ps\rangle\rangle$ over I_p defined as follows:

$$\begin{array}{rcl}
sv(X) & = & t(sv_1(X), sv_2(X)) \quad \forall X \in D; \\
frm & = & \{\langle t(i,j), f(t(i_1,j_1),\ldots,t(i_n,j_n))\rangle \mid \\
 & & \qquad (i,j) \in F \ \& \\
 & & \qquad frm_1(i) = f(i_1,\ldots,i_n) \ \& \\
 & & \qquad frm_2(j) = f(j_1,\ldots,j_n)\}; \\
mo(t(i,j)) & = & \texttt{GLB}(mo_1(i), mo_2(j)) \quad \forall (i,j) \in F; \\
ty(t(i,j)) & = & \texttt{GLB}(ty_1(i), ty_2(j)) \quad \forall (i,j) \in F; \\
ps & = & \{(t(i,j), t(i',j')) \mid (i,j) \in F \ \& \ (i',j') \in F \ \& \\
 & & \qquad (ps_1^*(i,i') \wedge ps_2^*(j,j')) \ \& \\
 & & \qquad frm(t(i,j)) = frm(t(i',j')) = undef\}.
\end{array}$$

Note that if there exists a new indice $k \in I_p$ such that $mo(k) = \bot$ or $ty(k) = \bot$, then the resulting abstract substitution β is $\bot$.

We must also check the consistency between the two sharing-components. If the two following properties are not satisfied, the GLB operation returns $\bot$:

$$\begin{array}{lll}
(1) & \forall i \in I_{p_1} : \forall j_k, j_l \in I_{p_2} & \\
 & \left.\begin{array}{l} (i,j_k) \in F \\ (i,j_l) \in F \\ mo_2(j_k) \neq ground \\ mo_2(j_l) \neq ground \end{array}\right\} & \Rightarrow ps_2^*(j_k, j_l) \\
(2) & \forall j \in I_{p_2} : \forall i_k, i_l \in I_{p_1} & \\
 & \left.\begin{array}{l} (i_k,j) \in F \\ (i_l,j) \in F \\ mo_1(i_k) \neq ground \\ mo_1(i_l) \neq ground \end{array}\right\} & \Rightarrow ps_1^*(i_k, i_l)
\end{array}$$

Finally, we return these two structural mappings:

$tr_1 : I_{p_1} \rightarrow I_p$ defined as $tr_1(i) = t(i,j) \mid \exists j$ such that $(i,j) \in \texttt{F}$
$tr_2 : I_{p_2} \rightarrow I_p$ defined as $tr_2(j) = t(i,j) \mid \exists i$ such that $(i,j) \in \texttt{F}$

7.4 Extension at Clause Entry (EXTC_{subst})

Operation 20. $\text{EXTC}_{subst}(c, \beta) = \beta'$

This operation extends an abstract substitution with the new free variables of the clause. It is used at the entry of a clause to include the variables in the body not present in the head.

Specification:

Let $D = \{x_1, \ldots, x_n\}$ be the set of variables in the head of c, $dom(\beta) = D$ and $D' = \{x_1, \ldots, x_{n+k}\}$ be the set of all variables in c. $\text{EXTC}_{subst}(c, \beta)$ produces

a substitution β' such that:

$$\begin{aligned} Cc(\beta') \;=\; & \{\, \theta : dom(\theta) = D' \;\&\\ & \theta_{/D} \in Cc(\beta) \;\&\\ & x_{n+1}\theta, \ldots, x_{n+k}\theta \text{ are distinct renaming variables } \&\\ & x_{n+j}\theta \notin codom(\theta_{/D})\ (1 \le j \le k)\ \}. \end{aligned}$$

Implementation:
Under the assumptions of the specification, the resulting abstract substitution $\beta' = \langle sv', frm', \langle mo', ty', ps' \rangle\rangle$ is defined as follows:

$$\begin{aligned} p' &= p + k.\\ m' &= m + k.\\ sv'(x_i) &= sv(x_i) \qquad (1 \le i \le n).\\ sv'(x_{n+i}) &= m + i \qquad (1 \le i \le k).\\ t(i) &= \begin{cases} i & (1 \le i \le m);\\ i - k & (m' < i \le p'). \end{cases}\\ frm' &= \{\langle i, f(i_1, \ldots, i_n)\rangle : \langle t(i), f(t(i_1), \ldots, t(i_n))\rangle \in frm\}.\\ mo'(i) &= \begin{cases} mo(t(i)) & (1 \le i \le m) \text{ or } (m' < i \le p');\\ var & (m < i \le m'). \end{cases}\\ ty'(i) &= \begin{cases} ty(t(i)) & (1 \le i \le m) \text{ or } (m' < i \le p');\\ anylist & (m < i \le m'). \end{cases}\\ ps' &= \{(i,j) \mid ps(t(i), t(j))\} \cup \{(i,i) \mid m < i \le m'\}. \end{aligned}$$

The new variables must be mapped onto indices $m+1, \ldots, m+k$. It is the reason the function $t : I_{p'} \to I_p$ is introduced to shift old indices greater than m (See Figure 7.1).

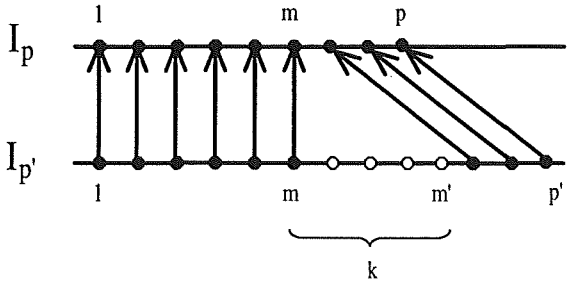


Figure 7.1: EXTC$_{subst}$, shift function $t : I_{p'} \to I_p$

7.5 Restriction at Clause Exit (RESTRC$_{subst}$)

Operation 21. RESTRC$_{subst}(c, \beta) = \langle \beta', tr \rangle$

This operation[5] restricts an (output) abstract substitution of a clause to the head variables only. It is used at the exit of a clause.

Specification:

Let D be the domain of β, and let D' be the set of variables in the head of c ($D' \subseteq D$). RESTRC$_{subst}(c, \beta)$ produces an abstract substitution β' such that:

$$Cc(\beta') = \{\theta_{/D}; \theta \in Cc(\beta)\}.$$

Moreover, this operation returns a structural mapping tr between β' and β ($tr : I_{p'} \rightarrow I_p$).

Implementation:

Under the assumptions of the specification, the resulting abstract substitution $\beta' = \langle sv', frm', \langle mo', ty', ps' \rangle \rangle$ and the structural mapping tr are defined as follows:

$$\begin{array}{rcl}
I & = & \text{REACHABLE}(sv(D'), frm); \\
m' & = & \#sv(D'); \\
p' & = & \#I; \\
t & = & \text{the unique strictly increasing function from } I \text{ to } I_{p'} \text{ (See Figure 7.2).} \\
sv'(x) & = & t(sv(x)) \quad \forall x \in D'; \\
frm' & = & \{\langle t(i), f(t(i_1), \ldots, t(i_n)) \rangle : \langle i, f(i_1, \ldots, i_n) \rangle \in frm \ \& \ i \in I\}; \\
mo'(t(i)) & = & mo(i) \quad \forall i \in I; \\
ty'(t(i)) & = & ty(i) \quad \forall i \in I; \\
ps' & = & \{(t(i), t(j)) : i, j \in I \ \& \ ps(i, j)\}; \\
tr & = & t^{-1}.
\end{array}$$

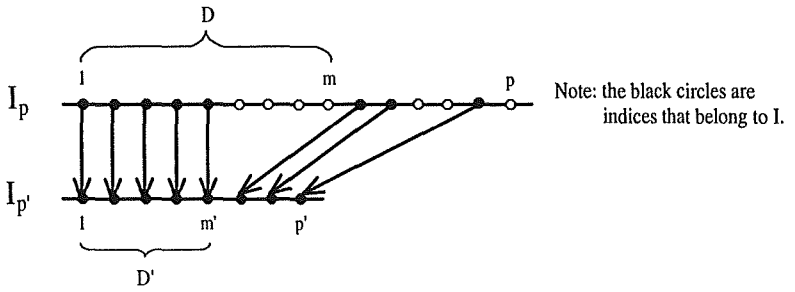


Figure 7.2: RESTRC$_{subst}$, shift function $t : I_p \rightarrow I_{p'}$

[5]The prototype of this operation has changed relatively to the one proposed in *GAIA* (RESTRC$_{subst}(c, \beta) = \beta'$). We now return explicitly the structural mapping tr.

7.6 Restriction before a Goal (RESTRG_{subst})

Operation 22. $\text{RESTRG}_{subst}(l, \beta) = \beta'$

This operation restricts an abstract substitution to (a renaming of) the variables occurring in the literal l. It is used before the (abstract) execution of a literal in the body of a clause.

Specification:

Let β be an abstract substitution on $D = \{x_1, \ldots, x_n\}$ and l be a literal $p(x_{i_1}, \ldots, x_{i_m})$ (or $x_{i_1} = x_{i_2}$ or $x_{i_1} = f(x_{i_2}, \ldots, x_{i_m})$). $\text{RESTRG}_{subst}(l, \beta)$ returns the abstract substitution obtained by these two steps:

(1) projecting β on $\{x_{i_1}, \ldots, x_{i_m}\}$, obtaining β_{aux}
(2) expressing β_{aux} in terms of $\{x_1, \ldots, x_m\}$ by mapping x_{i_k} *to* x_k, giving β'.

The resulting abstract substitution is expressed in terms of $\{x_1, \ldots, x_m\}$, that is, as an input abstract substitution for p/m.

Implementation:

The algorithm is exactly the same you can find in $\text{RESTRC}_{subst}(c, \beta)$, if you consider in this case that D' is the set of variables occurring in l (that is $D' = \{x_{i_1}, \ldots, x_{i_m}\}$), giving β_{aux}. The result β' is β_{aux} where you replace each x_{i_k} by x_k, such that $sv' : \{x_1, \ldots, x_m\} \rightarrow I_{p'}$.

7.7 General Unification between Two Terms

The purpose of this Section is to show the implementation of the unification at the abstract substitution level[6]. In the following, by abuse of language, we often use "abstract term i" to denote the information associated with an index i in a substitution β. We also use "abstract unification of abstract terms i and j" to denote the result of an operation whose result is an abstract substitution approximating the set of concrete substitutions resulting from the unification of the terms t_i and t_j in all the substitutions belonging to $Cc(\beta)$.

7.7.1 Overview

The kernel of the unification operation is a procedure to unify two abstract terms. Our abstract unification considers three cases depending upon the pattern components of i and j:

$$(1) frm(i) = frm(j) = undef \ ;$$
$$(2) frm(i) = undef \ \& \ frm(j) = f(j_1, \ldots, j_n) \ ;$$
$$(3) frm(i) = f(i_1, \ldots, i_n) \ \& \ frm(j) = f(j_1, \ldots, j_n).$$

[6] We use the approach proposed in [7] for the global schema of the implementation.

The rest of this section is organized in the following way: we start by introducing some notations which significantly simplify the definitions. We then present the suboperations FCTA (whose job is to manipulate indices), UNIF1 (that performs the first case (basic case)), SPECAT (that achieves a big part of the second case) and finally the operation UNIF (that resolves the entire problem).

7.7.2 Notation

The removal operations are rather frequent in the unification process and, instead of updating permanently the components, the equalities will be stored using a function $fi : I_q \rightarrow I_p$ such that $fi(i) = fi(j) \Rightarrow t_i = t_j$, where I_q denotes a set of old indices and I_p denotes a set of new indices. This allows us to simplify the presentation and the implementation as well. The idea is that the same value component needs only to be updated at the very end of each operation. So for the moment we restrict attention to two components omitting the same value component.

We call a δ-tuple the association (frm, α, fi) of two components frm and $\alpha = \langle mo, ty, ps \rangle$ both defined on the same set of indices I_p, and a function fi. The unification suboperations are defined on δ-tuples. Note also that we implicitly assume that a δ-tuple δ_k is associated to a tuple $(frm_k,\ \alpha_k,\ fi_k) = (frm_k,\ \langle mo_k, ty_k, ps_k \rangle,\ fi_k)$ (and similarly a δ-tuple δ' to a tuple $(frm',\ \alpha',\ fi') = (frm',\ \langle mo', ty', ps' \rangle,\ fi')$). As usual, we define the meaning of δ-tuples by means of a concretization function as follows:

$$\begin{aligned} Cc(\delta) &= \{(u_1, \ldots, u_q) : \exists (t_1, \ldots, t_p) \in Cc(frm) \cap Cc(\alpha) : \\ &\qquad\qquad u_i = t_{fi(i)} \ \ (1 \leq i \leq q)\} \\ &= \{(t_{fi(1)}, \ldots, t_{fi(q)}) : (t_1, \ldots, t_p) \in Cc(frm) \cap Cc(\alpha)\}. \end{aligned}$$

In the rest of the presentation, we will often have to write expressions such as $expr(t(i_1), \ldots, t(i_n))$ where $i_1, \ldots, i_n \in I_{p_2}$ and $t : I_{p_2} \rightarrow I_{p_1}$. We take the convention of representing those expressions as

$$expr(i_1, \ldots, i_n)\ \ (t),$$

meaning that all indices i_k from I_{p_2} in the expression have to be substituted by their values $t(i_k)$.

7.7.3 The sub-operation FCTA

Operation 23. $\texttt{FCTA}(i, j, \delta) = \delta'$

This operation amounts to adding an equality between terms t_i and t_j and to propagating this equality in the rest of tuple δ. The basic idea behind this operation is to remove a subterm which is no longer necessary.

Specification:

Let $\bar{u} = (u_1, \ldots, u_q)$ be a δ-tuple of terms. Then the following holds:

$$\left.\begin{array}{l} \bar{u} \in Cc(\delta) \\ u_i = u_j \end{array}\right\} \Rightarrow \bar{u} \in Cc(\delta')$$

Implementation:
Let us define two functions, assuming $max = max(i,j)\ (fi)$
1. $ti : I_p \rightarrow I_{p-1}$

$$ti(k) = \begin{cases} k & \text{if } k < max; \\ min(i,j)(fi) & \text{if } k = max; \\ k-1 & \text{if } k > max; \end{cases}$$

2. $it : I_{p-1} \rightarrow I_p$

$$it(k) = \begin{cases} k & \text{if } k < max; \\ k+1 & \text{if } k \geq max; \end{cases}$$

When applied to the components, the function ti removes one of the terms (the one with the largest index) by pushing leftwards the indices which are greater than the removed term while the function it allows us to retrieve previous information.
Finally, the function $\texttt{FCTA}(i,j,\delta) = \delta'$ is defined as

$$\begin{array}{lcl} frm' & = & \{\langle ti(k), f(ti(k_1),\ldots,ti(k_n))\rangle : \langle k, f(k_1,\ldots,k_n)\rangle \in frm\}; \\ mo' & = & mo \circ it; \\ ty' & = & ty \circ it; \\ ps' & = & \{(ti(k), ti(l)) : ps(k,l)\}; \\ fi' & = & ti \circ fi. \end{array}$$

Note that the function fi of the δ-tuple needs to be updated as well.

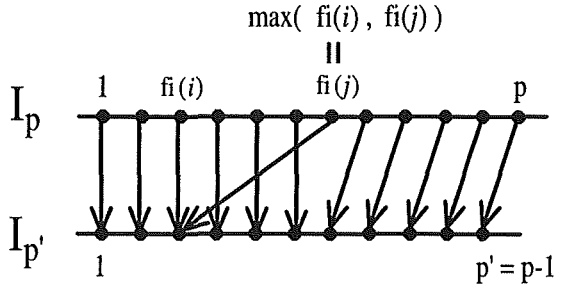


Figure 7.3: FCTA, shift function $ti : I_p \rightarrow I_{p-1}$

7.7.4 The sub-operation UNIF1

Operation 24. $\texttt{UNIF1}(i,j,\delta) = \langle \delta', ss\rangle$
This operation[7] is defined on a δ-tuple $\delta = (frm, \alpha, fi)$. It assumes that

[7]The prototype of this operation has changed relatively to the one proposed in *GAIA* ($\texttt{UNIF1}(i,j,\delta) = \delta'$). We now return explicitly an information about the sure success (ss) of the unification.

$frm(i) = frm(j) = undef$ (fi) and produces another δ-tuple $\delta' = (frm',\ \alpha',\ fi')$. Informally, $\texttt{UNIF1}(i, j, \delta)$ unifies subterms i and j in the δ-tuple δ, giving δ'. An information about the sure success (ss) of the unification is also returned.

Specification:

Given a p-tuple of terms $\bar{t} = (t_1, \ldots, t_p)$ and a substitution σ, the operation verifies

$$\left.\begin{array}{l} \sigma \in mgu(t_i, t_j) \\ \bar{t} \in Cc(\delta) \end{array}\right\} \Rightarrow \bar{t}\sigma \in Cc(\delta').$$

and

$$ss = true \Rightarrow \text{the (abstract) terms } fi(i) \text{ and } fi(j) \text{ (i.e., the terms } t_{fi(i)} \text{ and } t_{fi(j)}) \text{ are unifiable in } \delta.$$

Implementation:

Let $\delta = \langle frm, \alpha, fi \rangle$. The result of $\texttt{UNIF1}(i, j, \delta)$ is $\delta' = \texttt{FCTA}(i, j, \delta_1)$, where $\delta_1 = (frm_1,\ \alpha_1,\ fi_1)$ is computed as follows:

$$\begin{array}{rcl} frm_1 & = & frm; \\ \alpha_1 & = & \texttt{ALPHA\_UNIF1}(frm, \alpha, i, j)\ \ (fi); \\ fi_1 & = & fi. \end{array}$$

where $\texttt{ALPHA\_UNIF1}(frm, \alpha, i, j) = \alpha' = \langle mo', ty', ps' \rangle$ is defined as follows[8]:

Construction of mo':

Let $SH(k)$ be the condition $ps^*(i,k) \vee ps^*(j,k)$, defined for $(1 \leq k \leq p)$.

$$mo'(k) = \left\{ \begin{array}{l} mo(k) \text{ if } \left\{ \begin{array}{l} i \neq k \neq j \\ \neg SH(k) \end{array}\right. \\ \texttt{UAT\_MO}(mo(i), mo(j)) \text{ if } (k = i \vee k = j) \\ \texttt{MATCH\_MO}(mo(k), f, mo'(k_1), \ldots, mo'(k_n)) \text{ if } \left\{ \begin{array}{l} i \neq k \neq j \\ SH(k) \\ frm(k) = \\ \quad f(k_1, \ldots, k_n) \end{array}\right. \\ \texttt{IAT\_MO}_2(mo(k), mo'(i)) \ \text{ if } \left\{ \begin{array}{l} i \neq k \neq j \\ SH(k) \\ frm(k) = undef \\ (mo(i) = var) \vee (mo(j) = var) \end{array}\right. \\ \texttt{IAT\_MO}(mo(k)) \text{ if } \left\{ \begin{array}{l} i \neq k \neq j \\ SH(k) \\ frm(k) = undef \\ (mo(i) \neq var) \wedge (mo(j) \neq var) \end{array}\right. \end{array}\right.$$

[8]We made this algorithm a little more accurate than the one proposed in *GAIA*, by splitting the case where the frame is *undef* according to either i or j is a variable or not. The gain of accuracy comes from the fact that the operation $\texttt{IAT\_MO}_2$ is more precise than $\texttt{IAT\_MO}$.

Construction of ty': [9]

$$ty' = \begin{cases} list & \text{if } \begin{cases} mo'(k) = ground \\ ty^*(k) = anylist \end{cases} \\ anylist & \text{if } \begin{cases} mo'(k) = var \\ ty^*(k) = any \end{cases} \\ ty^* & \text{otherwise.} \end{cases}$$

where

$$ty^*(k) = \begin{cases} ty(k) \text{ if } \begin{cases} i \neq k \neq j \\ \neg SH(k) \\ frm(k) = undef \end{cases} \\ \texttt{UAT\_TY}(ty(i), ty(j)) \text{ if } (k = i \vee k = j) \\ \texttt{MATCH\_TY}(ty(k), f, ty'(k_1), \ldots, ty'(k_n)) \text{ if } \begin{cases} i \neq k \neq j \\ frm(k) = \\ \quad f(k_1, \ldots, k_n) \end{cases} \\ \texttt{IAT\_TY}_2(ty(k), ty'(i)) \text{ if } \begin{cases} i \neq k \neq j \\ SH(k) \\ frm(k) = undef \\ (mo(i) = var) \vee (mo(j) = var) \end{cases} \\ \texttt{IAT\_TY}(ty(k)) \text{ if } \begin{cases} i \neq k \neq j \\ SH(k) \\ frm(k) = undef \\ (mo(i) \neq var) \wedge (mo(j) \neq var) \end{cases} \end{cases}$$

Construction of ps':

$$ps' = \begin{cases} ps_a & \text{if } ground(mo'(i)) \\ ps_a \cup ps_b & \text{otherwise.} \end{cases}$$

where

$$\begin{aligned} ps_a &= \{(k,l) \mid ps(k,l) \;\&\; \neg ground(mo'(k)) \;\&\; \neg ground(mo'(l))\}, \\ ps_b &= \{(k,l) \mid \exists k', l' : ps(k',k) \;\&\; ps(l',l) \;\&\; k', l' \in \{i,j\}\}. \end{aligned}$$

The pattern component frm remains the same, as no new pattern is introduced.

In the mode component, a subterm having no sharing with subterms i or j is not affected by the unification and keeps its mode. The mode of i and j is given by the abstract unification of their old mode. If a subterm has sharing with i and j and has a well-defined pattern, its new mode is given by the matching operation on modes, given the old mode of the subterm and the new modes of the arguments of its pattern. The matching is necessary because subterms i and j may be reachable from k, and since their modes may have been updated,

[9] We note that a *ground anylist* is in fact a *ground list* and that a *var any* is in fact a *var anylist*.

the mode of k may need to be updated as well. If a subterm has an undefined pattern, its new mode is given by the abstract instantiation of its old mode.

The constructing of the type component is analogous to the mode component.

In the sharing component, two cases are distinguished depending on the resulting modes of the unified subterms. If their new mode is *ground*, only the old sharing is considered and updated to take into account the new groundness information. Otherwise, the sharing component should merge the sharing information on i and j.

Finally, the boolean ss (sure success) is set to *true* if one of the three following conditions is satisfied[10]:

$$(1) \quad (\, i = j \,) \quad (fi).$$

$$(2) \quad (\, (mo(i) = var) \ \wedge \ (mo(j) = var) \,) \quad (fi).$$

$$(3) \quad \left(\begin{array}{c} (\, (mo(i) = var) \ \vee \ (mo(j) = var) \,) \\ \neg ps(i,j) \\ (ty(i) = list) \Rightarrow (ty(j) \leq anylist) \\ (ty(j) = list) \Rightarrow (ty(i) \leq anylist) \end{array} \right) (fi).$$

7.7.5 The sub-operation SPECAT

Operation 25. $\texttt{SPECAT}(i, j, \delta) = \delta'$

This operation is defined on a δ-tuple $\delta = \langle frm, \alpha, fi \rangle$. $\texttt{SPECAT}(i, j, \delta)$ is useful for the unification of two terms t_i, t_j where $frm(i) = undef$ and $frm(j) = f(j_1, \ldots, j_n)$. Such an unification can be achieved in two steps:

1. the unification of t_i and $f(y_1, \ldots, y_n)$ giving σ where $y_1, \ldots, y_n$ are new variables;
2. the unification of $(y_1, \ldots, y_n)\sigma$ and $t_{j_1}, \ldots, t_{j_n}$.

The operation SPECAT performs the first step. The second step is carried out by the general unification procedure (UNIF).

Specification:

Given $\bar{t} = (t_1, \ldots, t_p)$, a p-tuple of terms, σ a substitution, $y_1, \ldots, y_n$, n distinct variables not occurring in $\bar{t}$, the operation $\texttt{SPECAT}(i, j, \delta) = \delta'$ verifies

$$\left. \begin{array}{l} \bar{t} \in Cc(\delta) \\ t_i, t_j \text{are unifiable} \\ \sigma \in mgu(f(y_1, \ldots, y_n), t_i) \end{array} \right\} \ \Rightarrow \ (t_1, \ldots, t_p, y_1, \ldots, y_n)\sigma \in Cc(\delta').$$

[10]The first two conditions come from [9]. We add the third one, in order to respect the type constraint.

Implementation:

The implementation returns $\delta' = \bot$ if $fi(i) \in \texttt{REACHABLE}(fi(j), frm)$: this case corresponds to the "occur check" problem. Otherwise let $p' = p + n$. The $\Re$-component α' is obtained by adding to α the correspondence between the term t_i and the set of terms $\{t_{p+1}, \ldots, t_{p+n}\}$, the pattern component is defined by adding the new pattern, and the new function $fi' : I_{q+n} \rightarrow I_{p+n}$ is obtained by including the new indices (where $dom(fi) = I_q$). Note that $1 \leq i, j \leq q$.

More precisely,

$$\begin{array}{lcl} frm' & = & frm \cup \{\langle fi(i), f(p+1, \ldots, p+n)\rangle\}; \\ \alpha' & = & \texttt{ALPHA\_SPECAT}(frm, \alpha, i, j)\ (fi); \\ fi'(k) & = & \begin{cases} fi(k) & \text{if } k \leq q; \\ k - q + p & \text{if } k > q. \end{cases} \end{array}$$

where $\texttt{ALPHA\_SPECAT}(frm, \alpha, i, j) = \alpha' = \langle mo', ty', ps'\rangle$ is defined as follows[11]:

The mode component mo', considering $\langle M_1, \ldots, M_n\rangle = \texttt{EXTR\_MO}(f, mo(i))$:

(1) $k \leq p$

if $i \neq k$ **and** ($\neg ps^*(i,k) \vee k \in \texttt{REACHABLE}(j, frm)$) **then**
$mo'(k) = mo(k)$
else if $k = i$ **then**
$mo'(k) = ground$ **if** $\texttt{CONS\_MO}(f, var, \ldots, var) = ground$
$mo'(k) = \texttt{CONS\_MO}(f, var, \ldots, var)$ **if** $mo(i) = var$
$mo'(k) = \texttt{GLB}(novar, \texttt{LUB}(mo(i), \texttt{CONS\_MO}(f, var, \ldots, var)))$ **if** $mo(i) > var$
$mo'(k) = mo(i)$ **otherwise**
else if $k \neq i$ **and** $frm(k) = g(k_1, \ldots, k_m)$ **then**
$mo'(k) = \texttt{MATCH\_MO}(mo(k), g, mo'(k_1), \ldots, mo'(k_m)))$
else if $k \neq i$ **and** $frm(k) = undef$ **then**
$mo'(k) = \texttt{IAT\_MO}_2(mo(k), mo'(i))$ **if** $mo(i) \geq var$
$mo'(k) = mo(k)$ **otherwise**

(2) $p < k \leq p + n$

if $mo(i) = var$ **then** $mo'(k) = var$
if $mo(i) > var$ **then** $mo'(k) = \texttt{LUB}(M_{k-p}, var)$
else $mo'(k) = M_{k-p}$

The type component ty' [12]:

$$ty' = \begin{cases} list & \text{if } \begin{cases} mo'(k) = ground \\ ty^*(k) = anylist \end{cases} \\ anylist & \text{if } \begin{cases} mo'(k) = var \\ ty^*(k) = any \end{cases} \\ ty^* & \text{otherwise.} \end{cases}$$

[11] We made this algorithm a little more accurate than the one proposed in *GAIA*. In particular, remark that the new mode of i is at least *novar*.

[12] We note that a *ground anylist* is in fact a *ground list* and that a *var any* is in fact a *var anylist*.

where ty^* is computed as below, considering $\langle T_1, \ldots, T_n \rangle = \texttt{EXTR\_TY}(f, ty(i))$:

(1) $k \leq p$
  if $i \neq k$ **and** ($\neg ps^*(i,k) \vee k \in \texttt{REACHABLE}(j, frm)$) **then**
    $ty^*(k) = ty(k)$
  else if $k = i$ **then**
    $ty^*(k) = \texttt{UAT\_TY}(ty(i), \texttt{CONS\_TY}(f, anylist, \ldots, anylist))$
  else if $k \neq i$ **and** $frm(k) = g(k_1, \ldots, k_m)$ **then**
    $ty^*(k) = \texttt{MATCH\_TY}(ty(k), g, ty'(k_1), \ldots, ty'(k_m)))$
  else if $k \neq i$ **and** $frm(k) = undef$ **then**
    $ty^*(k) = \texttt{IAT\_TY}_2(ty(k), ty'(i))$ **if** $mo(i) \geq var$
    $ty^*(k) = ty(k)$ **otherwise**
(2) $p < k \leq p + n$
  $ty^*(k) = T_{k-p}$

The sharing component ps':

$$ps' = (ps \backslash \tilde{ps}_1) \cup \tilde{ps}_2 \cup ps_3 \cup ps_4,$$

where

$$\begin{array}{lcl}
ps_1 & = & \{\ (i,k) \mid 1 \leq k \leq p \textbf{ and } ps(i,k)\ \}, \\
ps_2 & = & \{\ (k, p+l) \mid 1 \leq k \leq p \textbf{ and } ps(i,k) \textbf{ and } k \neq i \textbf{ and } 1 \leq l \leq n \textbf{ and} \\
 & & mo'(k) \neq ground \neq mo'(p+l)\ \}, \\
ps_3 & = & \{\ (k,k) \mid p < k \leq p+n\ \} \textbf{ if } mo(i) \geq var \\
 & & \emptyset \textbf{ otherwise} \\
ps_4 & = & \emptyset \textbf{ if } mo(i) = var \\
 & & \{\ (k,l) \mid p < k, l \leq p+n \textbf{ and } M_{k-p} \neq ground \neq M_{l-p}\ \} \textbf{ otherwise}
\end{array}$$

Note that $\tilde{ps}$ is the symmetrical relation deduced from ps.

Justification:

The implementation is based on a reasoning about the unification process of $f(y_1, \ldots, y_n)$ and t_i, under the assumption that t_i and t_j are unifiable.

By hypothesis, t_j is of the form $f(t_{j_1}, \ldots, t_{j_n})$. Therefore, t_i is either a variable y or a compound term $f(u_1, \ldots, u_n)$. Hence, two cases must be distinguished in the reasoning. They are, however, amalgamated in the implementation mainly by means of operation LUB.

Now consider the case where t_i is a variable (say, y). This case is possible only if $var \leq mo(i)$. Then σ is simply $\{y \leftarrow f(y_1, \ldots, y_n)\}$. The mode of t_i becomes $\texttt{CONS\_MO}(f, var, \ldots, var)$. Variables $y_1, \ldots, y_n$ are not instantiated: their modes remain var.

On the contrary, if t_i is the compound term $f(u_1, \ldots, u_n)$, σ is equal to $\{y_1 \leftarrow u_1, \ldots, y_n \leftarrow u_n\}$, since the y_i are new distinct variables.
EXTR_MO(f, $mo(i)$) provides the modes $M_1, \ldots, M_n$ of $u_1, \ldots, u_n$, which are the new modes for $y_1, \ldots, y_n$. The mode of t_i is not modified.

Now what is the effect of applying σ to any other term t_k? Clearly, $t_k\sigma$ differs from t_k only if t_i is a variable y and if t_k contains y. (In the other cases, only $y_1, \ldots, y_n$ are modified). This cannot happen if t_k does not share with t_i or if t_k is a subterm of t_j, because t_j should contain y and the unification would fail. In any other cases, if the pattern of t_k is known, we adjust its mode according to the new mode of its subterms; if it is not, t_k becomes $t_k\{y \leftarrow f(y_1, \ldots, y_n)\}$. So its new mode is IAT_MO$_2$($mo(k)$, CONS_MO($f, var, \ldots, var$)).

7.7.6 The operation UNIF

Operation 26. UNIF$(i, j, \delta) = \langle \delta', ss \rangle$
Let us present the main procedure for unification UNIF$(i, j, \delta) = \delta'$ which consists mainly of the three cases mentioned in the overview[13]. Informally speaking, procedure UNIF(i, j, δ) unifies subterms i and j in the δ-tuple δ. In the following, we say that $(u_1, \ldots, u_m)$ is a prefix of $(t_1, \ldots, t_n)$ $(m \leq n)$ iff $u_i = t_i$ $(1 \leq i \leq m)$.

Specification:
Given $\bar{u} = (u_1, \ldots, u_q)$, a q-tuple of terms, and σ a substitution, the operation verifies

$$\left.\begin{array}{l} \sigma \in mgu(u_i, u_j) \\ \bar{u} \in Cc(\delta) \end{array}\right\} \Rightarrow \exists \bar{u}' \in Cc(\delta') : \bar{u}\sigma \text{ is a prefix of } \bar{u}'.$$

and

$$ss = true \Rightarrow \text{the (abstract) terms } fi(i) \text{ and } fi(j) \text{ (i.e., the terms } t_{fi(i)} \text{ and } t_{fi(j)}) \text{ are unifiable in } \delta.$$

Implementation:
The skeleton of the algorithm is the same as the one proposed in *GAIA*, but

[13]The prototype of this operation has changed relatively to the one proposed in *GAIA* (UNIF$(i, j, \delta) = \delta'$). We now return explicitly an information about the sure success (ss) of the unification.

differs by the insertion in each case of a computation of the boolean ss.

if $i = j \ \ (fi)$ **then**
$\quad \delta' = \delta$
$\quad ss = true$
else if $(frm(i) = undef = frm(j) \ \ (fi))$ **then**
$\quad \langle \delta', ss \rangle = \texttt{UNIF1}(i, j, \delta)$
else if $((frm(i) = undef \ \ (fi)) \ \&\ \texttt{SPECAT}(i, j, \delta) = \bot) \vee$
$\quad ((frm(j) = undef \ \ (fi)) \ \&\ \texttt{SPECAT}(j, i, \delta) = \bot) \vee$
$\quad ((frm(i) = f(i_1, \ldots, i_n) \ \&\ frm(j) = g(j_1, \ldots, j_n) \ \ (fi)) \ \&$
$\quad (f \neq g \vee n \neq m))$ **then**
$\quad\quad \delta' = \bot$
$\quad\quad ss = false$
else
$\quad \delta' = \texttt{FCTA}(i, j, \delta_n)$ **and** $ss = \bigwedge_{k=0}^{n} ss_k$ where

$$\delta_0 = \begin{cases} \texttt{SPECAT}(i, j, \delta) & \textbf{if } (frm(i) = undef \ \ (fi)) \\ \texttt{SPECAT}(j, i, \delta) & \textbf{if } (frm(j) = undef \ \ (fi)) \\ \delta & \textbf{otherwise} \end{cases}$$

$$ss_0 = \begin{cases} true & \textbf{if } (\ frm(i) \neq undef \ \ \&\ \ frm(j) \neq undef\ ) \ \ (fi) \\ true & \textbf{if } \begin{pmatrix} frm(i) = undef \\ mo(i) = var \\ \neg ps^*(i, j) \\ (ty(i) = list) \Rightarrow (ty(j) \leq anylist) \end{pmatrix} \ \ (fi) \\ true & \textbf{if } \begin{pmatrix} frm(j) = undef \\ mo(j) = var \\ \neg ps^*(i, j) \\ (ty(j) = list) \Rightarrow (ty(i) \leq anylist) \end{pmatrix} \ \ (fi) \\ false & \textbf{otherwise} \end{cases}$$

$(frm_0(i) = f(i_1, \ldots, i_n) \ \&\ frm_0(j) = f(j_1, \ldots, j_n) \ \ (fi_0))$
$\langle \delta_k, ss_k \rangle = \texttt{UNIF}(i_k, j_k, \delta_{k-1}) \ (1 \leq k \leq n).$

Justification:

The implementation mimics a recursive algorithm for concrete unification as long as at least one of the patterns of u_i and u_j is known.

The base case, when none of the patterns are known, is handled by `UNIF1`. In this case the specification of `UNIF` reduces to the specification of `UNIF1`.

The simplest recursive case takes place when i and j have compatible patterns, that is, when $frm(i) = f(i_1, \ldots, i_n)$ and $frm(j) = f(j_1, \ldots, j_n)$. Then

$u_i = f(u_{i_1}, \ldots, u_{i_n})$ and $u_j = f(u_{j_1}, \ldots, u_{j_n})$. It is a well-known result of logic programming that $\sigma = \sigma_n$, where $\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_n$ are defined as follows:

$$\begin{array}{lcl} \sigma_0 & = & \{\}; \\ \sigma'_k & \in & mgu(u_{i_k}\sigma_{k-1}, u_{j_k}\sigma_{k-1}); \\ \sigma_k & = & \sigma_{k-1}\sigma'_k. \end{array}$$

$\bar{u}_\sigma$ can be computed by first unifying u_{i_1} and u_{j_1} and then applying σ_1 to $\bar{u}$. Terms $u_{i_2}\sigma_1$ and $u_{j_2}\sigma_1$ of the new q-tuple can then be unified, giving σ_2, and so on. The recursive calls $\texttt{UNIF}(i_k, j_k, \sigma_{k-1})$ mimic this sequence of operations on the abstract domain.

The nonsymmetric case, that is, when only one pattern (say, $frm(j)$) is known, is the most complex. It is reduced to the symmetric case by means of operation SPECAT. Suppose that u_i is unifiable to u_j, which is of the form $f(u_{j_1}, \ldots, u_{j_n})$. Then, there exist two substitutions σ_0 and σ' such that

$$\begin{array}{l} \sigma_0 \in mgu(f(y_1, \ldots, y_n), u_i), \\ \sigma' \in mgu(f(y_1, \ldots, y_n)\sigma_0, u_j), \\ \bar{u}\sigma \text{ is a prefix of } (u_1, \ldots, u_q, y_1, \ldots, y_n)\sigma_0\sigma', \end{array}$$

where $y_1, \ldots, y_n$ are new distinct renaming variables. The proof can be sketched as follows: u_i is either a variable, say y, or a compound term, say $f(s_1, \ldots, s_n)$, because u_i and u_j are unifiable. In the first case, choose $\sigma_0 = \{y \leftarrow f(y_1, \ldots, y_n)\}$ and $\sigma' = \{y_1 \leftarrow u_{j_1}, \ldots, y_n \leftarrow u_{j_n}\}$. In the second case, choose $\sigma_0 = \{y_1 \leftarrow s_1, \ldots, y_n \leftarrow s_n\}$ and $\sigma' = \sigma$. Operation $\texttt{SPECAT}(i, j, \sigma)$ computes σ_0 such that $(u_1, \ldots u_q, y_1, \ldots, y_n)\sigma_0\sigma'$ is a prefix of some $\bar{u}' \in Cc(\sigma')$. Of course, $\bar{u}\sigma$ is also a prefix of $\bar{u}'$.

The above reasoning shows that operation SPECAT allows one to avoid a combinatorial case analysis in the definition of the abstract operation. The cases where a subterm can be a variable, a compound term, or both, are handled uniformly and are reduced to the case where the pattern is known.

7.7.7 The operation UNIF_LIST

Operation 27. $\texttt{UNIF\_LIST}(List, \delta) = \langle \delta', ss \rangle$
This operation is a generalization of $\texttt{UNIF}(i, j, \delta)$.

Specification:
This operation[14] receives as inputs a list of pairs $(i_1, j_1), \ldots, (i_n, j_n)$ and a δ-tuple, and produces as output another δ-tuple, where the terms i_k, j_k have been unified. An information about the sure success (ss) of all the unifications is also returned.

[14]The prototype of this operation has changed relatively to the one proposed in *GAIA* ($\texttt{UNIF\_LIST}(List, \delta) = \delta'$). We now return explicitly an information about the sure success (ss) of the unification.

Implementation:
UNIF_LIST$(List, \delta) = \langle \delta_n, ss \rangle$ where

$$\begin{array}{lcl} \delta_0 & = & \delta \\ ss_0 & = & \left\{ \begin{array}{ll} true & \textbf{if } \delta \neq \bot \\ false & \textbf{otherwise} \end{array} \right. \\ \langle \delta_k, ss_k \rangle & = & \texttt{UNIF}(i_k, j_k, \delta_{k-1}) \quad (1 \leq k \leq n) \end{array}$$

and

$$ss = \bigwedge_{k=0}^{n} ss_k$$

7.8 Unification between Two Substitutions

Operation 28. UNIF_SUBST$(\beta_1, \beta_2) = \beta'$
This operation unifies two abstract substitutions. It is notably used when normalizing a formal specification into an abstract behaviour.

Specification:
Let β_1 and β_2 be two abstract substitutions such that

$$\{X_1, \ldots, X_n\} = dom(\beta_1) = dom(\beta_2).$$

Let θ_1 and θ_2 be two substitutions such that $dom(\theta_i) = dom(\beta_i)$ $(i = 1, 2)$. Let σ be a standard substitution. The following holds:

$$\left. \begin{array}{l} \theta_1 \in Cc(\beta_1) \;\&\; \theta_2 \in Cc(\beta_2) \\ \theta_1\sigma = \theta_2\sigma \end{array} \right\} \;\Rightarrow\; \theta_1\sigma = \theta_2\sigma \in Cc(\beta').$$

We note that this operation is symmetric.

Implementation:
The implementation proceeds in two steps: First it builds a unique term that contains both substitutions. Then it unifies the corresponding arguments of the substitutions.

More precisely:

$$\delta = \langle frm, \langle mo, ty, sv\rangle, fi\rangle \text{ with}$$

$$frm = frm_1 \cup \{\langle i + p_1, f(i_1 + p_1, \ldots, i_n + p_1)\rangle : \langle i, f(i_1, \ldots, i_n)\rangle \in frm_2\}$$

$$fi = id \quad \text{where } id \text{ is the identity function into } I_{p_1+p_2}$$

$$\begin{array}{lcll} mo : I_{p_1+p_2} & \to & Modes & \text{is such that} \\ mo(i) & = & mo_1(i) & \text{if } i \le p_1; \\ & & mo_2(i-p_1) & \text{if } i > p_1. \end{array}$$

$$\begin{array}{lcll} ty : I_{p_1+p_2} & \to & Types & \text{is such that} \\ ty(i) & = & ty_1(i) & \text{if } i \le p_1; \\ & & ty_2(i-p_1) & \text{if } i > p_1. \end{array}$$

$$ps = ps_1 \cup \{(i + p_1, j + p_1) : ps_2(i, j)\}.$$

Then $\texttt{UNIF\_SUBST}(\beta_1, \beta_2)$ produces $\beta' = \langle sv', frm', \alpha'\rangle$, where

$$\begin{array}{rcl} \langle\langle \alpha', fi'\rangle, \bullet\rangle & = & \texttt{UNIF\_LIST}(List, \delta); \\ sv' & = & fi' \circ sv_1; \\ List & = & ((sv_1(X_{i_1}), sv_2(X_1) + p_1), \ldots, (sv_1(X_{i_k}), sv_2(X_k) + p_1)). \end{array}$$

7.9 Unification of Two Variables ($\texttt{UNIF\_VAR}_{subst}$)

Operation 29. $\texttt{UNIF\_VAR}_{subst}(\beta) = \langle \beta', ss, sf, tr, U\rangle$

This operation[15] unifies $X_1\theta$ and $X_2\theta$ for all $\theta \in Cc(\beta)$. 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.

Specification:

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

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

[15]The prototype of this operation has changed relatively to the one proposed in *GAIA* ($\texttt{UNIF\_VAR}_{subst}(\beta) = \beta'$).

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

Implementation:

The implementation of $\texttt{UNIF\_VAR}_{subst}(\beta)$ with $\beta = \langle sv, frm, \alpha \rangle$ produces an abstract substitution $\beta' = \langle sv', frm', \alpha' \rangle$ with:

$$\begin{array}{rcl} \langle \langle frm', \alpha', fi' \rangle, ss \rangle & = & \texttt{UNIF}(sv(X_1), sv(X_2), \langle frm, \alpha, id \rangle); \\ sv' & = & fi' \circ sv \end{array}$$

where id denotes the identity function.
The structural mapping fi' is explicitly returned (i.e., $tr = fi'$).

Further more, this operation computes the two boolean values ss (sure success), returned by the UNIF operation, and sf (sure failure), set to *true* if $\delta' = \bot$.

Finally, we explain the way to build the set of indices U:

$\forall k \in I_p$, we have $k \in U$ iff

$$\begin{array}{ll} (frm(k) = frm'(tr(k)) = undef \ \wedge \ mo(k) = mo'(tr(k)) \in \{ground, var\}) & \vee \\ mo(k) = ground & \vee \\ frm(k) = f & \vee \\ (frm(k) = f(i_1, \ldots, i_n) \ \wedge \ i_1, \ldots, i_n \in U) & \vee \\ frm(k) = [\,] & \vee \\ (frm(k) = [i_1 | i_2] \ \wedge \ i_2 \in U) & \end{array}$$

When the pattern is a list (last case), note that we don't care of the possible size change of i_1 (i.e., we don't check if $i_1 \in U$), because of the norm definition of a list that refer to its list-length.

7.10 Unification of a Variable and a Functor

Operation 30. $\texttt{UNIF\_FUNC}_{subst}(\beta, f) = \langle \beta', ss, sf, tr, U \rangle$

This abstract operation[16] unifies a variable X_1 with a term $f(X_2, \ldots, X_n)$.

Specification:

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

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

[16]The prototype of this operation has changed relatively to the one proposed in *GAIA* ($\texttt{UNIF\_FUNC}_{subst}(\beta, f) = \beta'$).

$ss = true \Rightarrow (\forall\theta \in Cc(\beta) : X_1\theta$ and $f(X_2,\ldots,X_n)\theta$ are unifiable);
$sf = true \Rightarrow (\forall\theta \in Cc(\beta) : X_1\theta$ and $f(X_2,\ldots,X_n)\theta$ are not unifiable).

Implementation:

The implementation is achieved by creating a new subterm $p+1$ containing the new function to unify and unifying terms $p+1$ and $sv(X_1)$.
Assuming that $i_k = sv(X_k)$ $(1 \le l \le n)$, the implementation of UNIF_FUNC(β) produces $\beta' = \langle sv', frm', \alpha'\rangle$, where

$$\begin{array}{rcl} \langle\langle frm', \alpha', fi'\rangle, ss\rangle & = & \texttt{UNIF}(i_1, p+1, \delta_1); \\ sv' & = & fi' \circ sv. \end{array}$$

where $\delta_1 = \langle frm_1, \langle mo_1, ty_1, ps_1\rangle, fi_1\rangle$ is computed below:

$$\begin{array}{rcl} frm_1 & = & frm \cup \{\langle p+1, f(i_2,\ldots,i_n)\rangle\} \\ fi_1 & = & id \text{ where } id \text{ is the identical function into } I_{p+1} \\ mo_1(i) & = & \begin{cases} mo(i) & \text{if } (1 \le i \le p), \\ \texttt{CONS\_MO}(f, mo(i_2),\ldots,mo(i_n)) & \text{if } (i = p+1), \end{cases} \\ ty_1(i) & = & \begin{cases} ty(i) & \text{if } (1 \le i \le p), \\ \texttt{CONS\_TY}(f, ty(i_2),\ldots,ty(i_n)) & \text{if } (i = p+1), \end{cases} \\ ps_1 & = & ps \end{array}$$

We return also explicitly the structural mapping $tr = fi'$. The computation of ss, sf and U is achieved in the same way as UNIF_VAR$_{subst}$.

7.11 Extension of the result of a Goal (EXTG$_{subst}$)

Operation 31. EXTG$_{subst}(l, \beta_1, \beta_2) = \langle\beta', tr_1, tr_2\rangle$

This operation[17] instantiates (abstractly) a clause substitution (i.e., β_1) with the result of the execution of a procedure call[18] (i.e., β_2). It is used after the execution of a literal to propagate the results of the literal to all variables of the clause.

Specification:

Let $X_{i_1},\ldots,X_{i_k}$ be the sequence of variables occurring in the literal l. Let us define $D = \{X_{i_1},\ldots,X_{i_k}\}$. Let β_1 be an abstract substitution such that $\{X_{i_1},\ldots,X_{i_k}\} \subseteq \{X_1,\ldots,X_m\} = dom(\beta_1)$. Let β_2 be an abstract substitution such that $\{X_1,\ldots,X_k\} = dom(\beta_2)$. Let θ_1 and θ_2 be such that $dom(\theta_i) = dom(\beta_i)$ $(i = 1, 2)$. Let σ be a standard substitution. The following holds:

$$\left.\begin{array}{l} \theta_1 \in Cc(\beta_1)\ \&\ \theta_2 \in Cc(\beta_2) \\ (\forall j : 1 \le j \le k : X_j\theta_2 = X_{i_j}\theta_1\sigma) \\ dom(\sigma) \subseteq codom(\theta_{1/D}) \\ (codom(\theta_1) - codom(\theta_{1/D})) \cap codom(\sigma) = \emptyset \end{array}\right\} \Rightarrow \theta_1\sigma \in Cc(\beta').$$

[17]The prototype of this operation has changed relatively to the one proposed in *GAIA* (EXTG$_{subst}(l, \beta_1, \beta_2) = \beta'$). We now return explicitly the two structural mappings tr_1 and tr_2.

[18]Here, a procedure call means the (abstract) execution of a literal (either $p(X_{i_1},\ldots,X_{i_k})$ or a built-in (e.g., the unification $X_{i_1} = X_{i_2}$ or $X_{i_1} = f(X_{i_2},\ldots,X_{i_k})$)).

Further more, $\texttt{EXTG}_{subst}$ returns the two structural mappings $tr_k : I_{p_k} \rightarrow I_{p'}$ between β_k and β' $(k = 1, 2)$.

Implementation:

The implementation proceeds in two steps: First it builds a unique term that contains both substitutions. Then it unifies the corresponding arguments of the substitutions.

More precisely, let $X_{i_1}, \ldots, X_{i_k}$ be the sequence of the variables occurring in l and assume that:

$$\delta = \langle frm, \langle mo, ty, sv \rangle, fi \rangle \text{ with}$$

$$frm = frm_1 \cup \{\langle i + p_1, f(i_1 + p_1, \ldots, i_n + p_1) \rangle : \langle i, f(i_1, \ldots, i_n) \rangle \in frm_2\}$$

$$fi = id \text{ where } id \text{ is the identity function into } I_{p_1+p_2}$$

$$mo : I_{p_1+p_2} \rightarrow Modes \quad \text{is such that}$$
$$mo(i) = \begin{array}{ll} mo_1(i) & \text{if } i \leq p_1; \\ mo_2(i - p_1) & \text{if } i > p_1. \end{array}$$

$$ty : I_{p_1+p_2} \rightarrow Types \quad \text{is such that}$$
$$ty(i) = \begin{array}{ll} ty_1(i) & \text{if } i \leq p_1; \\ ty_2(i - p_1) & \text{if } i > p_1. \end{array}$$

$$ps = ps_1 \cup \{(i + p_1, j + p_1) : ps_2(i, j)\}.$$

Then $\texttt{EXTG}_{subst}(l, \beta_1, \beta_2)$ produces $\beta' = \langle sv', frm', \alpha' \rangle$, where

$$\begin{array}{rcl} \langle \langle \alpha', fi' \rangle, \bullet \rangle & = & \texttt{UNIF\_LIST}(List, \delta); \\ sv' & = & fi' \circ sv_1; \\ List & = & ((sv_1(X_{i_1}), sv_2(X_1) + p_1), \ldots, (sv_1(X_{i_k}), sv_2(X_k) + p_1)). \end{array}$$

Finally, the operation returns tr_1 and tr_2 built in this way:

$$\begin{array}{rcl} tr_1 & = & fi' \circ t^1 \quad \text{with } t^1 \text{ the identity function into } I_{p_1}. \\ tr_2 & = & fi' \circ t^2 \quad \text{with } \forall i \in I_{p_2} : t^2(i) = i + p_1. \end{array}$$

Chapter 8

Operations on Abstract Sequences

See Section 4.3.2 (page 29) where the domain of Abstract Sequences is defined.

The following operations on `Sizes` and on `Abstract Sequences` come from *Automated Verification* [14]. We provide in this Chapter all the implementations (except those which rely on the polyhedra library).

8.1 Constraint mapping

We define first the important notion of *Constraint Mapping* that will be frequently used because it simplifies the expressivity (and also the implementation) of the presented algorithms. Informally, the idea is to restrict or to extend a concrete domain (or abstract domain) while keeping constraints. In our concern, *Constrained Mapping* have been introduced as a formalism to manipulate indices. More precisely,

Definition:
Let I and I' be two finite sets of indices and $tr : I \rightarrow I'$ be a function. The *concrete constrained mapping* of tr is the pair of dual functions, $tr^{>}_{*} : \wp(\mathcal{T}^I) \rightarrow \wp(\mathcal{T}^{I'})$ and $tr^{<}_{*} : \wp(\mathcal{T}^{I'}) \rightarrow \wp(\mathcal{T}^I)$ 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 map-*

ping is any sound approximation $tr^{>} : A_I \to A_{I'}$ and $tr^{<} : A_{I'} \to A_I$ of a concrete one, i.e.,

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

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

Implementation:
A generic implementation of *Constrained Mapping* is discussed in [7]. But in our case, we need only to manipulate indices: you can find in the appendix of [13] a specific implementation for the domain $\texttt{Sizes}_{I_p}$.

8.2 Operations on $\texttt{Sizes}_{I_p}$

See Section 4.3.1 (page 28) where the domain $\texttt{Sizes}_{I_p}$ is defined.

8.2.1 The operation ($\texttt{SUM}_{sol}$)

Operation 32. $\texttt{SUM}_{sol}(E_1, E_2) = E'$
This operation is used to express the length of an abstract sequence (i.e, the number of its solutions) obtained by concatenating two other abstract sequences.

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

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

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.

8.2.2 The operation ($\texttt{MULT}_{sol}$)

Operation 33. $\texttt{MULT}_{sol}(E_1, E_2) = E'$
This operation is used to express the length of an abstract sequence (i.e, the number of its solutions) obtained by "multiplying" two other abstract sequences.

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

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

Implementation:
Let sol_1 and sol_2 be two new variables.

$$E' = tr^{<}_{sol}(E_1[sol \mapsto sol_1] \cup E_2[sol \mapsto sol_2] \cup \{[\![sol = sol_1 * sol_2]\!]\})$$

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.
The problem with this implementation is that the new equation $[\![sol = sol_1 * sol_2]\!]$ is not linear (a solution cannot be calculated with the polyhedra library [15]). A temporary solution can consist of replacing multiplications by additions, if it is possible, as follows:
If sol_1 is a known single value (e.g., $sol_1 = 0$ or $sol_1 = 1$ or $sol_1 = 2$ or ...), then:

$$[\![sol = \underbrace{sol_2 + sol_2 + \cdots + sol_2}_{sol_1 \text{ times}}]\!]$$

If sol_2 is a known single value (e.g., $sol_2 = 0$ or $sol_2 = 1$ or $sol_2 = 2$ or ...), then:

$$[\![sol = \underbrace{sol_1 + sol_1 + \cdots + sol_1}_{sol_2 \text{ times}}]\!]$$

8.3 Extension at Clause Entry (EXTC)

Operation 34. $\texttt{EXTC}(c, \beta) = B$
This operation extends the domain of β to the set of all variables occurring in the clause c.

Specification:
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.

Implementation:
Let $\beta' = \texttt{EXTC}_{subst}(c, \beta)$.

The abstract sequence $B = \langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$ is defined by

$$\begin{aligned} \beta_{in} &= \beta \\ \beta_{ref} &= \beta \\ \beta_{out} &= \beta' \\ E_{ref\_out} &= \{\} \\ E_{sol} &= \{[\![sol = 1]\!]\} \end{aligned}$$

The abstract substitution β_{ref} is identical to β_{in} because the head of the clause is unifiable with any call since it contains distinct variables (indeed remember that the clause is in a normalized form). Similarly, β_{out} is obtained by extending β_{in} with information about the local variables. Since they are brand-new, their mode, type, and sharing information is obviously obtained (this is done by $\texttt{EXTC}_{subst}(c, \beta)$). All size constraints between terms can be inferred by establishing a correspondence between the indices of β_{ref} and those of β_{out}, thus the component E_{ref_out} is empty because we only depict essential constraints. Finally, the component E_{sol} expresses that the unification of the head of the clause succeeds exactly once.

8.4 Restriction at Clause Exit (RESTRC)

Operation 35. $\texttt{RESTRC}(c, B) = B'$

This operation 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.

Specification:

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.

Implementation:

Let $B = \langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$
and let $\langle \beta', tr_{out} \rangle = \texttt{RESTRC}_{subst}(c, \beta_{out})$, where tr_{out} is the structural mapping between β' and β_{out} $(tr_{out} : I'_{out} \rightarrow I_{out})$.
The abstract sequence $B' = \langle \beta'_{in}, \beta'_{ref}, \beta'_{out}, E'_{ref\_out}, E'_{sol} \rangle$ is defined by

$$\begin{aligned} \beta'_{in} &= \beta_{in} \\ \beta'_{ref} &= \beta_{ref} \\ \beta'_{out} &= \beta' \\ E'_{ref\_out} &= ((in_{ref} \circ tr_{ref}) + (in_{out} \circ tr_{out}))^{<}(E_{ref\_out}) \\ E'_{sol} &= E_{sol} \end{aligned}$$

where $tr_{ref} : I'_{ref} \rightarrow I_{ref}$ and is equal to $id_{ref} : I_{ref} \rightarrow I_{ref}$ (that is the identical function), because β_{ref} hasn't changed. Note that E'_{sol} doesn't need to be updated because the sole indices occurring in this component are ones taken from β_{ref} (left unchanged) and from the special indice *sol*.

8.5 Restriction before a Goal (RESTRG)

Operation 36. $\texttt{RESTRG}(l, B) = \beta$
This operation restricts the output domain of B to (a renaming of) the variables occurring in the literal l.

Specification:
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 literal 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 literal is a call of.

Implementation:
Let $B = \langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$.
The result is $\beta = \texttt{RESTRG}_{subst}(l, \beta_{out})$.

8.6 Looking up a behaviour of a Predicate

Operation 37. $\texttt{LOOKUP}(\beta, p, SBeh) = (success, B_{out})$
This operation is 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.

Specification:
$success \Rightarrow \exists se \mid \langle B, se \rangle \in Beh_p \wedge \beta \leq input(B)$.

Implementation:
The implementation is straightforward.

8.7 Checking term sizes for a recursive Call

Operation 38. $\texttt{CHECK\_TERM}(l, B, se) = term$
This operation is 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.

Specification:
If the value *term* is true and the literal l is $p(X_{i_1}, \ldots, X_{i_n})$, then $\forall \langle \theta, S \rangle \in Cc(B)$, we must have:

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

Implementation:
The implementation relies on operations on (in)equation systems. See [15] that contains a library to do such comparisons on linear expressions.

8.8 Unification of Two Variables (UNIF_VAR)

8.8.1 Overview

The operation UNIF_VAR executes the built-ins $X_i = X_j$ at the abstract level. The implementation is as follows: first, we use the version of the operation applying on abstract substitutions, here called UNIF_VAR_{subst}[1], 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 structural mapping 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 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.

8.8.2 Refinement operations

Operation 39. $\text{REF}_{ref}(\beta_1, \beta_2, tr_{1,2}) = \langle \beta', tr' \rangle$
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 $tr_{1,2} : I_1 \rightarrow I_2$ be a structural mapping between β_1 and β_2. $\text{REF}_{ref}(\beta_1, \beta_2, tr_{1,2})$ produces an abstract substitution β' over I' and a structural mapping $tr' : I' \rightarrow I_2$ between β' and β_2 such that $dom(\beta') = dom(\beta_k)$ $(k = 1, 2)$, $\beta' \leq \beta_1$ and

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

[1] See operation 29 (page 70) where the operation UNIF_VAR_{subst} is specified and implemented.

Implementation:

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

where the operations $\mathtt{REF}_{frm}$ and $\mathtt{REF}_{\alpha}$ are described below.

Operation 40. $\mathtt{REF}_{frm}(\beta_1, \beta_2, tr_{1,2}) = \langle \beta', tr' \rangle$

It refines the abstract substitution β_1 only using the frame component of β_2.

Specification:

The specification is the same as the one of $\mathtt{REF}_{ref}$.

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
   $mo^i(j) \leq novar$, $frm^i(j) = undef$ and $frm_2(tr^i(j)) = f(k_1, \ldots, k_n)$.
   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;
   - $sv^{i+1} = sv^i$;
   - $frm^{i+1} = frm^i \cup \{j \mapsto f(j_1, \ldots, j_n)\}$;
   - $tr^{i+1} = tr^i \cup \{j_1 \mapsto k_1, \ldots, j_n \mapsto k_n\}$;
   - $mo^{i+1}(j) = mo^i(j)$ for all $j \in I^i$ and
     $\langle mo^{i+1}(j_1), \ldots, mo^{i+1}(j_n) \rangle = \mathtt{EXTR\_MO}(f, mo^i(j))$;
   - $ty^{i+1}(j) = ty^i(j)$ for all $j \in I^i$ and
     $\langle ty^{i+1}(j_1), \ldots, ty^{i+1}(j_n) \rangle = \mathtt{EXTR\_TY}(f, ty^i(j))$;
   - $ps^{i+1} = ps^i \cup \{(j_l, k) |\ l \in \{1, \ldots, n\},$
     $mo^{i+1}(j_l) \neq ground,$
     $(j, k) \in ps^i\}$.

3. Otherwise, $\beta' = \beta^i$ and $tr' = tr^i$.

Operation 41. $\mathtt{REF}_{\alpha}(\beta_1, \beta_2, tr_{1,2}) = \langle \beta', tr' \rangle$

It refines β_1 only considering the α component of β_2.

Specification:

The specification is the same as the one of $\mathtt{REF}_{ref}$.

Implementation:
The implementation is as follows:

$$
\begin{array}{lcll}
I' & = & I_1 & \\
sv' & = & sv_1 & \\
frm' & = & frm_1 & \\
mo'(i) & = & mo_1(i) \sqcap \texttt{UNIST\_MO}(mo_2(tr_{1,2}(i))) & \text{for all } i \in I' \\
ty'(i) & = & ty_1(i) \sqcap \texttt{UNIST\_TY}(ty_2(tr_{1,2}(i))) & \text{for all } i \in I' \\
ps' & = & ps_1 & \\
tr' & = & tr_{1,2}. &
\end{array}
$$

8.8.3 Unification of two variables

We are now in position to define UNIF_VAR.

Operation 42. $\texttt{UNIF\_VAR}(\beta) = B'$
This operation executes the unification $X_1 = X_2$ on the abstract substitution β.

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

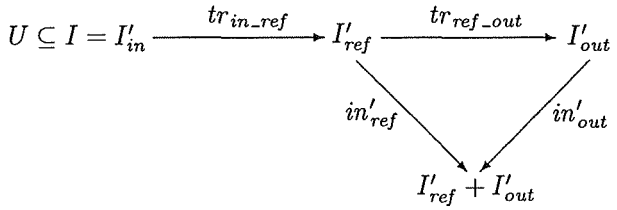
$$
\left.\begin{array}{l} \theta \in Cc(\beta) \\ \sigma \in mgu(X_1\theta, X_2\theta) \end{array}\right\} \Rightarrow \langle \theta, < \theta\sigma > \rangle \in Cc(B')
$$

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

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

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

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}.



8.9 Unification of a Variable and a Functor

Operation 43. $\texttt{UNIF\_FUNC}(\beta, f) = B'$

This operation executes the unification $X_1 = f(X_2, \ldots, X_n)$ on the abstract substitution β, where $n-1$ is the arity of f.

Specification:

The specification is similar to the UNIF_VAR one.

Implementation:

The implementation is identical to UNIF_VAR, where you replace the call to $\texttt{UNIF\_VAR}_{subst}$ by a call to $\texttt{UNIF\_FUNC}_{subst}$.

8.10 Abstract Concatenation (CONC)

Operation 44. $\texttt{CONC}(B_1, B_2) = B'$

This operation is used to concatenate the (abstract) results obtained from the execution of a procedure and a clause. It is the counterpart for abstract sequences of the operation UNION, used in [8], 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. 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 $\bot$, the clauses are exclusive, and no sum is computed: we only collect the numbers of solutions of the two clauses.

Specification:

Let $B_k = \langle \beta_{in}, \beta^k_{ref}, \beta^k_{out}, E^k_{ref\_out}, E^k_{sol} \rangle$ $(k = 1, 2)$ be two abstract sequences with $dom_{out}(B_1) = dom_{out}(B_2)$. $\texttt{CONC}(B_1, B_2)$ returns an abstract sequence 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.
$B' = \langle \beta'_{in}, \beta'_{ref}, \beta'_{out}, E'_{ref\_out}, E'_{sol} \rangle$ with:

$$
\begin{array}{lcl}
\beta'_{in} & = & \beta_{in} \\
\langle \beta'_{ref}, tr^1_{ref}, tr^2_{ref}, st \rangle & = & \mathtt{EXT\_LUB}(\beta^1_{ref}, \beta^2_{ref}) \\
\langle \beta'_{out}, tr^1_{out}, tr^2_{out} \rangle & = & \mathtt{LUB}(\beta^1_{out}, \beta^2_{out}) \\
E'_{ref\_out} & = & ((in_{ref} \circ tr^1_{ref}) + (in_{out} \circ tr^1_{out}))^{<}(E^1_{ref\_out}) \\
 & & \sqcup \\
 & & ((in_{ref} \circ tr^2_{ref}) + (in_{out} \circ tr^2_{out}))^{<}(E^2_{ref\_out})
\end{array}
$$

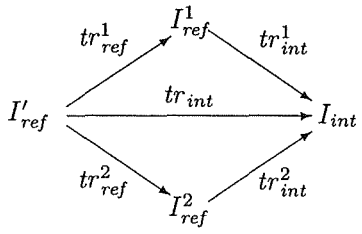
$$
E'_{sol} = \begin{cases}
(tr^1_{ref} + \{sol \mapsto sol\})^{<}(E^1_{sol}) \sqcup \\
(tr^2_{ref} + \{sol \mapsto sol\})^{<}(E^2_{sol}) \sqcup & \text{if } st \\
(tr_{int} + \{sol \mapsto sol\})^{<}(\mathtt{SUM}_{sol}(\overline{E}^1_{sol}, \overline{E}^2_{sol})) & \\
 & \\
(tr^1_{ref} + \{sol \mapsto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

where

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

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

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



The least upper bound operator $\sqcup$ between (in)equation systems is implemented as convex union (see [15]).

8.11 Extension of the result of a Goal (EXTG)

Operation 45. $\texttt{EXTG}(l, B_1, B_2) = B'$

This operation computes 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 literal l on B_1 can be computed as an instantiation by some substitution, which yields B_2 (when applied on $\texttt{RESTRG}(l, B_1)$). The operation EXTG extends the effect of the instantiation on the whole sequence B_1 (taking into account necessary renaming to avoid name clashes).

Specification:

The abstract substitution B' must satisfy the following property. For all $\langle \theta, < \theta_1, \ldots, \theta_n > \rangle \in Cc(B_1)$, if $\theta'_k = \texttt{RESTRG}(l, \theta_k)$, $\langle \theta'_k, S'_k \rangle \in Cc(B_2)$, and $S_k = \texttt{EXTG}(l, \theta_k, S'_k)$ for every $k \leq n$, then $\langle \theta, S_1 :: \ldots :: S_n \rangle \in Cc(B')$.

Implementation:

Let $X_{i_1}, \ldots, X_{i_k}$ be the sequence of variables occurring in the literal l.
Let $B_k = \langle \beta^k_{in}, \beta^k_{ref}, \beta^k_{out}, E^k_{ref\_out}, E^k_{sol} \rangle$ $(k = 1, 2)$.
$\texttt{EXTG}(l, B_1, B_2)$ constructs $B' = \langle \beta'_{in}, \beta'_{ref}, \beta'_{out}, E'_{ref\_out}, E'_{sol} \rangle$ as follows[2]:

$$
\begin{array}{rcl}
\beta'_{in} & = & \beta^1_{in} \\
\langle \beta'_{out}, tr^1_{out}, tr^2_{out} \rangle & = & \texttt{EXTG}_{subst}(l, \beta^1_{out}, \beta^2_{out}) \\
\langle \beta'_{ref}, \bullet \rangle & = & \texttt{REF}_{ref}(\beta^1_{ref}, \beta'_{out}, tr_{ref^1\_out'}) \\
E'_{ref\_out} & = & ((in'_{ref} \circ tr^1_{ref}) + (in'_{out} \circ tr^1_{out}))^{>}(E^1_{ref\_out}) \\
 & & \sqcap \\
 & & ((in'_{ref} \circ tr^2_{ref}) + (in'_{out} \circ tr^2_{out}))^{>}(E^2_{ref\_out}) \\
E'_{sol} & = & (tr_{int} + \{sol \mapsto sol\})^{<}(\texttt{MULT}_{sol}(\overline{E}^1_{sol}, \overline{E}^2_{sol}))
\end{array}
$$

where

$$
\begin{array}{lcl}
\langle \beta_{int}, tr^1_{int}, tr^2_{int} \rangle & = & \texttt{GLB}(\beta^1_{ref}, \beta^*_{ref}) \\
\overline{E}^1_{sol} & = & (tr^1_{int} + \{sol \mapsto sol\})^{>}(E^1_{sol}) \\
\overline{E}^2_{sol} & = & (tr^2_{int} + \{sol \mapsto sol\})^{>}(E^2_{sol})
\end{array}
$$

with β^*_{ref} constructed from β^2_{ref} by renaming and extending the domain of β^2_{ref}. More precisely, $dom(\beta^2_{ref}) = \{X_1, \ldots, X_k\}$ is extended by first renaming $X_1, \ldots, X_k$ to $X_{i_1}, \ldots, X_{i_k}$ (that is the effective sequence of variables occurring in the literal l) and then by adding the variables belonging to the domain of β^1_{ref} that does not appear yet in β^*_{ref} (the implementation of this extension is exactly the same as $\texttt{EXTC}_{subst}$, page 55).
tr_{int} denotes the structural mapping between β'_{ref} and β_{int}.
The greatest lower bound operator $\sqcap$ between (in)equation systems can be implemented using [15].

[2] $tr_{ref^1\_out'}$ denotes the structural mapping between β^1_{ref} and β'_{out}.

Chapter 9

Coding

In this Chapter, we briefly discuss the development effort concerning the coding in Java of our analyser described so far. Unfortunately, the work is still under progress; indeed the operations related to the size components based on the polyhedron library of D.K. Wilde [15] have not been implemented yet.

9.1 Data Structures

Most data structures used to implement the analyser are the simple transformations of the ones presented in the previous Chapters.

Indeed our objective was not to implement efficiently the analyser[1]. Our choice was to remain very close to the paper description. This manner to proceed was very interesting for debugging purpose because of the readability of the Java code. Further more, the code can be extended with minor effort by adding more sophisticated abstract domains.

Java belongs to the Object-Oriented Programmation paradigm: it results that all (more or less) is an object. Each domain is then represented by a class (e.g., Mode, Type, ASubst, ASeq, ...). The sole notable exception is the domain of abstract tuples, that does not correspond to any class. The reason is that the domain of abstract substitutions was "flattened" (like it was done in *GAIA* [8]), in order to make easier the implementation. What about the abstract operations, they were just inserted in the classes describing their corresponding abstract domains.

We have of course implemented the concept of mathematical relations, symmetrical relations and functions. Those classes belong to the package `mystructure`.

[1]Note that this kind of analyser will usually be used only one time, typically just before or just after the compilation process: therefore the efficiency is not a major issue.

9.2 Description of the library

The library is composed of 75 files (among which 14 files were automatically generated by JavaCC).

In Table 9.1 we give a brief description of the different Java packages.

Package	Description of the package
root package	Contains the launcher and the abstract semantics of the analyser.
myio	Contains some basic operations on input/output with files.
mystructure	Contains the structures we use to implement relations, functions, ...
parsing.prolog	Contains the parser of Prolog procedures.
parsing.beh	Contains the parser of formal specifications.
program	Contains the abstract tree representing a Prolog program.
spec	Contains the abstract tree representing a formal specification.
adom	Contains all the abstract domains and the abstract operations.

Table 9.1: Description of the packages.

You will find in Appendix D more information about the content of those packages, and some UML-like diagrams report the organization inside the packages (showing the structural relations between classes).

Chapter 10

Conclusion

10.1 Contribution of this paper

In this report, we have presented the theoretical and methodological aspects of a generic analyser for Prolog programs based on a verification approach. For that purpose, we "unify" the concepts and notations of different references (mainly [7], [8], [13] and [14]).

A complete domain of abstract sequences has been presented. This domain allowed 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 algorithms of all the abstract operations have been fully described.

We have provided an implementation of the analyser in Java. At the time of writing, the implementation is still underway, since the abstract domain is complex: we have been able to rewrite most of the code of *GAIA* [8] but we still have to implement the operations related to the size components based on the polyhedron library of D.K. Wilde [15].

Hence, we have built a practical system in which some state-of-the-art techniques of Prolog program verification have been integrated.

10.2 Towards a full analyser

Implementing a complete analyser is a long term project however, since it entails **1)** to cover other important features of Prolog such as arithmetic built-ins, test predicates, negation by failure, and the cut, **2)** to implement a large number of (cooperating) abstract domains, and **3)** to extend the scope of the analyser to

some classes of non terminating programs. Let us explain shortly how to reach these three points.

Analyzing (almost) full Prolog. Although our analyser has been presented for pure Prolog, it can be readily extended to deal with most non pure features of Prolog.
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.
The treatment of the cut requires to enhance the concrete and abstract domains with so-called "cut information" in the style of [9]; 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 modelled through the cut, it can also be handled simply.
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.

Implementing a complete set of domains. The abstract domain presented in this paper is conceptually generic. However the particular instance that we have described is able to handle programs dealing with lists accurately, but not other programs. A further step can then be to extend the analyser with more powerful abstract domains for types. We can also improve the treatment of sharing by adding a complementary domain for linearity information. Finally we could attempt to design more powerful domains for the size components, based on non linear constraints and/or computer algebra.

Extending the verification scope of the analyser. Some aspects of Prolog program verification have not been deal with this paper. A first issue is the "occur check problem". Our analyser assumes that the occur-check is performed during unification. It is nevertheless straightforward to enhance the operations `UNIF_VAR` and `UNIF_FUNC` with an additional result parameter specifying whether the occur-check is needed or not. Classical abstract domains (including sharing and linearity information) will then allow us to solve the problem quite satisfactorily. A second issue is the fact that our analyser cannot deal with non terminating procedures at all (i.e., it always reject them), but yet such procedures can sometimes be considered as correct if they "produce all their solutions" or if the user is interested in merely "existential" termination (i.e., the procedures produce at least one solution).

Let's go towards a full analyser!

Appendix

Appendix A

Syntax of Pure Prolog and of specifications

A.1 ABNF conventions

In this Appendix, we give the representation conventions of **ABNF** (**A**ugmented **B**ackus/**N**aur **F**orm) we adopt to explicit the concrete syntax of Prolog procedures and of their formal specifications:

- Nonterminals are written between angular brackets < >
- "x" is a terminal and denotes the string x
- The meta-symbol | means choice
- [a..z] is a shortcut to denote "a" | "b" | ... | "z"
- [x] denotes that x is optional
- (x)* denotes multiplicity (none, one or more occurrences of x)
- (x)+ denotes proper multiplicity (one or more occurrences of x)
- <EOF> denotes the End Of the considered File
- Between two tokens (terminals) all possible sequences of spaces (" " or "\t") or carriage returns ("\r" or "\n") are accepted.

We have chosen the JavaCC tool (Java Compiler-Compiler) to implement the parsing of our defined grammars. The transition from the ABNF syntax to the JavaCC syntax is quite easy to understand. However some transformations about the structure of the given ABNF grammars were needed to remove some left recursion issues and to make it LL(1) for its major part. So the parser will be more efficient because it has to lookahead only one token to decide which

rules taking.

Further more Java codes were added in the action parts of these grammars where we construct the related abstract tree.

The user must read the ABNF form that defines the same language at any way. Indeed, he needs just to know the flat representation, without any concerns about the optimality of such one (found in the JavaCC representation).

A.2 Concrete syntax of Pure Prolog procedures

We give now the concrete syntax in ABNF of the file containing the Prolog procedures given by the user.

All texts following a symbol "%" until the end of the line and texts between the symbols "/*" and "*/" are comments. They will not be considered by our parser (the latter will just pass across).

Our parser will accept a subset of the Pure Prolog language:

```
<FileOfProcedures>  ::=    (<clauses>)* <EOF>

<clause>            ::=    <predicate> [":-" <goal>] "."

<predicate>         ::=    <ident> ["(" <terms> ")"]

<ident>             ::=    <letter-lo>(<letter>|<digit>|<underscore>)*
<letter>            ::=    <letter-lo> | <letter-up>
<letter-lo>         ::=    [a..z]
<letter-up>         ::=    [A..Z]
<digit>             ::=    [0..9]
<natural>           ::=    0 | ([1..9] <digit>*)
<underscore>        ::=    "_"

<terms>             ::=    <term> ("," <term>)*
<term>              ::=    <variable> | <functor> | <natural>

<variable>          ::=    (<letter-up>|<underscore>)
                                    (<letter>|<digit>|<underscore>)*

<functor>           ::=    <ident> ["(" <terms> ")"]
                       |   "[" [<cont-list-terms>] "]"
<cont-list-terms>   ::=    <terms> ["|" <term>]

<goal>              ::=    <literal> ("," <literal>)*
```

```
<literal>          ::=    <unification> | <predicate>

<unification>      ::=    <term> "=" <term>
```

A.3 Concrete syntax of formal specifications

Formal specifications given by the user are required to be defined according to the following grammar.

All texts following a symbol "%" until the end of the line and texts between the symbols "/*" and "*/" are comments. They will not be considered by our parser (the latter will just pass across).

```
<FileOfSpecifications> ::=  (<specification>)* <EOF>

<specification>  ::=  <proc-name>
                          "("
                             <in-part>   ","
                             <ref-part>  ","
                             <out-part>  ","
                             <srel-part> ","
                             <sol-part>  ","
                             <sexpr-part>
                           ")"

<proc-name>      ::=  <identifier-lo>
<func-name>      ::=  <identifier-lo>
<arg-name>       ::=  <identifier-up>
<identifier-lo>  ::=  <letter-lo> (<letter>|<digit>|<underscore>)*
<identifier-up>  ::=  <letter-up> (<letter>|<digit>|<underscore>)*

<letter>         ::=  <letter-lo> | <letter-up>
<letter-lo>      ::=  [a..z]
<letter-up>      ::=  [A..Z]
<digit>          ::=  [0..9]
<natural>        ::=  0 | ([1..9] <digit>*)
<underscore>     ::=  "_"

<tag>            ::=  "_ref" | "_out"
<tag-arg-name>   ::=  <arg-name> <tag>

<in-part>        ::=  "in"    "(" [<abstr-subst>] ")"
<ref-part>       ::=  "ref"   "(" [<abstr-subst>] ")"
<out-part>       ::=  "out"   "(" [<abstr-subst>] ")"
```

```
<srel-part>       ::=  "srel"  "(" [<size-relation>] ")"
<sol-part>        ::=  "sol"   "(" [<sol-relation>] ")"
<sexpr-part>      ::=  "sexpr" "(" [<expression>] ")"

<abstr-subst>     ::=  <args-list> [";" <noshare-decl>]
<args-list>       ::=  <arg> ["," <args-list>]

<arg>             ::=  [<arg-name> ":"]
                           (
                             <underscore>
                           | <mode>   [<type>] [<frame>]
                           | [<mode>]  <type>  [<frame>]
                           | [<mode>] [<type>]  <frame>
                           )

<mode>            ::=  "ground" | "noground" | "var" |
                       "novar" | "gv" | "ngv" | "any"
<type>            ::=  "list" | "anylist" | "any"

<frame>           ::=  <func-name> ["(" <args-list> ")"] |
                       "[" [<cont-list-args>] "]" |
                       <natural>
<cont-list-args>  ::=  <args-list> ["|" <arg>]

<noshare-decl>    ::=  "noshare" "=" "{" [<noshare-list>] "}"
<noshare-list>    ::=  <noshare-pair> ["," <noshare-list>]
<noshare-pair>    ::=  "(" <arg-name> "," <arg-name> ")"

<size-relation>   ::=  <inequality> ["," <size-relation>]
<sol-relation>    ::=  <sol-inequality> ["," <sol-relation>]
<sexpr-expr>      ::=  <expression>

<inequality>      ::=  <tag-expression> <ineq-operator> <tag-expression>
<sol-inequality>  ::=  "sol" <ineq-operator> <tag-expression> |
                       <tag-expression> <ineq-operator> "sol"
<ineq-operator>   ::=  "=" | "<="

<expression>      ::=  <arg-name> | <natural> |
                       <expression> <operator> <expression> |
                       "(" <expression> ")"
<tag-expression>  ::=  <tag-arg-name> | <natural> |
                       <tag-expression> <operator> <tag-expression> |
                       "(" <tag-expression> ")"
<operator>        ::=  "+" | "-" | "*"
```

The "semantics" of such a formal specification containing information about

modes, types, possible variable sharing between arguments, sizes relations, number of solutions, termination, is explained informally in Section 2.3, based on a simple example. Formally, the semantics is explicated in terms of a(n) (abstract) behaviour, which is one of the objects of our abstract domains that "packages" this kind of information. This "translation process" is explained in Appendix B.

Appendix B

From a formal specification to a behaviour

In this Appendix, we want to explicit the "transformational semantics" of a formal specification provided by the user. In other words, it consists of explicating how to transform such a formal specification into its related (abstract) behaviour structure.

For that purpose, we will use some of the abstract operations which are specified and implemented in the Chapters 6, 7 and 8.

The process that we will describe here is done by applying successively rewriting and refinement rules, and by detecting inconsistencies on data encoded by the user. This "normalization process" will also be able to infer some information let understood by the user (notably due to the use of the terminal symbol "_").

Note that the meaning of the underscore differs according to the normalization process occurs "*within*" an abstract substitution (i.e., the formal object `<abstr-subst>`) or "*between*" abstract substitutions (between `<in-part>` and `<ref-part>` or between `<ref-part>` and `<out-part>`). In the first case, an argument annotated with an underscore (e.g., `A:_`) is used to mean that its mode is any, its type is any and its frame (or pattern) is *undef*. In the second case, occurrences of the symbol "_" in the `ref` part means that the information about the corresponding argument cannot be refined with respect to the `in` part. An underscore in the `out` part means that we give no specific information about the nature of the related argument after the execution of the logic procedure ; the pattern of that argument cannot be refined with respect to the `ref` part. More generally, the user is allowed to omit from the specification all pieces of information which can be inferred from another part.

The rest of this Appendix is subdivided into three parts: we explain how

to normalize an argument (i.e., <arg>), then an abstract substitution (i.e., <abstr-subst>) and finally a formal specification (i.e., <specification>).

B.1 Normalize an argument NORM_ARG(a,e_mo_a,e_ty_a)=a'

An argument a provided by the user respects the following syntax:

```
<arg> ::= [<arg-name> ":"]
          (
            <underscore>
          | <mode>   [<type>] [<frame>]
          | [<mode>]  <type>  [<frame>]
          | [<mode>] [<type>]  <frame>
          )
```

By now, we consider only the canonical form of a:

$$name_a \ : \ mo_a \ \ ty_a \ \ frm_a$$

where

$name_a$	denotes the user name of the argument (<arg-name>); if not specified, $name_a$ is noted "?".
mo_a	denotes the argument mode (<mode>); if not specified, mo_a is any.
ty_a	denotes the argument type (<type>); if not specified, ty_a is any.
frm_a	denotes the argument pattern $f(a_1,\ldots,a_n)$ (<frame>); if not specified, frm_a is noted "?". $a_1,\ldots,a_n$ are called the sub-arguments of a.

Starting from this canonical form of a, we will proceed by successive refinements on its mode and its type. To infer the modes and types of the sub-arguments of a, we use the notion of "expected mode" (e_mo_a) and "expected type" (e_ty_a) of an argument, meaning that the mode of the argument a must be at least "as constrained as" the mode e_mo_a and that the type of a must be at least "as constrained as" the type e_ty_a.

The normalization process of an argument - namely NORM_ARG(a, e_mo_a, e_ty_a) - refines mo_a and ty_a, following the next four steps. Note that the process stops and fails when an error occurs (e.g., if an abstract operation returns $\perp$).

1. For some kinds of mode, it is possible to refine the argument type:

   If (mo_a , ty_a) = (var , any) then ty_a = anylist, because any variable can be instantiated to a list.

If (mo_a , ty_a) = (ground , anylist) then ty_a = `list`, because a ground term that has the type `anylist` is in fact already instantiated to a list.

2. We have to ensure the consistency between the current mode and the expected mode, and between the current type and the expected type. It possibly refines mo_a and/or ty_a:

$$\begin{array}{lcl} mo_a & = & \texttt{GLB}(mo_a, e\_mo_a) \\ ty_a & = & \texttt{GLB}(ty_a, e\_ty_a) \end{array}$$

3. If frm_a is defined (e.g., $frm_a = f(a_1, \ldots, a_n)$), we can build the expected modes and expected types for $a_1, \ldots, a_n$ as follows[1]:

$$\begin{array}{lcl} \langle e\_mo_{a_1}, \ldots, e\_mo_{a_n} \rangle & = & \texttt{EXTR\_MO}(f/n, mo_a) \\ \langle e\_ty_{a_1}, \ldots, e\_ty_{a_n} \rangle & = & \texttt{EXTR\_TY}(f/n, ty_a) \end{array}$$

We can then execute $\texttt{NORM\_ARG}(a_i, e\_mo_{a_i}, e\_ty_{a_i})$ to refine the sub-arguments a_i. Therefore, at this process stage, we have that all the mo_{a_i} and ty_{a_i} are computed and refined.

4. Finally, we compute mo'_a and ty'_a, taking into account the modes and types of the sub-arguments[2]:

$$\begin{array}{lcl} mo'_a & = & \texttt{GLB}(mo_a, \texttt{CONS\_MO}(f/n, \langle mo_{a_1}, \ldots, mo_{a_n} \rangle)) \\ ty'_a & = & \texttt{GLB}(ty_a, \texttt{CONS\_TY}(f/n, \langle ty_{a_1}, \ldots, ty_{a_n} \rangle)) \end{array}$$

Let us show you some illustrations of the normalization process of an argument:

argument specification	normalized canonical form
`A: _`	`A: any any ?`
`B: var any`	`B: var anylist ?`
`C: ground anylist`	`C: ground list ?`
`D: f(ground,ground)`	`D: ground any f(ground any ?,ground any ?)`
`E: ground list [_|_]`	`E: ground list [ground any ?|ground list ?])`

And now, let us show you some examples of inconsistencies that will be detected:

argument specification	kind of error
`A: any list f(_)`	`f/1 is not the pattern of a list.`
`B: var any f(_)`	`a variable cannot have a pattern.`
`C: ground f(var)`	`f(var) cannot be ground.`

We are now in position to explain how to normalize an abstract substitution.

[1] See page 42 for the EXTR_MO operation and see page 46 for the EXTR_TY operation.
[2] See page 42 for the CONS_MO operation and see page 46 for the CONS_TY operation.

B.2 Normalize an abstract substitution NORM_ASUBST(<abstr-subst>)=β

The following abstract substitution specification (provided by the user)

<abstr-subst> ::= a_1 , ... , a_n;
noshare = $\{(name_{a_{i_1}}, name_{a_{j_1}}), \ldots, (name_{a_{i_p}}, name_{a_{j_p}})\}$

will be normalized and "paquetized" into an abstract substitution

$$\beta = \langle sv, frm, mo, ty, ps \rangle.$$

The arguments $a_1, \ldots, a_n$ are called the "main arguments" of the abstract substitution. The arguments which compose these main arguments are called the "sub-arguments". As an example, the abstract substitution

```
A:var , B:f(X:_,Y:_) ; noshare={(X,Y)}
```

has two main arguments (A and B) and B is composed of two sub-arguments (X and Y).

In order to construct β, we have first to normalize and refine each argument (the main arguments and the sub-arguments) occurring in <abstr-subst>. It is done by applying the previous operation NORM_ARG to every argument.
Next, we have to bind an indice to each argument. It consists of the construction of I_p, with its subset I_m which contains the indices of the main arguments. Each distinct argument has an distinct indice. Two arguments are equals if they have the same name. Note that two arguments without specified name (i.e., *name* =?) are distinct and therefore are attached to a distinct indice. In the next, we called i_a the corresponding indice of the argument a.

The same-value, frame, mode and type components of β are computed as follows:

$$\begin{array}{rcll} sv(X_j) & = & i_{a_j} \text{ for each main argument } a_j & \\ frm(i_a) & = & \begin{cases} f(i_{a_1}, \ldots, i_{a_n}) & \text{if } frm_a = f(a_1, \ldots, a_n) \\ undef & \text{if } frm_a = ? \end{cases} & \forall i_a \in I_p \\ mo(i_a) & = & mo_a \quad \forall i_a \in I_p & \\ ty(i_a) & = & ty_a \quad \forall i_a \in I_p & \end{array}$$

Detection of errors due to the noshare *component.*
Replace each $name_{a_{i_k}}$ in noshare by its corresponding indice, so that we can view it as a set of indice pairs. Then, for each $(i, j) \in$ noshare, we have to check if (i, j) may or not belong to noshare (if it is not the case, the process fails).
An error occurs if there exists $(i, j) \in$ noshare such that one of these conditions happens:

- $i = j \ \wedge \ mo(i) = mo(j) \leq noground$

- $i \neq j \ \wedge \ frm(j) = f(...,i,...) \ \wedge \ mo(i) \leq noground$
- $i \neq j \ \wedge \ frm(i) = f(...,j,...) \ \wedge \ mo(j) \leq noground$
- $i \neq j \wedge frm(i) = f(...,k,...) \wedge frm(j) = g(...,k,...) \wedge mo(k) \leq noground$

Refinement of some arguments modes.
Suppose that no inconsistence were detected up to now. Let $(i,j) \in$ noshare. In the following cases, the mode component mo of β can be updated:

- $i = j \ \} \Rightarrow mo(i) = mo(j) = ground$
- $\left.\begin{array}{l} i \neq j \\ frm(j) = f(...,i,...) \end{array}\right\} \Rightarrow mo(i) = ground$
- $\left.\begin{array}{l} i \neq j \\ frm(i) = f(...,j,...) \end{array}\right\} \Rightarrow mo(j) = ground$
- $\left.\begin{array}{l} i \neq j \\ frm(i) = f(...,k,...) \\ frm(j) = g(...,k,...) \end{array}\right\} \Rightarrow mo(k) = ground$

Construction of the ps-component of β.
First we build the set $NOSHARE$ satisfying the two following rules:

$$(1) \qquad (i,j) \in \texttt{noshare} \qquad \Rightarrow (i,j) \in NOSHARE$$

$$(2) \qquad \left.\begin{array}{l} (i,j) \in NOSHARE \\ i \neq j \\ frm(i) = f(...,k,...) \\ frm(j) = g(...,l,...) \end{array}\right\} \Rightarrow (k,l) \in NOSHARE$$

We are now in position to construct ps as follows:

$$ps = ps_{aux} \setminus NOSHARE$$

where

$$ps_{aux} = \{(i,j) \mid i,j \in I_p \wedge mo(i) \neq ground \wedge mo(j) \neq ground\}$$

We are now in position to explain the semantics of a formal specification.

B.3 Normalize a specification NORM_SPEC(specification)=$\langle B, se \rangle$

The following formal specification

```
<specification> ::= p(<in-part>,
                      <ref-part>,
                      <out-part>,
                      <srel-part>,
                      <sol-part>,
                      <sexpr-part>)
```

will be refined, transformed and "paquetized" into the behaviour $\langle B, se \rangle$ which will belong to Beh_p, where:

- $B = \langle \beta_{in}, \beta_{ref}, \beta_{out}, E_{ref\_out}, E_{sol} \rangle$ is built from the first five parts.
- se is built from the sexpr part.

We decompose this problem in the three next steps. Note that when an inconsistency is detected (e.g., if an abstract operation returns $\perp$), the whole process stops.

1. Normalization process "*within*" the abstract substitutions.

   $$\begin{array}{lcl} \beta_{in} & = & \texttt{NORM\_ASUBST(<in-part>)} \\ \beta_{ref}^{aux} & = & \texttt{NORM\_ASUBST(<ref-part>)} \\ \beta_{out}^{aux} & = & \texttt{NORM\_ASUBST(<out-part>)} \end{array}$$

   where the operation NORM_ASUBST(<abstr-subst>) is described in the part "Normalize an abstract substitution" of this Section.

   At this process stage, we can detect all the inconsistencies occurring within an abstract substitution.

2. Normalization process "*between*" the abstract substitutions[3].

   β_{ref} and β_{out} are computed as follows:

   $$\begin{array}{lcl} \beta_{ref} & = & \texttt{GLB}(\beta_{in}, \beta_{ref}^{aux}) \\ \beta_{out} & = & \texttt{GLB}(\beta_{out}^{aux}, \texttt{UNIF\_SUBST}(\beta_{ref}, \beta_{out}^{aux})) \end{array}$$

   At this process stage, we can detect the following kinds of inconsistencies:

   - The domains of the three abstract substitutions β_{in}, β_{ref} and β_{out} are not the same.
     Example:

     ```
     <in-part>  ::=  A:_
     <ref-part> ::=  A:_ , B:_ , C:_
     <out-part> ::=  A:_ , B:_
     ```

   - The greatest lower bound between the abstract substitutions β_{in} and β_{ref} fails.
     Example:

[3] See page 53 for the GLB operation and see page 69 for the UNIF_SUBST operation.

```
<in-part>  ::=  A:var
<ref-part> ::=  A:ground
```

- The unification between the abstract substitutions β_{ref} and β_{out} fails. Example:

```
<ref-part> ::=  A:f(_,_)
<out-part> ::=  A:g(_)
```

3. Normalization of the `srel`, `sol` and `sexpr` parts into E_{ref_out}, E_{sol} and se respectively.

   For doing this job, we have to keep in memory the mappings that bind each user argument name with its corresponding indice (map_{ref} and map_{out}) and a mapping between the main argument names and their related normalized variables:

$$\begin{array}{lccc} map_{ref}: & UserArgs_{ref} & \rightarrow & I_{p_{ref}} \\ & \texttt{A\_ref} & \mapsto & i \end{array}$$

$$\begin{array}{lccc} map_{out}: & UserArgs_{out} & \rightarrow & I_{p_{out}} \\ & \texttt{A\_out} & \mapsto & i \end{array}$$

$$\begin{array}{lccc} map_{norm}: & UserArgs & \not\rightarrow & NormVars \\ & \texttt{A} & \mapsto & X_i \end{array}$$

   In fact, these mappings will be constructed during the normalization process acting on the `in`, `ref` and `out` parts (in the two previous points).

   We can then construct the two injection functions in_{ref} and in_{out}:

$$in_{ref}: \; I_{p_{ref}} \rightarrow I_{p_{ref}} + I_{p_{out}}$$

$$in_{out}: \; I_{p_{out}} \rightarrow I_{p_{ref}} + I_{p_{out}}$$

   Finally, to obtain E_{ref_out}, E_{sol} and se, we have just to replace each argument name occurring in `<srel-part>`, `<sol-part>` and `<sexpr-part>` by its corresponding indice or related normalized variable, applying the rules depicted in the following table:

argument name	replacement
A_ref	$in_{ref} \circ map_{ref}$(A_ref)
A_out	$in_{out} \circ map_{out}$(A_out)
A	map_{norm}(A)

Appendix C

Normalization of a program

The semantics and algorithms of the analyser were defined on normalized logic programs. In this Appendix, we first provide the syntax of normalized programs and then we provide the algorithm that performs the normalization process.

C.1 Syntax of normalized programs

We show the abstract syntax of normalized programs, given by B. Le Charlier et al. [9][1]:

The variables occurring in a literal are distinct; all clauses of a procedure have exactly the same head; if a clause uses m different program variables, these variables are $x_1, \ldots, x_m$.

P	$\in$	$Programs$	P	$::=$	$pr \mid pr\ P$
pr	$\in$	$Procedures$	pr	$::=$	$c \mid c\ pr$
c	$\in$	$Clauses$	c	$::=$	$h : -g.$
h	$\in$	$ClauseHeads$	h	$::=$	$p(x_1, \ldots, x_n)$
g	$\in$	$ClauseBodyPrefixes$	g	$::=$	$<> \mid g\ ,\ l$
l	$\in$	$Literals$	l	$::=$	$p(x_{i_1}, \ldots, x_{i_n}) \mid b$
b	$\in$	$Built-ins$	b	$::=$	$x_i = x_j \mid x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$
p	$\in$	$ProcedureNames\ (\mathcal{P})$			
f	$\in$	$Functors$			
x	$\in$	$ProgramVariables\ (PV)$			

C.2 Advantages of normalization

The advantages of normalized programs come from the fact that a(n) (input or output) substitution for a procedure p/n is always expressed in terms of variables

[1]The sole notable difference between [9] and our paper is that here a normalized program may contain distinct procedures having the same name but with a different arity.

$x_1, ..., x_n$. This greatly simplifies all of the traditional problems encountered with renaming.

C.3 A sample normalized program

Let us show you the normalized counterpart of the procedures `list/1` and `select/3` whose usual Prolog codes and their normalized versions are depicted in Figure C.1 and Figure C.2.

```
list([]).
list([_|LS]):- list(LS).
```

```
list(X1):- X1=[].
list(X1):- X1=[X2|X3], list(X3).
```

Figure C.1: The procedure `list/1` and its normalized version.

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

```
select(X1, X2, X3):- X2=[X1|X3], list(X3).
select(X1, X2, X3):- X2=[X4|X5], X3=[X4|X6], select(X1, X5, X6).
```

Figure C.2: The procedure `select/3` and its normalized version.

C.4 Algorithm of normalization

Every Prolog program can be written in a normalized form. We explicit here the rules that transform a Prolog procedure into its normalized form.

A Prolog procedure is normalized when its composing clauses are normalized.

The normalization of a Prolog clause is done by executing successively normalization processes acting on the literals that compose that clause. In this manner, the Prolog clause

```
p(t1,...,tn) :- l1, ... , lq.
```

is normalized as follows[2]:

```
p($1,...,$n) :- norm[$1=t1] , ... , norm[$n=tn] ,
                norm[l1] , ... , norm[lq].
```

[2]The normalized variables are denoted $i, in place of X_i, in order to make no confusion with user variables $X, Y, W, \ldots$

where the normalization process of a literal `li` has the effect to replace it by a sequence of normalized literals `li1,...,lin` which are semantically equivalent to `li`.

This normalization process is denoted by 'norm':

```
norm : Literals --> P(Literals)
          li        norm[li] = li1,...,lin
```

It remains now to show the different cases which occur during the normalization process of a given literal. In fact we always come down to one of these two interdependent problems (related to the two existing types of a literal):

- **Case A :** `p(t1,...,tn)`
  It corresponds to the normalization of a predicate or a functor structure.
- **Case B :** `t1 = t2`
  It corresponds to the normalization of an unification between two terms.

with `tj` $\in$ {`$i`, `X`, `f(t1,...,tn)`}
where `$i` denotes a normalized variable
`X` denotes a user variable
`f(t1,...,tn)` denotes a functor of arity n (n possibly zero)

The normalization process supposes the existence of a mapping which will update and keep in memory the correspondences between the user variables and the normalized variables replacing them in the current building of the normalized form.

There exists a mapping for each clause. It is denoted by 'map':

```
map : User Variables --> Normalized Variables
          X                  map(X) = $i
```

This mapping will allow the user to analyze the results of the static analysis (when reading the output reports), able in this way to correlate the normalized variables with the variables initially provided by the user.

We explain now how to normalize `p(t1,...,tn)` and `t1=t2`.

Case A: p(t1,...,tn)

This structure is transformed into:

```
norm[$i1=t1] , ... , norm[$in=tn] , p($i1,...,$in)
```

where:
- all `$ij` must be different from each other
- we must keep in memory the mappings `map(tj) = $ij` when `tj` is a user variable (e.g., `X`)

Note that we have come down to the **Case B** normalization problem (consider the list of successive `norm[$ij=tj]`).

As an example, let us normalize the structure `p(f(X),X,Y,X,C)`.
Suppose that the mapping is in the following current state:

```
map(X) = $1
map(Y) = $2
```

The structure `p(f(X),X,Y,X,C)` is therefore transformed into:

```
norm[$3=f(X)],norm[$1=X],norm[$2=Y],norm[$4=X],
                              norm[$5=C],p($3,$1,$2,$4,$5)
```

As you can see, the normalized variables $1,$2,$3,$4 and $5 are all distinct. You see also that we have used the existing mapping. Finally, note that in order to maintain the condition of difference between normalized variables, the second `X` has been attached to $4 and no more to $1.

The mapping has been updated as follows:

```
map(X) = $1
map(Y) = $2
map(C) = $5
```

Case B: t1 = t2

Five subcases are to be considered:

(1) `$i = t`
(2) `t = $i`
(3) `X = t`
(4) `t = X`
(5) `f(t1,...,tn) = g(w1,...,wm)`

Subcase (1): `$i = t`

We have several possibilities depending of the nature of `t`:

- `$i = $j` with `i = j`
  We can delete this item.
- `$i = $j` with `i` $\neq$ `j`
  Already normalized.
- `$i = X`
  We can come down to the case `$i = $j` where `$j = map(X)` if this mapping exists or `$j` is a new normalized variable never used before in the current clause otherwise.

- `$i = f`
  Already normalized.
- `$i = f(t1,...,tn)`
  It suffices to normalize `f(t1,...,tn)` (**Case A**) with the constraint that the normalized variables appearing in the normalized functor must be all different from each other **and** also with the supplementary constraint that these latter must be all different from `$i`. We then obtain:

  ```
  norm[$i1 = t1] , ... , norm[$in = tn] , $i = f($i1,...,$i2)
  ```

The subcases (2), (3), (4) and (5) come down to the subcase (1) as follows:

Subcase (2): `t = $i`

It comes down to normalize `$i = t` (subcase (1)).

Subcase (3): `X = t`

It comes down to normalize `$i = t` (subcase (1)) where `$i = map(X)` if this mapping exists or `$i` is a new normalized variable never used before in the current clause otherwise.

Subcase (4): `t = X`

It comes down to normalize `X = t` (subcase (3)).

Subcase (5): `f(t1,...,tn) = g(w1,...,wm)`

It is normalized into:

```
norm[$i1 = f(t1,...,tn)] , norm[$i2 = g(w1,...,wm)] , $i1 = $i2
```

where `$i1` and `$i2` are two distinct normalized variables never used before in the current clause.

Note that there is an unification error when `f`$\neq$ `g` or when `n` $\neq$ `m`, but such an error will be treated later, during the abstract execution of the literals.

Appendix D

Description of the packages

In this Appendix we report in some tables the informal description of the packages and the classes they contain.

Package	Description of the package
root package	Contains the launcher and the abstract semantics of the analyser.
myio	Contains some basic operations on input/output with files.
mystructure	Contains the structures we use to implement relations, functions, ...
parsing.prolog	Contains the parser of Prolog procedures.
parsing.beh	Contains the parser of formal specifications.
program	Contains the abstract tree representing a Prolog program.
spec	Contains the abstract tree representing a formal specification.
adom	Contains all the abstract domains and the abstract operations.

Table D.1: Description of the packages.

Class	Description of the class
Start	The launcher of the analyzer.
	It calls the different parsers and the analyser.
Analyser	Contains the abstract execution (abstract semantics) of the verification.

Table D.2: Description of the "root" package.

Class	Description of the class
MyFile	Contains some static methods used for processing with files.

Table D.3: Description of the package `myio`.

Class	Description of the class
Spec	Represents a <specification> defined in Appendix A.3 (page 95), that is a formal specification given by the user.
AbstrSubst	Represents an <abstr-subst> defined in Appendix A.3 (page 95), that is an abstract substitution given by the user.
Arg	Represents an <arg> defined in Appendix A.3 (page 95), that is an argument of a formal abstract substitution given by the user.
ArgFrame	Represents a <frame> defined in Appendix A.3 (page 95), that contains the frame (pattern) information of a user-given argument.
VariableOut	Represents a 'out' <tag-arg-name> occurring in a formal specification given by the user, in Appendix A.3 (page 95).
VariableRef	Represents a 'ref' <tag-arg-name> occurring in a formal specification given by the user, in Appendix A.3 (page 95).

Table D.4: Description of the package spec.

Interface	Description of the interface
MyObject	An object that can belong to a MySet object.
Class	**Description of the class**
MyBoolean	A boolean.
MyInteger	An integer.
Pair	A pair of two MyObject objects "(x,y)".
MySet	A(n) (ordered) set of MyObject, with no duplications.
MyRelation	A mathematical concept of relation.
MyFunction	A mathematical concept of function.
MySymRelation	A mathematical concept of symmetrical relation.
I2I	A function from Indices to Indices.
V2V	A function from Variables to Variables.
I2Frame	A function from Indices to Frames.
I2Mode	A function from Indices to Modes.
I2Type	A function from Indices to Types.
I2Variable	A function from Indices to Variables.
Variable2I	A function from Variables to Indices.
IxI2I	A function from (Indices X Indices) to Indices.
IxI	A symmetrical relation into (Indices X Indices).
VxV	A symmetrical relation into (Variables X Variables).

Table D.5: Description of the package mystructure.

Interface	**Description of the interface**
Literal	Represents a `<literal>` defined in Appendix A.2 (page 94).
Term	Represents a `<term>` defined in Appendix A.2 (page 94).
Class	**Description of the class**
Program	The abstract tree representing internally a Prolog program in its normalized form or not. It contains a set of Procedure objects.
Procedure	Represents a Prolog procedure. It contains its defined clauses (i.e., an ordered sequence of Clause objects).
Clause	Represents a `<clause>` defined in Appendix A.2 (page 94).
Variable	Represents a `<variable>` defined in Appendix A.2 (page 94).
PredicateFunctor	Represents either a Predicate or a Functor.
Functor	Represents a `<functor>` defined in Appendix A.2 (page 94).
Predicate	Represents a `<predicate>` defined in Appendix A.2 (page 94).
Unification	Represents a `<unification>` defined in Appendix A.2 (page 94).
EmptyLiteral	Represents the empty literal (i.e., that is always true). It is used for normalization purpose in Appendix C.4 (page 108).
Mapping	Keeps information about correspondences between the terms defined by the user and the new fresh normalized variables when constructing the normalization form of a Clause.

Table D.6: Description of the package `program`.

Interface	Description of the interface
Exp	Represents an <expression> and a <tag-expression> defined in Appendix A.3 (page 95).
Class	**Description of the class**
Indice	Represents an indice.
Frame	Represents a frame of indices defined in Section 4.1 (page 19).
SV	Represents the domain SV, that is the same-value component of an Abstract Substitution, defined in Section 4.2.1 (page 21).
FRM	Represents the domain FRM, that is the frame-value (or pattern) component of an Abstract Substitution, defined in Section 4.2.2 (page 21).
Mode	Represents the domain Modes, with the related abstract operations, defined in Section 4.2.3 (page 22).
ModesI	Represents the domain ModesI, that is the mode component of an Abstract Substitution, defined in Section 4.2.4 (page 23).
Type	Represents the domain Types, with the related abstract operations, defined in Section 4.2.5 (page 23).
TypesI	Represents the domain TypesI, that is the type component of an Abstract Substitution, defined in Section 4.2.6 (page 24).
PSharingI	Represents the domain PSharingI, that is the possible sharing component of an Abstract Substitution, defined in Section 4.2.7 (page 24).
ASubst	Represents the domain of Abstract Substitutions, defined in Section 4.2.9 (page 26). It contains the related abstract operations.
Sizes	Represents the domain SizesI, defined in Section 4.3.1 (page 28).
Sol	Represents the terminal <sol>.
BinaryExp	Represents a binary expression, defined in Appendix A.3 (page 95).
Constant	Represents a <natural> defined in Appendix A.3 (page 95).
Delta	Represents the δ-tuple defined in Section 7.7.2 (page 59). It is the structure used when implementing the general unification between two terms (UNIF).
EqInEq	Represents an <inequality> or a <sol-inequality>, defined in Appendix A.3 (page 95).
ASeq	Represents the domain of Abstract Sequences, defined in Section 4.3.2 (page 29). It contains the related abstract operations.
BehaviouralPair	Represents a behavioural pair of a behaviour, defined in Section 4.4 (page 31).
Behaviour	Represents a behaviour of a procedure and contains its defined behavioural pairs, defined in Section 4.4 (page 31).
SBeh	The abstract tree representing internally a family of behaviours.

Table D.7: Description of the package adom.

Now follow some UML-like diagrams where we report the organization of the different packages. Figure D.1 shows the representation conventions we adopted.

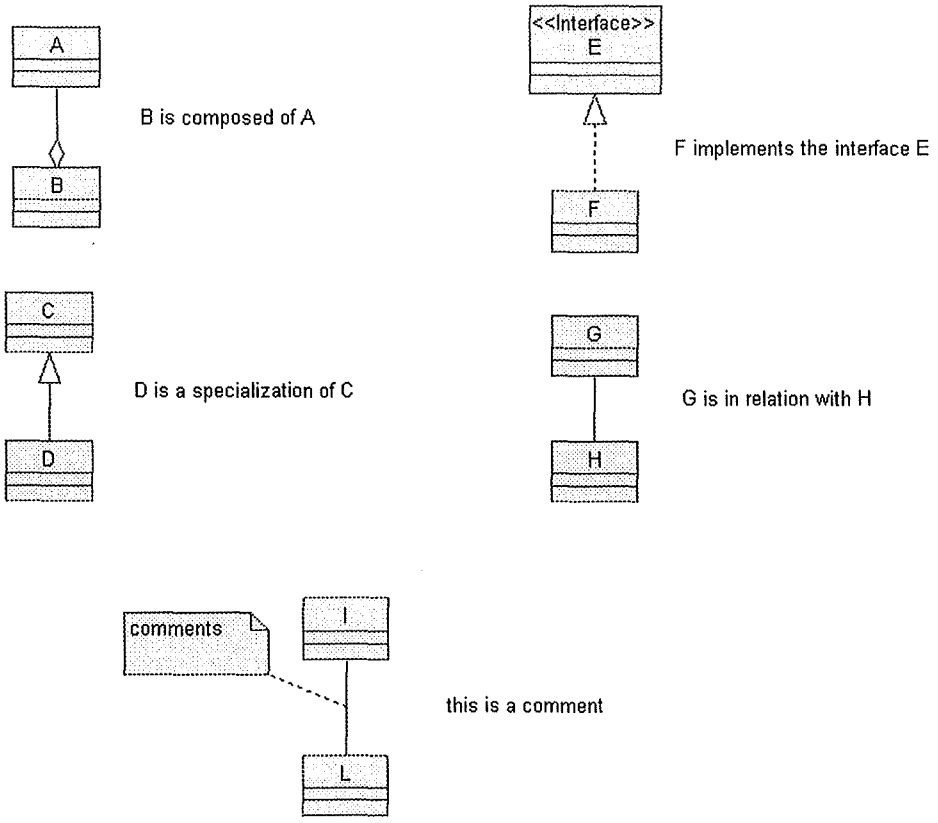


Figure D.1: UML conventions.

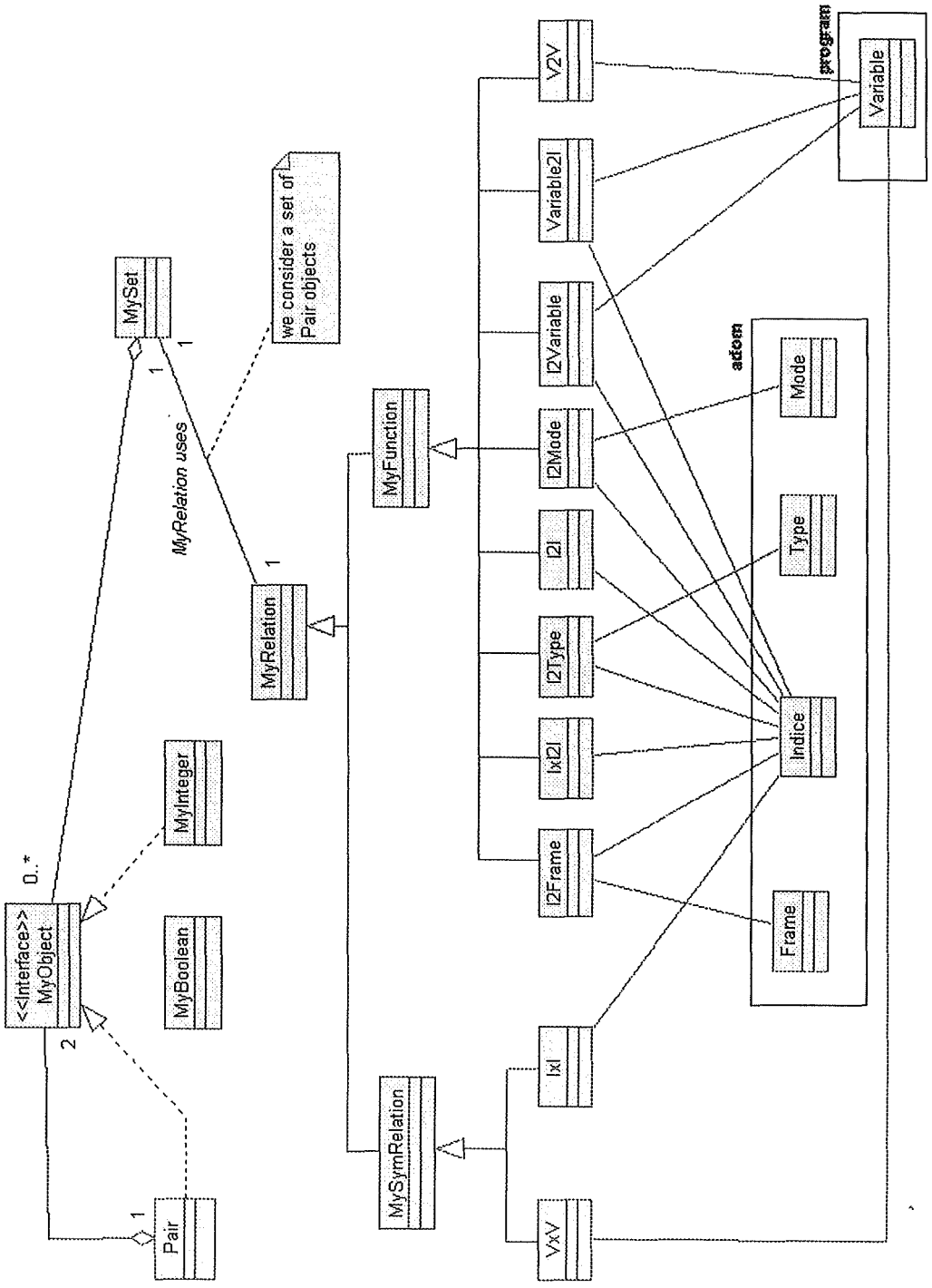


Figure D.2: Class diagram of the package `mystructure`.

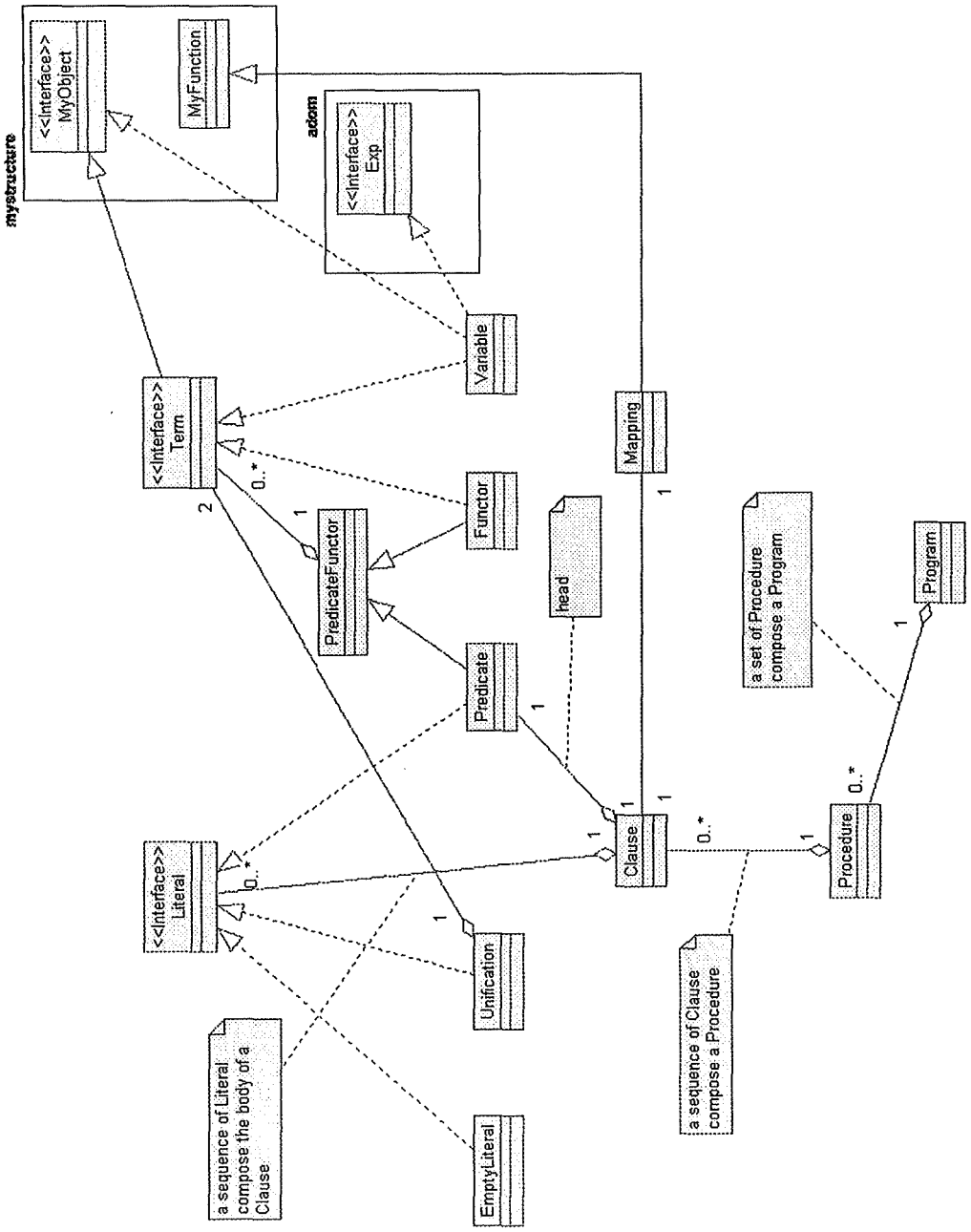


Figure D.3: Class diagram of the package `program`.

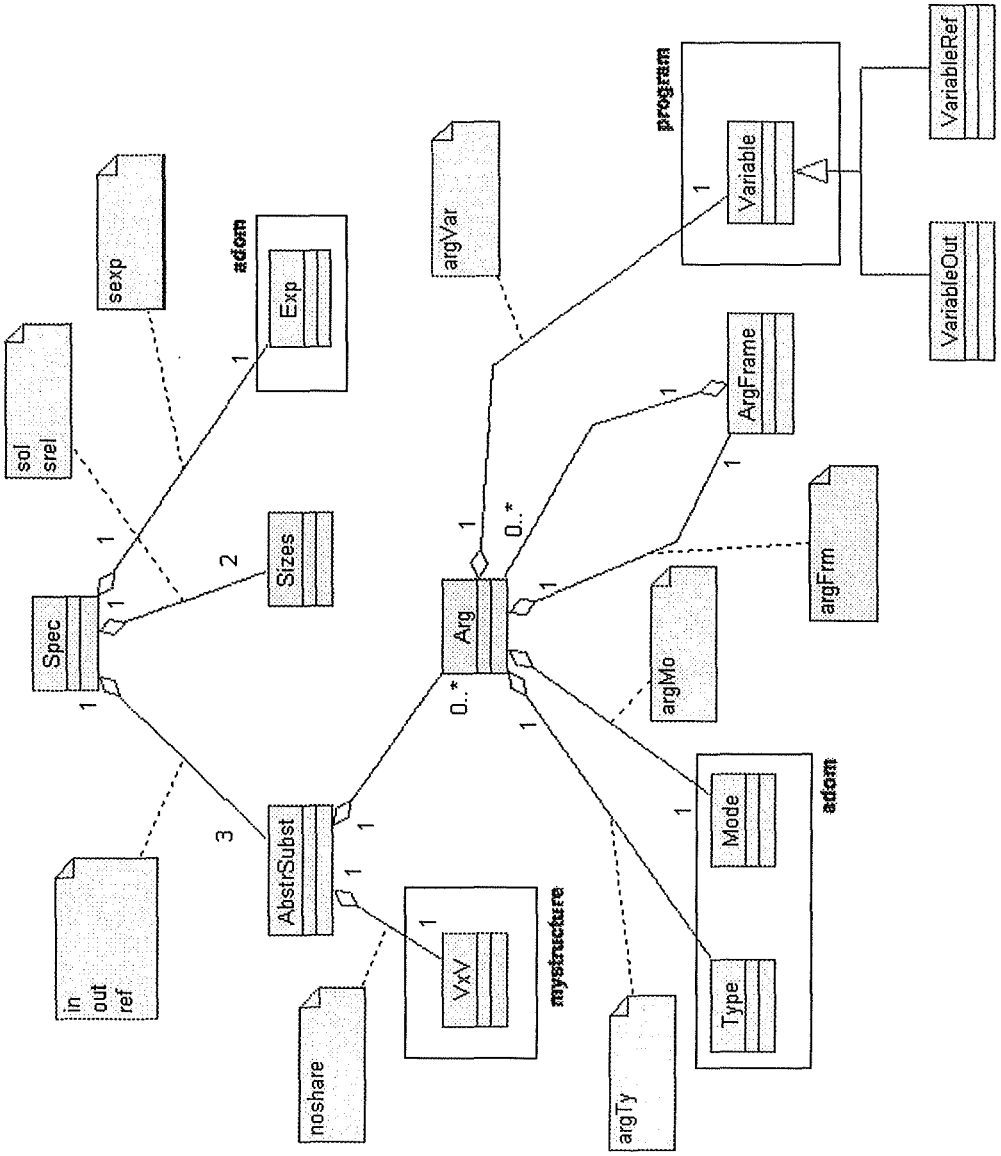


Figure D.4: Class diagram of the package spec.

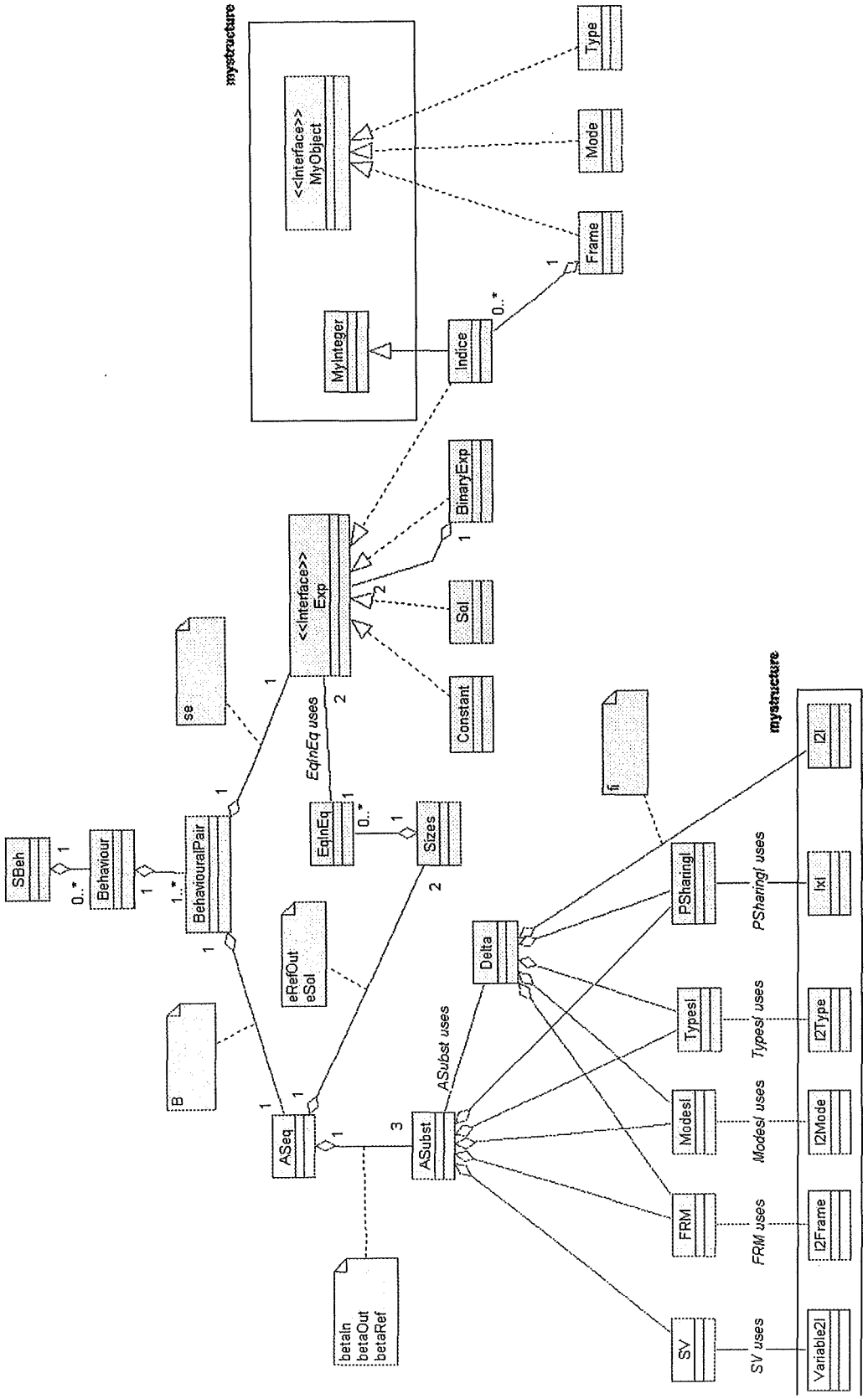


Figure D.5: Class diagram of the package adom.

Appendix E

The algorithm of the analyser

In this Appendix we put the implementation of the three main procedures of our analyser, namely `analyze_program`, `analyze_procedure` and `analyze_clause`.

```
PROCEDURE analyze_program(P, SBeh) =
      success ← true
      for all p ∈ 𝒫, for all ⟨B, se⟩ ∈ Beh_p
         success ← success ∧ analyze_procedure(p, B, se, SBeh)
      return success.
```

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

PROCEDURE analyze_clause$(c, B, se, SBeh)$ =
  $\beta_{in} \leftarrow input(B)$
  $B_0 \leftarrow$ EXTC(c, β_{in})
  for $k \leftarrow 1$ **to** s **do**
    $\beta^k_{inter} \leftarrow$ RESTRG(l_k, B_{k-1})
    switch (l_k) **do**
      case $X_{i_1} = X_{i_2}$:
        $B^k_{aux} \leftarrow$ UNIF_VAR(β^k_{inter})
      case $X_{i_1} = f(X_{i_2}, \ldots, X_{i_m})$:
        $B^k_{aux} \leftarrow$ UNIF_FUNC(β^k_{inter}, f)
      case $q(X_{i_1}, \ldots, X_{i_m})$ **and** $q \neq p$:
        $\langle success_k, B^k_{aux} \rangle \leftarrow$ LOOKUP$(\beta^k_{inter}, q, SBeh)$
      case $p(X_{i_1}, \ldots, X_{i_n})$:
        $B^k_{aux} \leftarrow B$
        $success_k \leftarrow$ CHECK_TERM(l_k, B_{k-1}, se)
    $B_k \leftarrow$ EXTG$(l_k, B_{k-1}, B^k_{aux})$
  if there exists k **such that**
    either $l_k \equiv q(X_{i_1}, \ldots, X_{i_m}) \wedge \neg success_k$
    or $l_k \equiv p(X_{i_1}, \ldots, X_{i_n}) \wedge (\neg success_k \vee \beta^k_{inter} \not\leq \beta_{in})$
    then $success \leftarrow false$
    else $success \leftarrow true$ **and** $B_{out} =$ RESTRC(c, B_s)
  return $\langle success, B_{out} \rangle$

The analyser follows the standard top-down verification technique: for a given program, it analyzes each procedure; for a given procedure, it analyzes each clause; for a given clause, it analyzes 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.

Note that to make the code more readable, we have assumed that the algorithm stops and fails ($success = false$) if one of the sub-operations returns $\perp$.

The operation CONC$(B_1, \ldots, B_r)$ is a shortcut for CONC$(\ldots$CONC$(B_{r-1}, B_r)\ldots)$.

The analysis of a clause $\mathtt{c} \equiv \mathtt{p}(\mathtt{X_1}, \ldots, \mathtt{X_n}) : -\mathtt{l_1}, \ldots, \mathtt{l_s}.$ with respect to $\langle B, se \rangle \in Beh_p$ consists in the following steps:

1. extending the input substitution β_{in} of B to an abstract sequence B_0 on all the variables in the clause through the operation EXTC;
2. computing B_k from B_{k-1} and l_k ($l_k \in \{1, \ldots, s\}$);
3. restricting B_s to the variables in the head of c through the operation RESTRC.

Each B_k is computed from B_{k-1} and l_k by:

1. restricting the domain of the output abstract substitution β_{out} of B_{k-1} to the variables $X_{i_1}, \ldots, X_{i_n}$ of l_k and renaming them into $X_1, \ldots, X_n$ through the operation RESTRG;
2. executing the literal l_k with β^k_{inter} which returns an abstract sequence B^k_{aux};
3. propagating this result on B_{k-1} by computing $B_k = \texttt{EXTG}(l_k, B_{k-1}, B^k_{aux})$.

The execution of l_k with β^k_{inter} depends on the form of l_k:

1. If l_k is a built-in of the form $X_{i_1} = X_{i_2}$ then $B^k_{aux} = \texttt{UNIF\_VAR}(\beta^k_{inter})$.
2. If l_k is of the form $X_{i_1} = f(X_{i_2}, \ldots, X_{i_n})$ then $B^k_{aux} = \texttt{UNIF\_FUNC}(\beta^k_{inter}, f)$.
3. If l_k is a non-recursive call $q(X_{i_1}, \ldots, X_{i_m})$ (i.e., $q \neq p$) then the analyzer looks at $SBeh$, the set of behaviours, to find an abstract sequence general enough to give information about this call.
4. If l_k is a recursive call $p(X_{i_1}, \ldots, X_{i_n})$ then the analyzer checks whether the size the arguments decreases through the operation CHECK_TERM(l_k, B_{k-1}, se).

Appendix F

Output reports

F.1 Structure of an output report

Output reports are the files generated by our analyser. There is one output report for each pair of analyzed procedure/specification. The name of the files have the following format:

```
<procedure-name> "." <arity> "." <specification-number>
```

The content of such a file is a trace consisting of the different steps of the analysis. Let us show you the schema (the different parts) of an output report:

```
**********************************
*     PROCEDURE IDENTIFICATION     *
**********************************

 - Name:
      The name of the analyzed procedure.
 - Arity:
      The arity of the analyzed procedure.
 - Clauses:
      The enumeration of the procedure clauses.
      Each clause is given in its un-normalized form and in its
      normalized form. Further more, a mapping keeping the
      correspondences between user-given named arguments and
      normalized variables is shown. Note that the normalized
      variables have the format "$<natural>".

**********************************
*     BEHAVIOUR IDENTIFICATION     *
**********************************
```

```
 - Formal Specification:
      The formal specification (given by the user) currently
      analyzed.
 - Behaviour:
      The behavioural pair related to the formal specification.
      It consists of the normalized abstract sequence part
      (noted "B") and of the size expression part (noted "se").

************************************
*              ANALYSIS            *
************************************

 The effective analysis starts here.

 Each clause is analyzed step by step, according to the rules
 of the abstract execution of a clause:
    (1) execution of EXTC
    (2) for each each literal of the clause:
           (a) RESTRG
           (b) either UNIF_VAR, UNIF_FUNC or LOOKUP
           (c) EXTG
    (3) execution of RESTRC

 When a failure occurs (e.g., a sub-operation returns bottom
 or a condition is not satisfied), the analysis stops at that
 point (does not continue). Further more, the global analysis
 of the whole program stops, returning "success=false".

 Otherwise - if no error occurs previously - the analyser
 calculates the abstract concatenation of the abstract
 sequences computed for each clause (operation CONC):

  *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
  ABSTRACT CONCATENATION OF THE CLAUSES: CONC(Blist) = B_out
  *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-

   The result is then one of these two, according to we
   obtain a successful analysis:

   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
   ==> We have "B_out <= B" ==> we can infer the verification.
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-

   or an unsuccessful analysis:

   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
```

```
   ==> We have not "B_out <= B" ==> we cannot infer
                                                the verification.
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
```

As already mentioned in Section Coding, the operations related to the size components based on the polyhedron library have not been implemented yet, since the work is still under progress. It is the reason you will then find in some analyzed points of the report the label `/* not implemented yet */`; you will note also that the operation `CHECK_TERM` (checking if a size expression strictly decreases before executing a recursive call) returns always *true*, because it is not implemented.

F.2 A successful analysis

In this Section, we provide the output report of the analysis of the program `select/3` according to the formal specifications of `select/3` and `list/1` given in Section 2.3.

```
 - Current file: "mySelect.3.1"

**********************************
*     PROCEDURE IDENTIFICATION     *
**********************************

 - Name:  mySelect
 - Arity: 3

 - Clauses:
    (1) -> UnNormalized:
              mySelect(X,[X|T],T) :- list(T).
        -> Normalized:
              mySelect($1,$2,$3) :- $2=[$1|$3],list($3).
        -> Mapping:
              {$1->X,$2->[X|T],$3->T}

    (2) -> UnNormalized:
              mySelect(X,[H|T],[H|TS]) :- mySelect(X,T,TS).
        -> Normalized:
              mySelect($1,$2,$3) :- $2=[$4|$5],$3=[$4|$6],mySelect($1,$5,$6).
        -> Mapping:
              {$1->X,$2->[H|T],$3->[H|TS],$4->H,$5->T,$6->TS}


**********************************
*     BEHAVIOUR IDENTIFICATION     *
**********************************

 - Formal Specification:
```

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

 - Behaviour:
      ================mySelect/3================
      beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
               mo = {1->var,2->ground,3->var}
               ty = {1->anylist,2->any,3->anylist}
               ps = {(1,1),(3,3)}
      beta_ref: sv = {$1->1,$2->2,$3->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 = {$1->1,$2->2,$3->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 = {}
      E_ref_out = /* not implemented yet */
      E_sol = /* not implemented yet */
      ==========================================
      se = /* not implemented yet */

***********************************
*              ANALYSIS           *
***********************************

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
==>  Clause (1) to analyze:
         c = mySelect($1,$2,$3) :- $2=[$1|$3],list($3).

  ------------------------------------
  EXTC(c,betaIn) = B_0
  ------------------------------------

      ================B_0================
      beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
               mo = {1->var,2->ground,3->var}
               ty = {1->anylist,2->any,3->anylist}
               ps = {(1,1),(3,3)}
      beta_ref: sv = {$1->1,$2->2,$3->3}; frm = {}
                mo = {1->var,2->ground,3->var}
                ty = {1->anylist,2->any,3->anylist}
                ps = {(1,1),(3,3)}
```

```
    beta_out: sv = {$1->1,$2->2,$3->3}; frm = {}
              mo = {1->var,2->ground,3->var}
              ty = {1->anylist,2->any,3->anylist}
              ps = {(1,1),(3,3)}
    E_ref_out = /* not implemented yet */
    E_sol = /* not implemented yet */
    ==================================

------------------------------------
Literal to abstractly execute: l1 = $2=[$1|$3]
------------------------------------

    ------------------------------------
    RESTRG(l1,B_0) = beta^1_inter
    ------------------------------------
        beta^1_inter: sv = {$1->2,$2->1,$3->3}; frm = {}
                      mo = {1->var,2->ground,3->var}
                      ty = {1->anylist,2->any,3->anylist}
                      ps = {(1,1),(3,3)}

    ------------------------------------
    UNIF_FUNC(beta^1_inter,[$1|$3]) = B^1_aux
    ------------------------------------
        ================B^1_aux================
        beta_in: sv = {$1->2,$2->1,$3->3}; frm = {}
                 mo = {1->var,2->ground,3->var}
                 ty = {1->anylist,2->any,3->anylist}
                 ps = {(1,1),(3,3)}
        beta_ref: sv = {$1->2,$2->1,$3->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 = {$1->2,$2->1,$3->3}; frm = {2->[1|3]}
                  mo = {1->ground,2->ground,3->ground}
                  ty = {1->any,2->any,3->any}
                  ps = {}
        E_ref_out = /* not implemented yet */
        E_sol = /* not implemented yet */
        ======================================

    ------------------------------------
    EXTG(l1,B_0,B^1_aux) = B_1
    ------------------------------------
        ================B_1================
        beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
                 mo = {1->var,2->ground,3->var}
                 ty = {1->anylist,2->any,3->anylist}
                 ps = {(1,1),(3,3)}
        beta_ref: sv = {$1->1,$2->2,$3->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 = {$1->1,$2->2,$3->3}; frm = {2->[1|3]}
                mo = {1->ground,2->ground,3->ground}
                ty = {1->any,2->any,3->any}
                ps = {}
        E_ref_out = /* not implemented yet */
        E_sol = /* not implemented yet */
        ==================================

-------------------------------------
Literal to abstractly execute: l2 = list($3)
-------------------------------------

    -------------------------------------
    RESTRG(l2,B_1) = beta^2_inter
    -------------------------------------
        beta^2_inter: sv = {$1->1}; frm = {}
                      mo = {1->ground}
                      ty = {1->any}
                      ps = {}

    -------------------------------------
    LOOKUP(beta^2_inter,list($3),sbeh) = B^2_aux
    -------------------------------------
        ================B^2_aux================
        beta_in: sv = {$1->1}; frm = {}
                 mo = {1->ground}
                 ty = {1->any}
                 ps = {}
        beta_ref: sv = {$1->1}; frm = {}
                  mo = {1->ground}
                  ty = {1->list}
                  ps = {}
        beta_out: sv = {$1->1}; frm = {}
                  mo = {1->ground}
                  ty = {1->list}
                  ps = {}
        E_ref_out = /* not implemented yet */
        E_sol = /* not implemented yet */
        ======================================

    -------------------------------------
    EXTG(l2,B_1,B^2_aux) = B_2
    -------------------------------------
        ================B_2================
        beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
                 mo = {1->var,2->ground,3->var}
                 ty = {1->anylist,2->any,3->anylist}
                 ps = {(1,1),(3,3)}
```

```
          beta_ref: sv = {$1->1,$2->2,$3->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 = {$1->1,$2->2,$3->3}; frm = {2->[1|3]}
                    mo = {1->ground,2->ground,3->ground}
                    ty = {1->any,2->list,3->list}
                    ps = {}
          E_ref_out = /* not implemented yet */
          E_sol = /* not implemented yet */
          ===================================

  -------------------------------------
  RESTRC(c,B_2) = B_out
  -------------------------------------

      ================B_out================
      beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
               mo = {1->var,2->ground,3->var}
               ty = {1->anylist,2->any,3->anylist}
               ps = {(1,1),(3,3)}
      beta_ref: sv = {$1->1,$2->2,$3->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 = {$1->1,$2->2,$3->3}; frm = {2->[1|3]}
                mo = {1->ground,2->ground,3->ground}
                ty = {1->any,2->list,3->list}
                ps = {}
      E_ref_out = /* not implemented yet */
      E_sol = /* not implemented yet */
      ====================================
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
==>  Clause (2) to analyze:
         c = mySelect($1,$2,$3) :- $2=[$4|$5],$3=[$4|$6],mySelect($1,$5,$6).

  -------------------------------------
  EXTC(c,betaIn) = B_0
  -------------------------------------

      ================B_0================
      beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
               mo = {1->var,2->ground,3->var}
               ty = {1->anylist,2->any,3->anylist}
               ps = {(1,1),(3,3)}
      beta_ref: sv = {$1->1,$2->2,$3->3}; frm = {}
                mo = {1->var,2->ground,3->var}
                ty = {1->anylist,2->any,3->anylist}
```

```
            ps = {(1,1),(3,3)}
    beta_out: sv = {$1->1,$2->2,$3->3,$4->4,$5->5,$6->6}; frm = {}
            mo = {1->var,2->ground,3->var,4->var,5->var,6->var}
            ty = {1->anylist,2->any,3->anylist,4->anylist,5->anylist,
                  6->anylist}
            ps = {(1,1),(3,3),(4,4),(5,5),(6,6)}
    E_ref_out = /* not implemented yet */
    E_sol = /* not implemented yet */
    ==================================

-------------------------------------
Literal to abstractly execute: l1 = $2=[$4|$5]
-------------------------------------

    -------------------------------------
    RESTRG(l1,B_0) = beta^1_inter
    -------------------------------------
        beta^1_inter: sv = {$1->1,$2->2,$3->3}; frm = {}
                      mo = {1->ground,2->var,3->var}
                      ty = {1->any,2->anylist,3->anylist}
                      ps = {(2,2),(3,3)}

    -------------------------------------
    UNIF_FUNC(beta^1_inter,[$4|$5]) = B^1_aux
    -------------------------------------
        ================B^1_aux================
        beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
                 mo = {1->ground,2->var,3->var}
                 ty = {1->any,2->anylist,3->anylist}
                 ps = {(2,2),(3,3)}
        beta_ref: sv = {$1->1,$2->2,$3->3}; frm = {1->[4|5]}
                  mo = {1->ground,2->var,3->var,4->ground,5->ground}
                  ty = {1->any,2->anylist,3->anylist,4->any,5->any}
                  ps = {(2,2),(3,3)}
        beta_out: sv = {$1->1,$2->2,$3->3}; frm = {1->[2|3]}
                  mo = {1->ground,2->ground,3->ground}
                  ty = {1->any,2->any,3->any}
                  ps = {}
        E_ref_out = /* not implemented yet */
        E_sol = /* not implemented yet */
        ======================================

    -------------------------------------
    EXTG(l1,B_0,B^1_aux) = B_1
    -------------------------------------
        ================B_1================
        beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
                 mo = {1->var,2->ground,3->var}
                 ty = {1->anylist,2->any,3->anylist}
                 ps = {(1,1),(3,3)}
```

```
        beta_ref: sv = {$1->1,$2->2,$3->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 = {$1->1,$2->2,$3->3,$4->4,$5->5,$6->6};
                  frm = {2->[4|5]}
                  mo = {1->var,2->ground,3->var,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 = /* not implemented yet */
        E_sol = /* not implemented yet */
        ==================================

-------------------------------------
Literal to abstractly execute: 12 = $3=[$4|$6]
-------------------------------------

    -------------------------------------
    RESTRG(12,B_1) = beta^2_inter
    -------------------------------------
        beta^2_inter: sv = {$1->1,$2->2,$3->3}; frm = {}
                      mo = {1->var,2->ground,3->var}
                      ty = {1->anylist,2->any,3->anylist}
                      ps = {(1,1),(3,3)}

    -------------------------------------
    UNIF_FUNC(beta^2_inter,[$4|$6]) = B^2_aux
    -------------------------------------
        ================B^2_aux================
        beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
                 mo = {1->var,2->ground,3->var}
                 ty = {1->anylist,2->any,3->anylist}
                 ps = {(1,1),(3,3)}
        beta_ref: sv = {$1->1,$2->2,$3->3}; frm = {}
                  mo = {1->var,2->ground,3->var}
                  ty = {1->anylist,2->any,3->anylist}
                  ps = {(1,1),(3,3)}
        beta_out: sv = {$1->1,$2->2,$3->3}; frm = {1->[2|3]}
                  mo = {1->ngv,2->ground,3->var}
                  ty = {1->anylist,2->any,3->anylist}
                  ps = {(3,3)}
        E_ref_out = /* not implemented yet */
        E_sol = /* not implemented yet */
        ====================================

    -------------------------------------
    EXTG(12,B_1,B^2_aux) = B_2
    -------------------------------------
```

```
        ================B_2================
        beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
                 mo = {1->var,2->ground,3->var}
                 ty = {1->anylist,2->any,3->anylist}
                 ps = {(1,1),(3,3)}
        beta_ref: sv = {$1->1,$2->2,$3->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 = {$1->1,$2->2,$3->3,$4->4,$5->5,$6->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),(6,6)}
        E_ref_out = /* not implemented yet */
        E_sol = /* not implemented yet */
        ==================================

-------------------------------------
Literal to abstractly execute: l3 = mySelect($1,$5,$6)
-------------------------------------

    -------------------------------------
    RESTRG(l3,B_2) = beta^3_inter
    -------------------------------------
        beta^3_inter: sv = {$1->1,$2->2,$3->3}; frm = {}
                      mo = {1->var,2->ground,3->var}
                      ty = {1->anylist,2->any,3->anylist}
                      ps = {(1,1),(3,3)}

    -------------------------------------
    CHECK_TERM(l3,B3,se) = true
    -------------------------------------

    -------------------------------------
    LEQ(beta^3_inter,betaIn) = true
    -------------------------------------

    -------------------------------------
    B^3_aux = B
    -------------------------------------
        ================B^3_aux================
        beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
                 mo = {1->var,2->ground,3->var}
                 ty = {1->anylist,2->any,3->anylist}
                 ps = {(1,1),(3,3)}
        beta_ref: sv = {$1->1,$2->2,$3->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 = {$1->1,$2->2,$3->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 = {}
        E_ref_out = /* not implemented yet */
        E_sol = /* not implemented yet */
        ====================================

    ------------------------------------
    EXTG(13,B_2,B^3_aux) = B_3
    ------------------------------------
        ================B_3================
        beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
                mo = {1->var,2->ground,3->var}
                ty = {1->anylist,2->any,3->anylist}
                ps = {(1,1),(3,3)}
        beta_ref: sv = {$1->1,$2->2,$3->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 = {$1->1,$2->2,$3->3,$4->4,$5->5,$6->6};
                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 = {}
        E_ref_out = /* not implemented yet */
        E_sol = /* not implemented yet */
        ====================================

------------------------------------
RESTRC(c,B_3) = B_out
------------------------------------

    ================B_out================
    beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
            mo = {1->var,2->ground,3->var}
            ty = {1->anylist,2->any,3->anylist}
            ps = {(1,1),(3,3)}
    beta_ref: sv = {$1->1,$2->2,$3->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 = {$1->1,$2->2,$3->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 = {}
    E_ref_out = /* not implemented yet */
    E_sol = /* not implemented yet */
    =====================================
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>


*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
ABSTRACT CONCATENATION OF THE CLAUSES: CONC(Blist) = B_out
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-

  ================B_out================
  beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
           mo = {1->var,2->ground,3->var}
           ty = {1->anylist,2->any,3->anylist}
           ps = {(1,1),(3,3)}
  beta_ref: sv = {$1->1,$2->2,$3->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 = {$1->1,$2->2,$3->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 = {}
  E_ref_out = /* not implemented yet */
  E_sol = /* not implemented yet */
  =====================================

*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
==> We have "B_out <= B" ==> we can infer the verification.
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
```

F.3 An unsuccessful analysis

Consider the motivating sample of the Introduction. It concerns the declaratively correct but operationally incorrect procedure *delete*$(X, L, Ldel)$:

```
delete(X,L,Ldel) :- L=[H|T],not(X=H),Ldel=[H|Tdel],
                    delete_1(X,T,Tdel).
delete(X,L,Ldel) :- L=[H|T],X=H,Ldel=T,list(T).
```

with the following formal specification:

```
% Specification of "delete/3"
delete(in(X:any,L:ground,Ldel:any),
       ref(_,[_|list],anylist),
       out(ground,_,ground list),
       srel({L_ref = Ldel_out + 1}),
       sol({0 <= sol,sol <= L_ref}),
       sexpr(L))
```

Because the *not* predicate does not belong to the Pure Prolog subset – our parser cannot presently accept that lexical construction –, we temporarily replace $not(X = H)$ by a call to $not(X, H)$ where the formal specification is given here:

```
% Specification of "not/2"
not(in(X:ground,Y:ground),
    ref(_,_),
    out(_,_),
    srel(),
    sol(0 <= sol,sol <= 1),
    sexpr())
```

where we have specified in this way the fact that we allow calls to the *not* built-in only if its arguments are both ground.

At the reading of the following output report, we see that our analyser detects correctly the malposition of the *not* built-in.

```
 - Current file: "delete.3.1"

**********************************
*     PROCEDURE IDENTIFICATION     *
**********************************

 - Name:  delete
 - Arity: 3

 - Clauses:
    (1) -> UnNormalized:
```

```
            delete(X,L,Ldel) :- L=[H|T],not(H,X),Ldel=[H|Tdel],
                                delete(X,T,Tdel).
       -> Normalized:
            delete($1,$2,$3) :- $2=[$4|$5],not($4,$1),$3=[$4|$6],
                                delete($1,$5,$6).
       -> Mapping:
            {$1->X,$2->L,$3->Ldel,$4->H,$5->T,$6->Tdel}

   (2) -> UnNormalized:
            delete(X,L,Ldel) :- L=[H|T],H=X,Ldel=T,list(T).
       -> Normalized:
            delete($1,$2,$3) :- $2=[$4|$5],$4=$1,$3=$5,list($5).
       -> Mapping:
            {$1->X,$2->L,$3->Ldel,$4->H,$5->T}


**********************************
*     BEHAVIOUR IDENTIFICATION     *
**********************************

 - Formal Specification:
      delete
      (
        in(X:any,L:ground,Ldel:any),
        ref(_,[_|list],anylist),
        out(ground,_,ground list),
        srel({L_ref = Ldel_out + 1}),
        sol({0 <= sol,sol <= L_ref}),
        sexpr(L)
      )

 - Behaviour:
      ================delete/3================
      beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
               mo = {1->any,2->ground,3->any}
               ty = {1->any,2->any,3->any}
               ps = {(1,1),(1,3),(3,3)}
      beta_ref: sv = {$1->1,$2->2,$3->3}; frm = {2->[4|5]}
                mo = {1->any,2->ground,3->any,4->ground,5->ground}
                ty = {1->any,2->list,3->anylist,4->any,5->list}
                ps = {(1,1),(1,3),(3,3)}
      beta_out: sv = {$1->1,$2->2,$3->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 = {}
      E_ref_out = /* not implemented yet */
      E_sol = /* not implemented yet */
      ========================================
      se = /* not implemented yet */
```

```
************************************
*              ANALYSIS            *
************************************

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
==>  Clause (1) to analyze:
        c = delete($1,$2,$3) :- $2=[$4|$5],not($4,$1),$3=[$4|$6],
                                delete($1,$5,$6).

  ------------------------------------
  EXTC(c,betaIn) = B_0
  ------------------------------------

      ================B_0================
      beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
               mo = {1->any,2->ground,3->any}
               ty = {1->any,2->any,3->any}
               ps = {(1,1),(1,3),(3,3)}
      beta_ref: sv = {$1->1,$2->2,$3->3}; frm = {}
                mo = {1->any,2->ground,3->any}
                ty = {1->any,2->any,3->any}
                ps = {(1,1),(1,3),(3,3)}
      beta_out: sv = {$1->1,$2->2,$3->3,$4->4,$5->5,$6->6}; frm = {}
                mo = {1->any,2->ground,3->any,4->var,5->var,6->var}
                ty = {1->any,2->any,3->any,4->anylist,5->anylist,
                      6->anylist}
                ps = {(1,1),(1,3),(3,3),(4,4),(5,5),(6,6)}
      E_ref_out = /* not implemented yet */
      E_sol = /* not implemented yet */
      ==================================

  ------------------------------------
  Literal to abstractly execute: l1 = $2=[$4|$5]
  ------------------------------------

      ------------------------------------
      RESTRG(l1,B_0) = beta^1_inter
      ------------------------------------
          beta^1_inter: sv = {$1->1,$2->2,$3->3}; frm = {}
                        mo = {1->ground,2->var,3->var}
                        ty = {1->any,2->anylist,3->anylist}
                        ps = {(2,2),(3,3)}

      ------------------------------------
      UNIF_FUNC(beta^1_inter,[$4|$5]) = B^1_aux
      ------------------------------------
          ================B^1_aux================
          beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
                   mo = {1->ground,2->var,3->var}
```

```
                ty = {1->any,2->anylist,3->anylist}
                ps = {(2,2),(3,3)}
        beta_ref: sv = {$1->1,$2->2,$3->3}; frm = {1->[4|5]}
                  mo = {1->ground,2->var,3->var,4->ground,
                        5->ground}
                  ty = {1->any,2->anylist,3->anylist,4->any,5->any}
                  ps = {(2,2),(3,3)}
        beta_out: sv = {$1->1,$2->2,$3->3}; frm = {1->[2|3]}
                  mo = {1->ground,2->ground,3->ground}
                  ty = {1->any,2->any,3->any}
                  ps = {}
        E_ref_out = /* not implemented yet */
        E_sol = /* not implemented yet */
        ======================================

    --------------------------------------
    EXTG(l1,B_0,B^1_aux) = B_1
    --------------------------------------
        ================B_1================
        beta_in: sv = {$1->1,$2->2,$3->3}; frm = {}
                 mo = {1->any,2->ground,3->any}
                 ty = {1->any,2->any,3->any}
                 ps = {(1,1),(1,3),(3,3)}
        beta_ref: sv = {$1->1,$2->2,$3->3}; frm = {2->[4|5]}
                  mo = {1->any,2->ground,3->any,4->ground,
                        5->ground}
                  ty = {1->any,2->any,3->any,4->any,5->any}
                  ps = {(1,1),(1,3),(3,3)}
        beta_out: sv = {$1->1,$2->2,$3->3,$4->4,$5->5,$6->6};
                  frm = {2->[4|5]}
                  mo = {1->any,2->ground,3->any,4->ground,5->ground,
                        6->var}
                  ty = {1->any,2->any,3->any,4->any,5->any,
                        6->anylist}
                  ps = {(1,1),(1,3),(3,3),(6,6)}
        E_ref_out = /* not implemented yet */
        E_sol = /* not implemented yet */
        ==================================

------------------------------------
Literal to abstractly execute: l2 = not($4,$1)
------------------------------------

    ------------------------------------
    RESTRG(l2,B_1) = beta^2_inter
    ------------------------------------
        beta^2_inter: sv = {$1->2,$2->1}; frm = {}
                      mo = {1->any,2->ground}
                      ty = {1->any,2->any}
                      ps = {(1,1)}
```

```
      ------------------------------------
      LOOKUP(beta^2_inter,not($4,$1),sbeh) = B^2_aux
      ------------------------------------
          B^2_aux = null =====> FAILURE!!!


!!!!!!!!!!!!!!!!!!!!!!!!

 ==> The analysis of the clause
            "delete($1,$2,$3) :- $2=[$4|$5],not($4,$1),$3=[$4|$6],
                                 delete($1,$5,$6)."
     does not succeed...

!!!!!!!!!!!!!!!!!!!!!!!!
```

Bibliography

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

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

[3] L. Sterling and E. Shapiro. *The Art of Prolog, Advanced Programming Techniques.* Second Edition. The MIT Press. 1994. Cambridge, Massachusetts. London, England.

[4] P. Cousot and R. Cousot. *Abstract Interpretation and Application to Logic Programs.* The Journal of Logic Programming, 13(2–3):103–179, 1992.

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

[6] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. *Evaluation of the domain* `Prop`. The Journal of Logic Programming, 23(3):237–278, Elsevier Science 1995.

[7] A. Cortesi, B. Le Charlier, P. Van Hentenryck. *Combinations of abstract domains for logic programming: open product and generic pattern construction.* Science of Computer Programming 38:27–71, Elsevier Science 2000.

[8] 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.

[9] B. Le Charlier,S. Rossi, P. Van Hentenryck. *Sequence-Based Abstract Semantics of Prolog.* Theory and Practice of Logic Programming 2(1):25–84, Cambridge University Press, January 2002.

[10] B. Le Charlier, C. Leclère, S. Rossi, and A. Cortesi. *Automated Behavioural Verification of Prolog Programs.* Proceedings of APPIA-GULP-PRODE97, Moreno Falaschi, Marisa Navarro, Alberto Policriti editors, Jun. 1997, Grado (Italy), pp189-200

[11] B. Le Charlier, C. Leclère, S. Rossi, and A. Cortesi. *On the Design of an Automatic Tool for Prolog Program Verification.* Proceedings of the Workshop on Tools and Environments for (Constraint) Logic Programming97, Oct. 1997, Port Jefferson (USA)

[12] B. Le Charlier, C. Leclère, S. Rossi, and A. Cortesi. *Automated Verification of Behavioural Properties of Prolog Programs.* Proceedings of ASIAN97, Asian Computing Science Conference (to appear in Springer LNCS), Dec. 1997, Kathmandu, Nepal

[13] B. Le Charlier, C. Leclère, S. Rossi, and A. Cortesi. *Automated Verification of Prolog Programs.* Research Paper, Dipartimento di Matematica Pura ed Applicata, Universita' degli Studi di Padova, Italy, March 1997.

[14] B. Le Charlier, C. Leclère, S. Rossi, and A. Cortesi. *Automated Verification of Prolog Programs.* The Journal of Logic Programming 39:3–42, Elsevier Science 1999.

[15] 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.